Session 1: Freelancing and Remote Work Bootcamp

Table of Contents

Freelancing And A Remote Work Boot Camp 13
Abstract 13
1. Objective Of Boot Camp 14
1.1. ..... What is GitHub And Its Purpose in Boot Camp? 14
2. Design A Simple Html Page 14
2.1. Need Of GitHub OR GitHub Repository In Html 15
2.2. How To Sign Up For GitHub? 15
3. How To Allow Anyone To Change Code On GitHub? 28
3.1. Use Of Surge With Github Repository 36
3.2. Creation Of Account On Surge 38
4. Automatic Deployment Of Code on GitHub Repository 41
Result 48

Session 2: React Base App

Table of Contents

1. What Is React? 50
1.1. ..... Development Of React Base Application 51
1.2. Why React? 64

Session 3: React_Tutorial

Table of Contents

Learning React: 67
1.1 Simple “Hello World To React “ 67
1.2. Adding Properties To Simple Hello world 70
2.1. Different ways of writing a function 73
3. Tree Working 74
3.1 This is how a tree work. 75
4. CSS Style implementation 76
5. Repository Creation for Code And Its implementation 77

Session 4: React_Tutorial

Table of Contents

1. What is State? 86
1.2. ..... Example Of State ;Let's Look Into Deep86
2. How To Import useState 87
3. Render Based on State 87
4. Creation Of A Component 88
5. Import useState Inside React 90
6. Change The State When You Click The Button 93
How setLit works 93
6.2. Practical: Change The State When You Click The Button..... 94
7. Provide Function As An Arrow In The Event 95
8. Change The Background Color..... 97
8.2 CSS Style Implementation 98

Session 5: React's_Context_API_and_UseReducer_React_Hook

Table of Contents

1. Transfer of Data In A Tree Having Single property..... 103
2. Context.js component..... 106
3. Reducer..... 111

Session 6: Expense_Tracker_App

Table of Contents

1. React Expense App; how To Build That Application 118
a. Header.js120
b. Balance.js123
a. AccountSummary.js124
d. TransactionHistory127
e. AddTransaction.js131
f. context135

Session: 7 React

Table of Contents

1. Synchronous And Asynchronous Coding 144

2. Synchronous Coding Example 144

2.1. ..... How synchronous Coding Works145

3. Asynchronous coding 146

3.1. ..... Coding Example of Asynchronous146

3.2. ..... How Does Asynchronous coding Work147

3.3. Example..... 148

4. Callback Hell 150

5. Promise 153

5.1. Example 153

5.2. Example 155

6. Aynchronous function Enhance Code Beauty And Simplicity than Promises 157

6.1. ..... Example158

Session 8: Get Data in JavaScript and React

Table of Contents

1. How To Fetch Data..... 161

1.1. .....GetData161

1.2. ..... Post Data163

2. Postman 164

3. Fetch API Data Using UseEffect 165

3.1 Component With Multiple Properties 166

Session 9: Material UI & Charts.js

Table of Contents

1. Material UI Decreases CSS Coding 170
2. UI Material Needed For Installation 171
npm 171
Roboto Font 172
Font Icons 172
SVG Icons 172
Button 172
Contained Buttons 172
Text Buttons 173
Outlined Buttons 173
Handling clicks 174
Upload **B**utton 174
Sizes 174
Buttons with icons and label 174
Icon Buttons 175
3. Creating Button Using UI Materials 175
4. Grid 178
Spacing 178
Fluid Grids 179
4.1. Basic Grid Creation 180
5. App Bar 181
6. Bar chart 183

Session 10: Material UI & Charts.js

Contents

1. Development Of COVID 19 Tracker App 187

Session 11: React Router v6 and Web Animation API

Table of Contents

1. Add Router Into Application(Developing An Application)..... 198
2. Shoe Store 212

Session 12: Project 4A and 4B without React

Table of Contents

1. What Is Animation? 220
2. Web Animations API Concepts 220
2.1. ..... Core Concepts221
3. Uses 222
4. Project 4A And 4B Without React..... 223

Session 12(A): Project 4A and 4B without React

Table of Contents

1. Web Animation API Without Using React..... 234
2. Web Animation Using App React 244

Session 13: Programming TypeScript

Contents

1.Defining TypeScript……255
2. Transpiler vs Compiler……..……..255
3. Why TypeScript Not java……258
3.2. TYPE SYSTEM……260
4. TYPE……266
5. STRUCTURAL TYPING……267

Session 13(A): Programming TypeScript

Contents

1. Creating React Quiz App(TypeScript) 272

Session 13(B): Programming TypeScript

Table of Contents

Quiz React App 291

Session 14: Progressive Web Apps (PWA) and Deploying

Table of Contents

1. Progressive Web Application 306
2. Register A Service Worker 310
3. Build A React Application 313

Session 14(A): Progressive Web Apps (PWA) and Deploying

Table of Contents

Push Notification Using PWA With Service Worker 321

Session 15: Test_React_Components_in_TypeScript

Table of Contents

Testing 342
How to Write Test Cases 344
Travis CI 348

Session 15(A): Test_React_Components_in_TypeScript

Table of Contents

Basic Pomodoro Timer App-Test Driven React App 354

Session 16: Intro to GraphQL

Table of Contents

Basics Tutorial - Introduction 379
GraphQL - A Query Language for APIs 379
The Schema Definition Language (SDL) 379
Fetching Data with Queries 380
Basic Queries 380
Queries with Arguments 381
Writing Data with Mutations 381
Realtime Updates with Subscriptions 382
Defining a Schema 383
Big Picture (Architecture) 385
Set Up Apollo Server 400

Session 16(A): GraphQL_in_Detail

Table of Contents

Queries And Muatation :Apollo Server & Apollo Client 408
1. Apollo Server 408
2. Apollo Client 413

Session 17: Building_SpaceX_App_with_QraphQL

Table of Contents

Building Space X App By Using React , TypeScript,GraphQL 418

Session 18: Introduction_to_Redux

Table of Contents

1. Using Redux with React Hooks 438

2. Redux Toolkit 449

Session 18(A): Async_Redux_Toolkit

Table of Contents

1. CreateAsyncThunk 455
2. Building API Mocking Servers with Mirage 463
3. Setting Up Redux For Use In A Real-World Application 467

Session 18(B): Redux_Toolkit_and_Mirage

Table of Contents

1. How We Can Create Our Server On Front End Is Introoduce By Mirages 477
2. Build Complete frontend Features Even If Your API Doesn't Exist With the Helps Miragejs 478

Session 19: Forms_with_Formik_and_Yup

Table of Contents

1. Working With Forms 486
2. Overview: What is Formik? 489
3. Schema Validation with Yup 494

Session 19(A): Multi-Page_Forms_using_Formik

Table of Contents

MultiSteps Forms:Creating Linear -Alternative Level 503

Session 20: Component Driven Development (CDD)

Table of Contents

Introduction To Component Driven Develoment 516
Starting Out:StoryBook 534

Session 20(A): Component Driven Development (CDD) Part 2

Table of Contents

Bit for React 552

Session 21

Table of Contents

What are modern applications and how to develop them? 568
Modern App: 568
Monolithic App: 568
Difference between monolithic 568
and modern app: 568
Modern Application 568
Modern App Developing 569
Three Client site framework 569
AWS Application 569
Client site rendering 573
Server Site Rendering 573
Static Site Generation 573
Static site generation is easily accessable to server? 573
Jamstack 574
Gatsby 574
What is Contentful? 574
Netlify 574
Serverless Lamda Function 575

Session 22

Table of Contents

Gatsby 576
What is Gatsby ? 577
How to Install Gatsby? 577
Gatsby Typescript Helloworld 579

Page Component Liniking 583
Sub Components 584
How To Built a Sub Component ? 585
Layout Components 587
Global Styles 591
Component Scoped CSS 593

Session 23

Table of Contents

Programmatically Create Pages 596
Where Does Logic Goes In Jamstack? 596
Difference between Build time & Run time 596
Build Time 596
Run Time 596
Client Only Route 599
Material UI 603

Session 24

Table of Contents

Fetch Data Build Time Contentful 607
How to fetch data? 607
Fetch Richest Contentful 611
Dotenv 619
Programmatically Create Pages 620

Session 25

Table of Contents

Netlify HelloWorld 623
React Netlify Helloworld 631
Gatsby Develop 638
React Netlify Functions Helloworld 642

Session 26

Table of Contents

Bootcamp 647
How to add multiple libraries? 651
FAUNADB 651
What is FaunaDb? 651
What is interface ? 652
Create key 658

Session 27

Table of Contents

Gatsby Netlify Function Faunadb 663
Gatsby Netlify Function Faunadb Formik 663

Session 28

Table of Contents

Gatsby Netlify Functions Apollo Server Lambda 667
Gatsby Netlify Functions Apollo Server Lambda Faunadb 668

Session 29

Table of Contents

Jamstack Todo 676

Session 30

Table of Contents

Project 4 700
What is bookmarking application? 700

Session 31

Table of Contents

Project 3 713
Creation Of Virtual Lolly 713
What Is a Virtual Lolly? 713
Programming a Virtual lolly 714

Session 32

Table of Contents

Learn To Build An Ecommerce Website and app using JAMstack. 722
What is Ecommerce? 722
What You Need To Built An eCommerce Website 735
WordPress OR WooCommerce 739
SEO 746

Session 33

Table of Contents

Payments Using Stripe Checkout 752
Project “0” (Gatsby Stripe Checkout) 752

Session 34

Table of Contents

Payments With Stripe API 765

Session 35

Table of Contents

Building Shopping Cart Using Snapcart 785
Project - 2 (Gatsby Snipcart) 785

Freelancing And A Remote Work Boot Camp

Abstract

This tutorial is going to teach the importance of Boot camp for free hosting website services by simple knowledge of CSS, java,hmtl.we will take into consideration the GitHub repository in Bootcamp, simple HTML page designing by using the GitHub repository in bootcamp2020. how GitHub repository helps in tracking changes made in code by developers, as well as how to change code and automated code change on GitHub repository

1. Objective Of Boot Camp

Its main objective is to educate ourselves and share knowledge with the community

Basic Knowledge Requirements For Boot Camp

- *CSS*
- **html**
- **javascript**

1.1. What is GitHub And Its Purpose in Boot Camp?

- Code versioning system used by the team for collaborating on software projects.
- Helps in keeping centralized repository code that assists all team members as well as customers.
- It also provides code versioning capability by tracking, changes made by individual.
- It also helps in maintaining history as a result of which the involvement of changes in source code by any team member can be kept into our records

Learning starts from taking baby steps of knowledge, so let's take a grip over html page designing

2. Design A Simple Html Page

2.1. Need Of GitHub OR GitHub Repository In Html

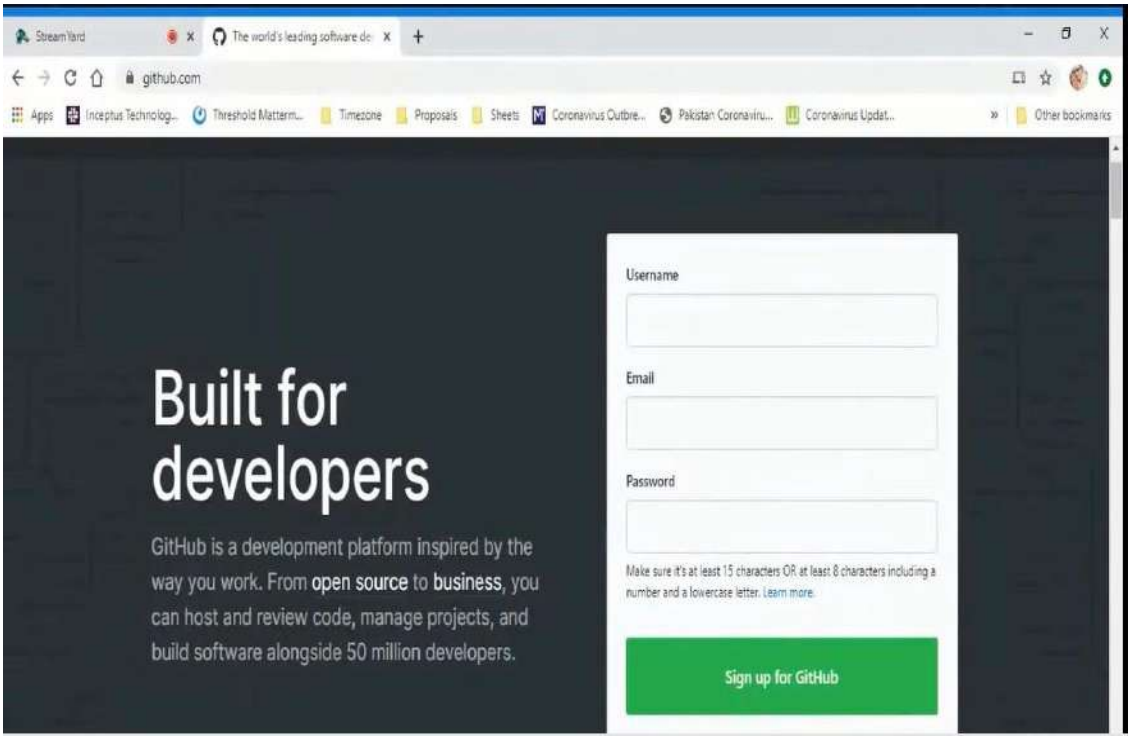
In designing a simple html there is no need of a repository since as if you are making multiple changes to the file, for that purpose to keep versioning control system and to have a knowledge who has made changes we need aa repository GitHub forthat.

2.2. How To Sign Up For GitHub?

There is no need of money to sign up GitHub just you have to put your email and password for that purpose.

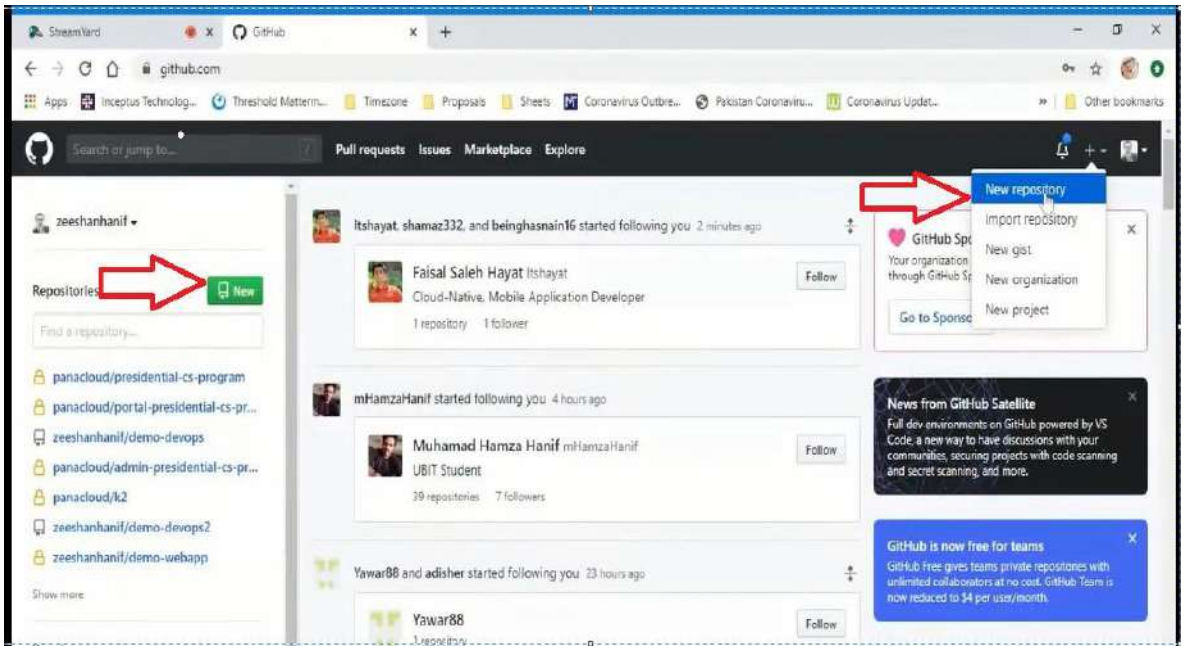
Follow these steps

- go to github.com
- enter your user Name
- enter your email
- enter your password

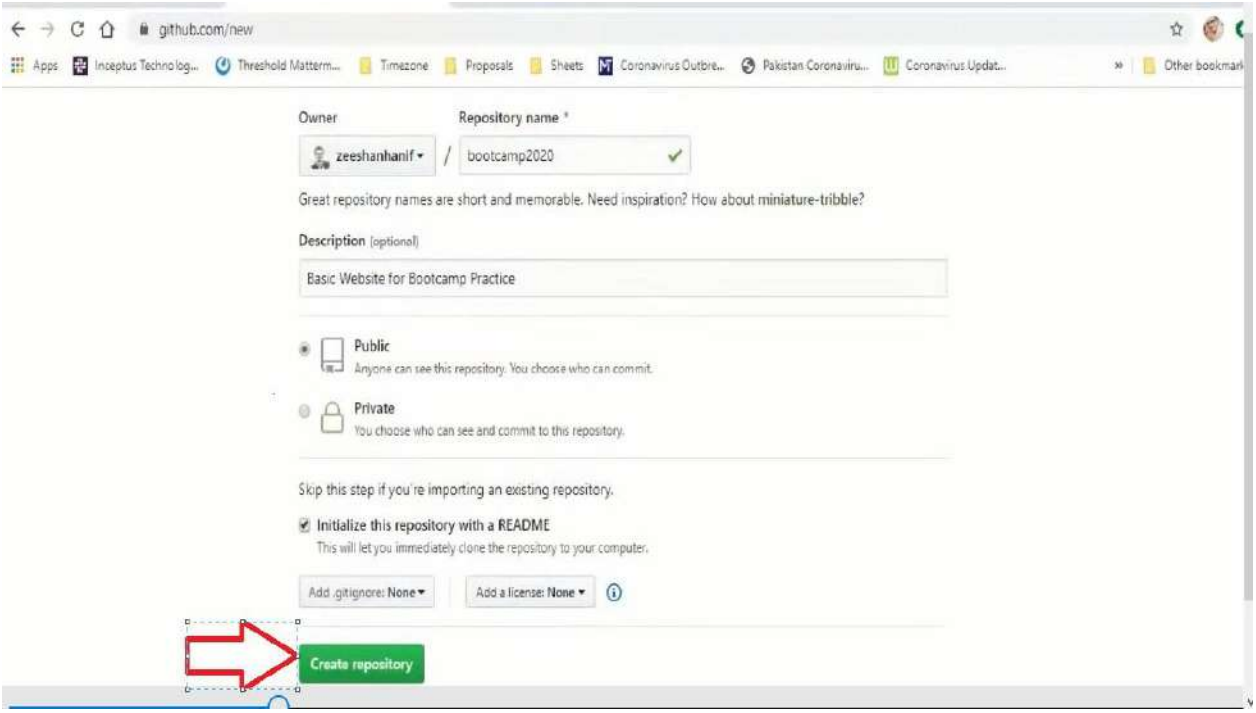


Now after signing up create a repository

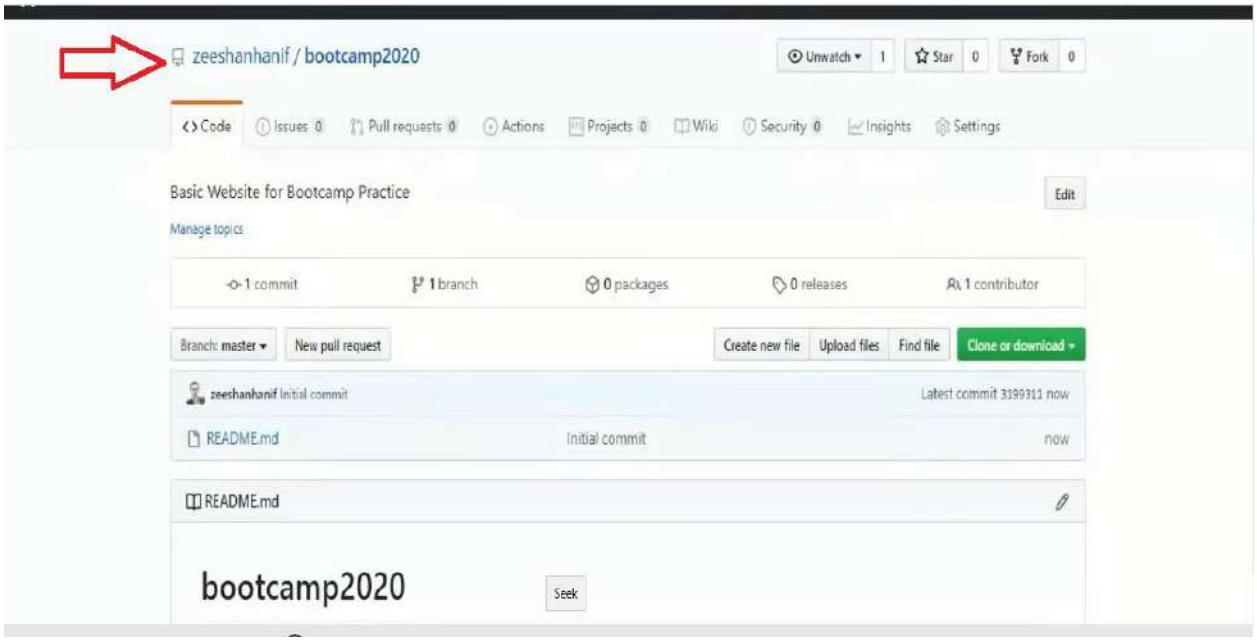
✓ either click at' + 'on the right top with bell icon or 'New' icon at the left top



- Write repository name
- Description
- Click to public
- Click to initialize this repository README
- Finally hit the button press to 'create repository' that's it.



❖ A New GITHUB REPOSITORY has been created as shown.

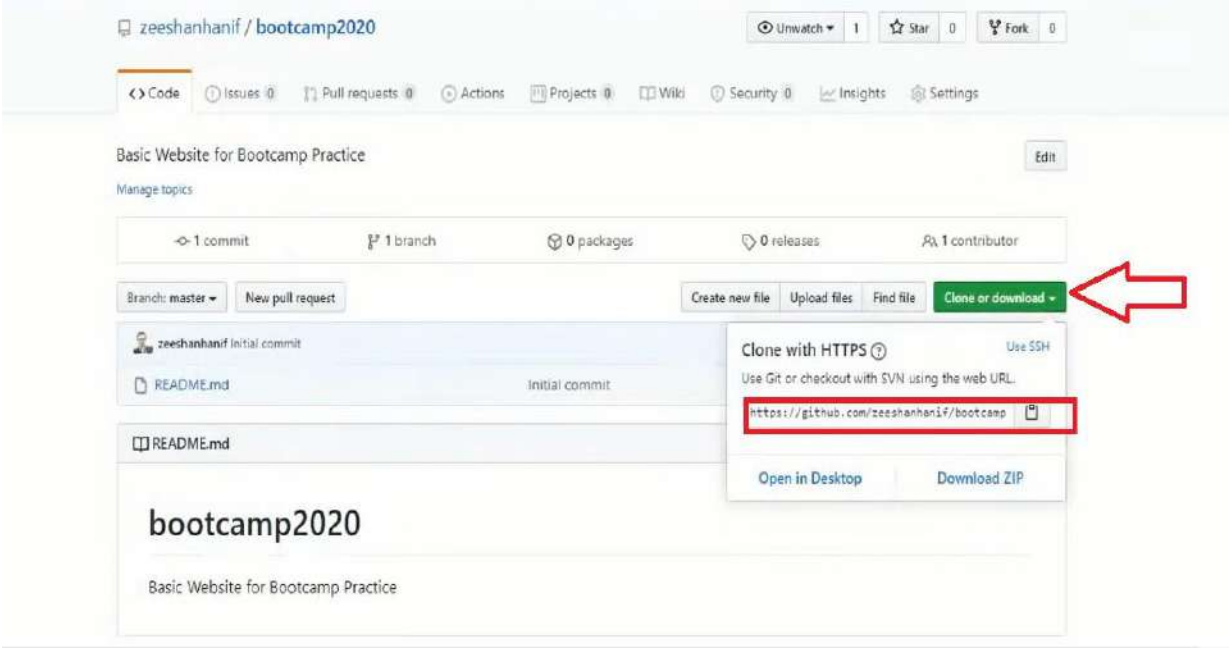


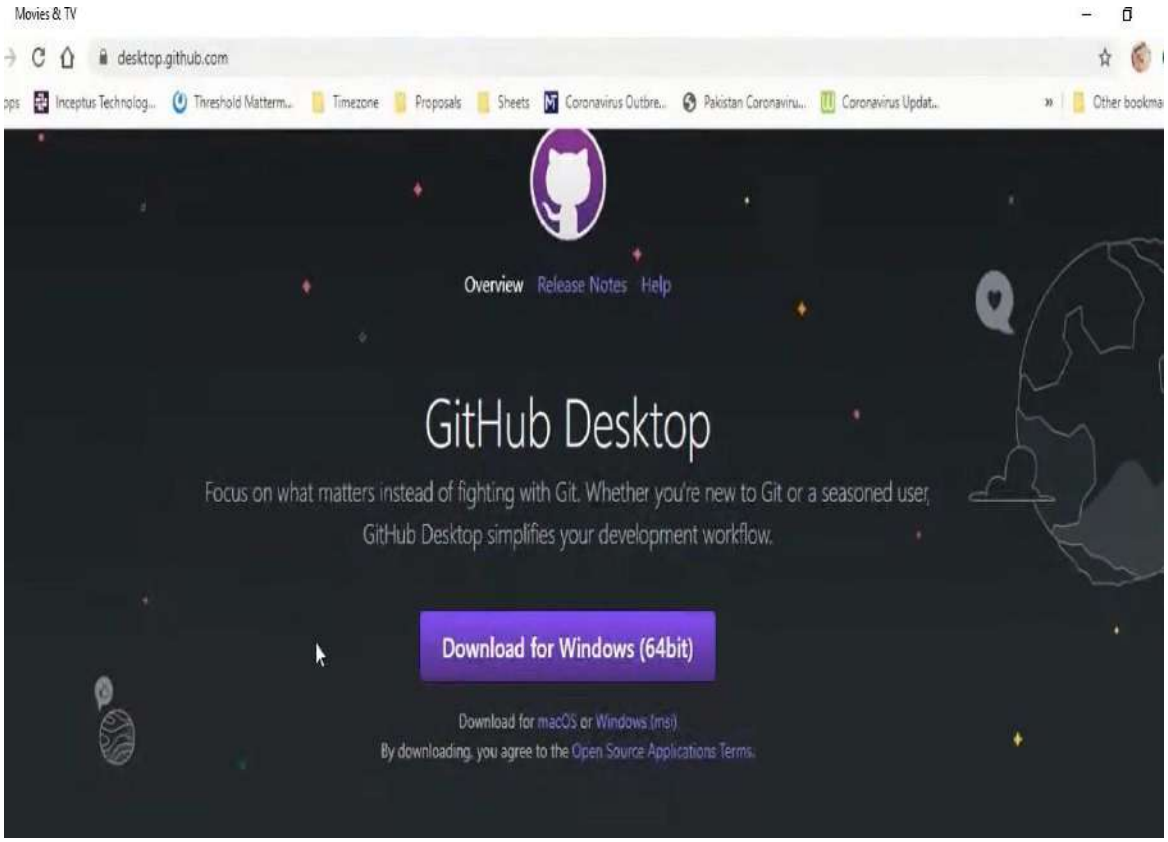
❖ **This is a GitHub repository, to make changes in it you need to copy the clone, do not download just copy its URL in UI tool or**

Terminal option

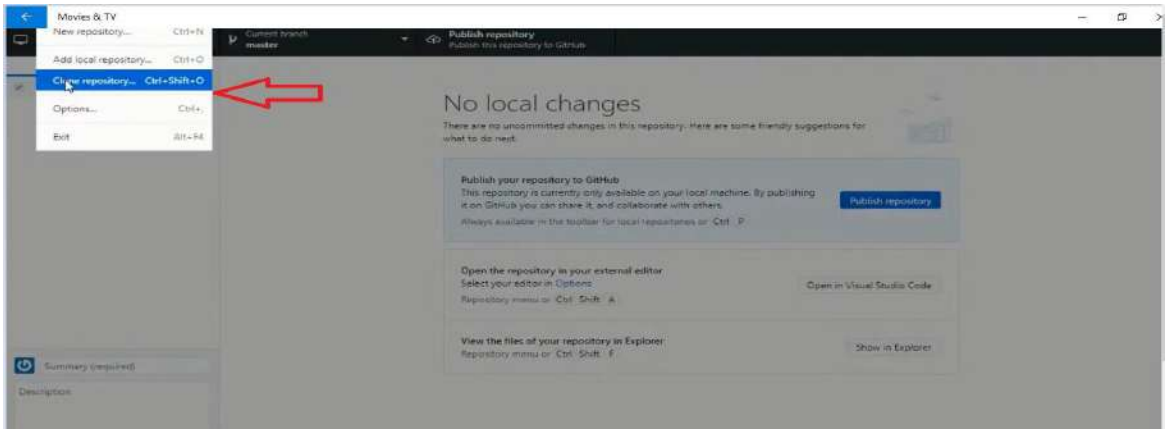
➢ Download GitHub desktop a UI tool for that purpose from "desktop.github.com"

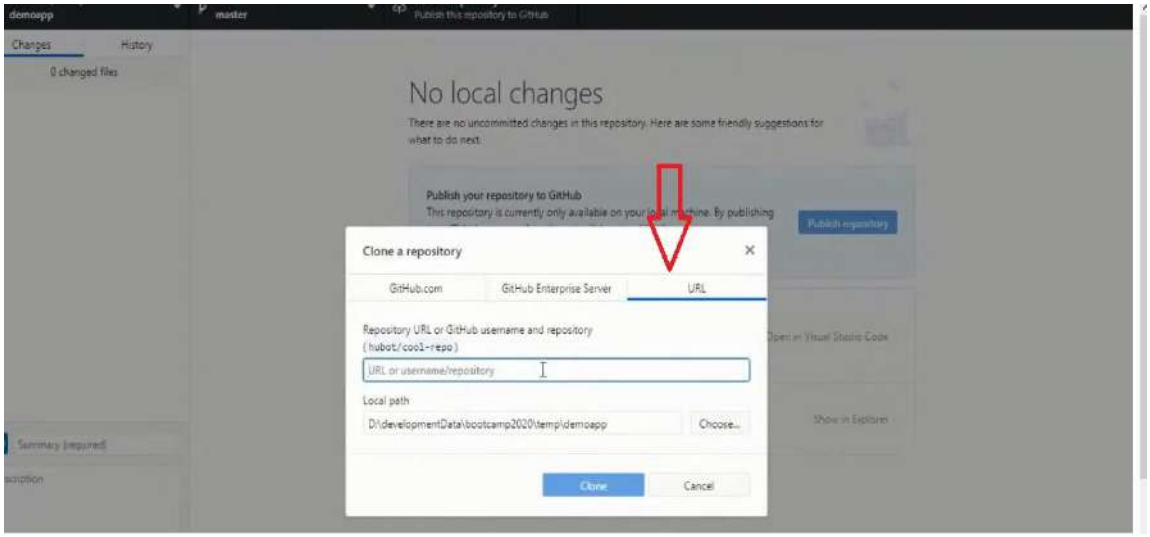
Download GitHub desktop

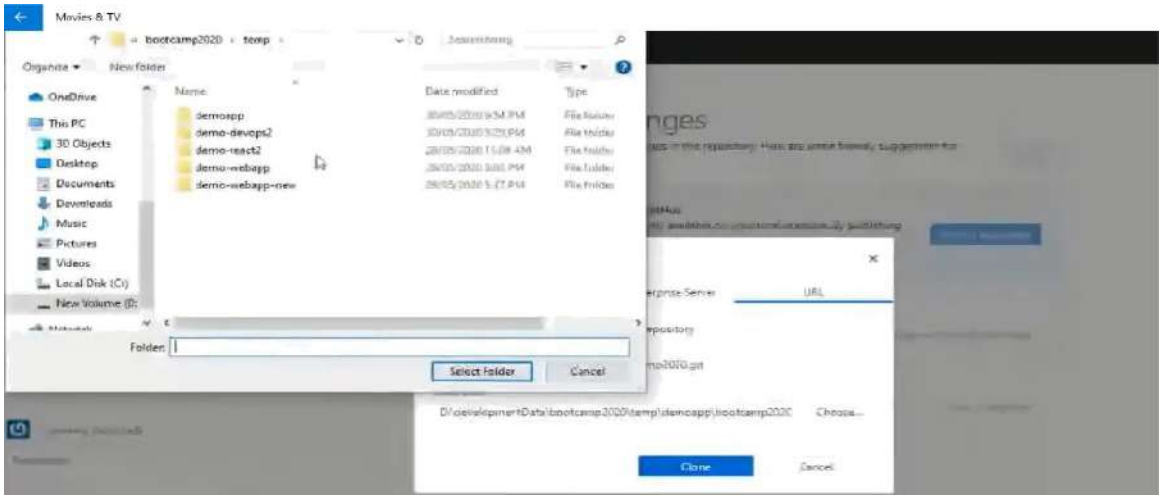




- click to clone repositoryoption
- select URL and paste over there copied URL from GitHub repository
- Select the path from your system you want to clone it
- click to clone now from the file menu
- cloning will be started as shown

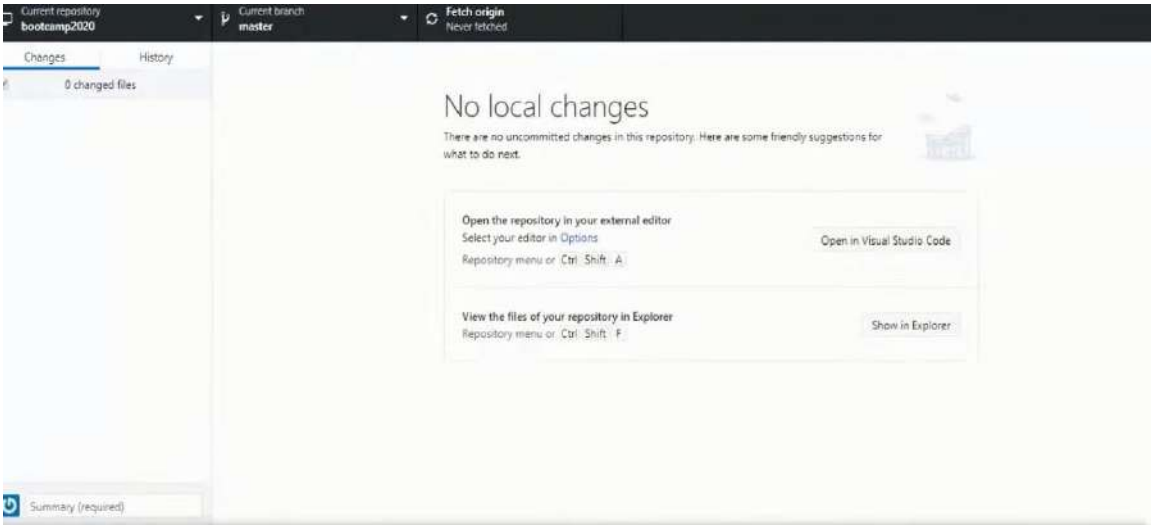


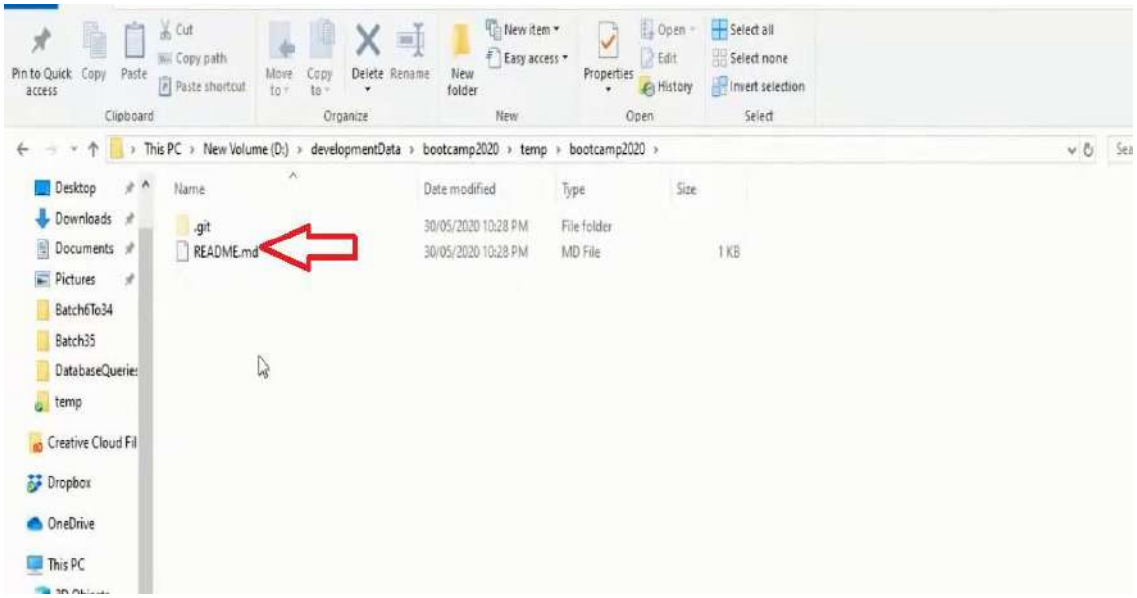




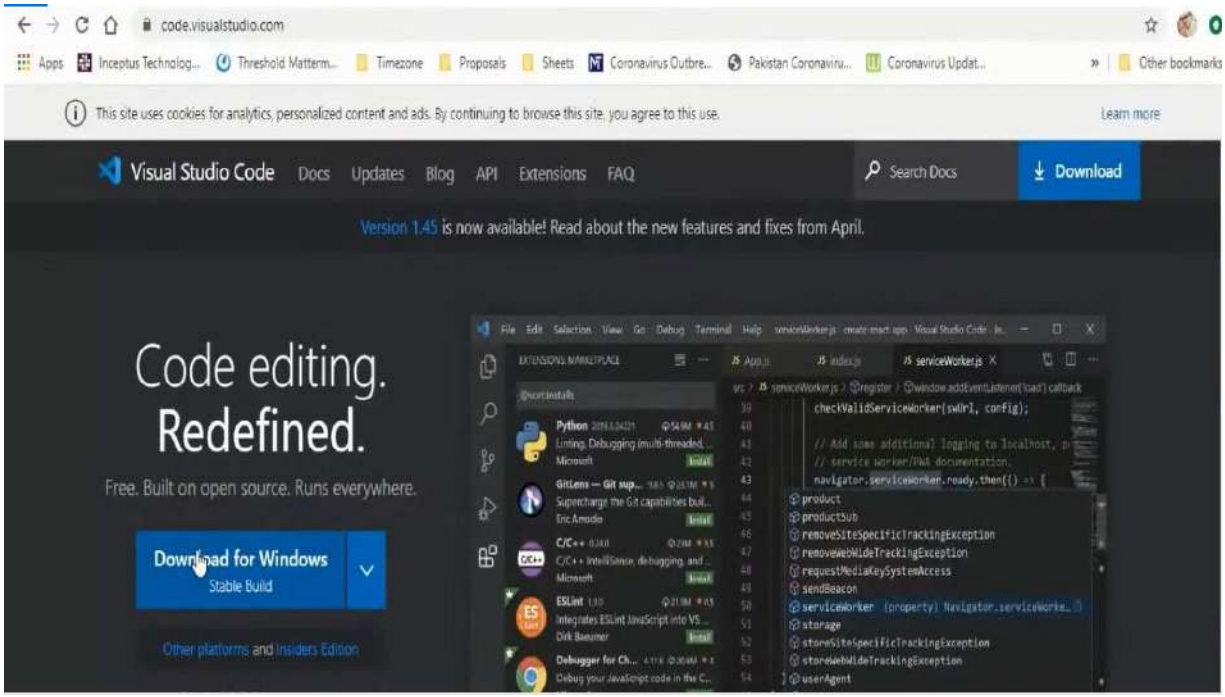
fJ' Cloning bO0tCdITI 2020

- go to your folder where you have installed your Bootcamp 2020
- *go to temp < READ.md and open it*



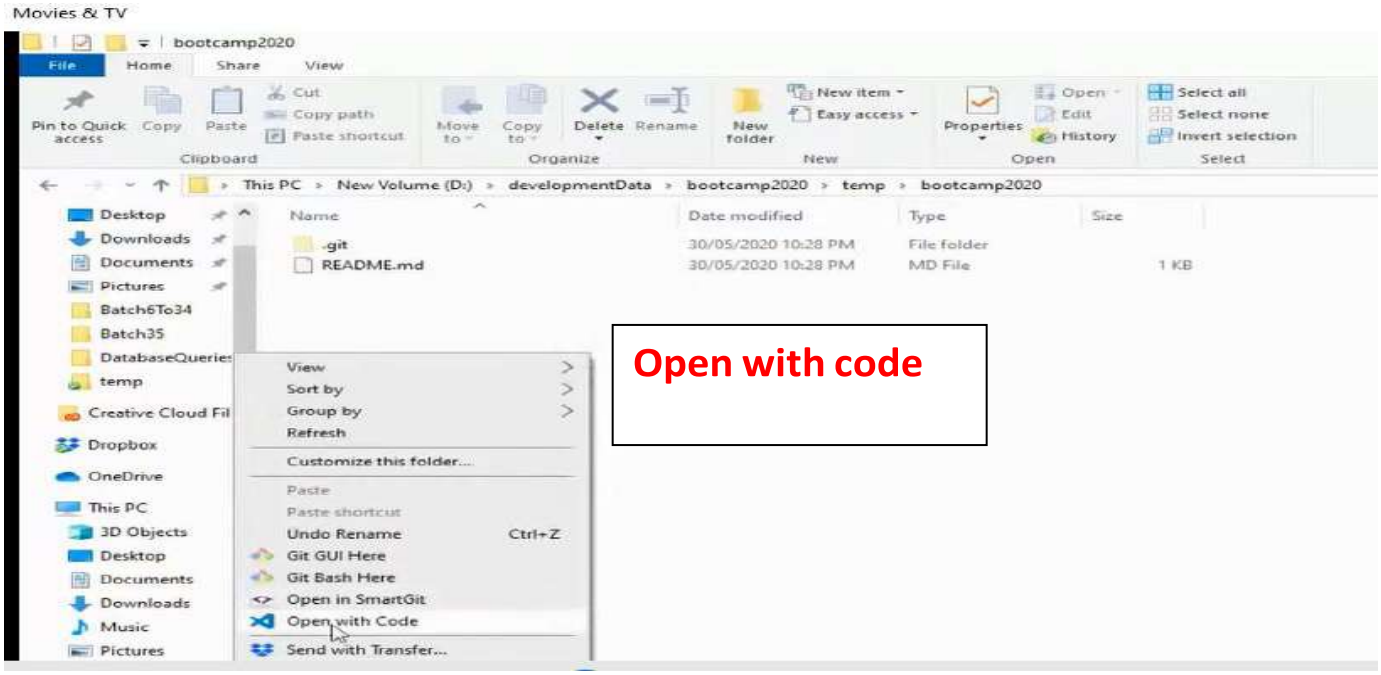


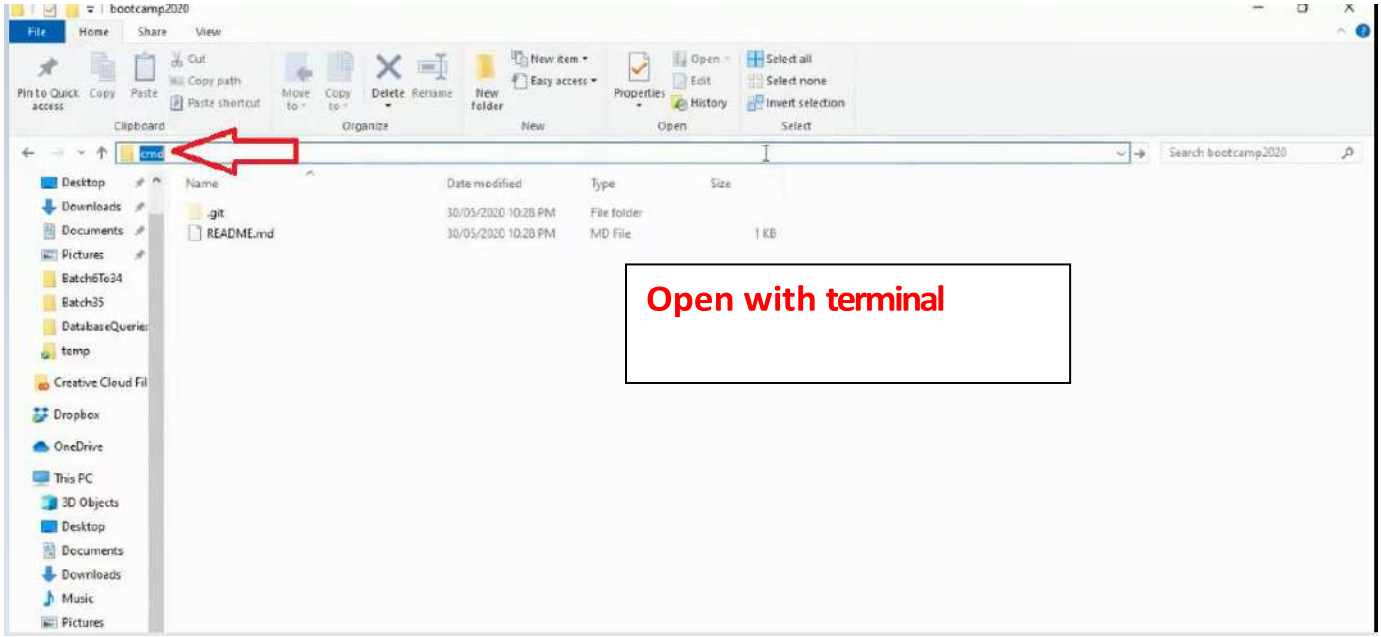
- To open new repository, download visual studio from code.visualstudio.com



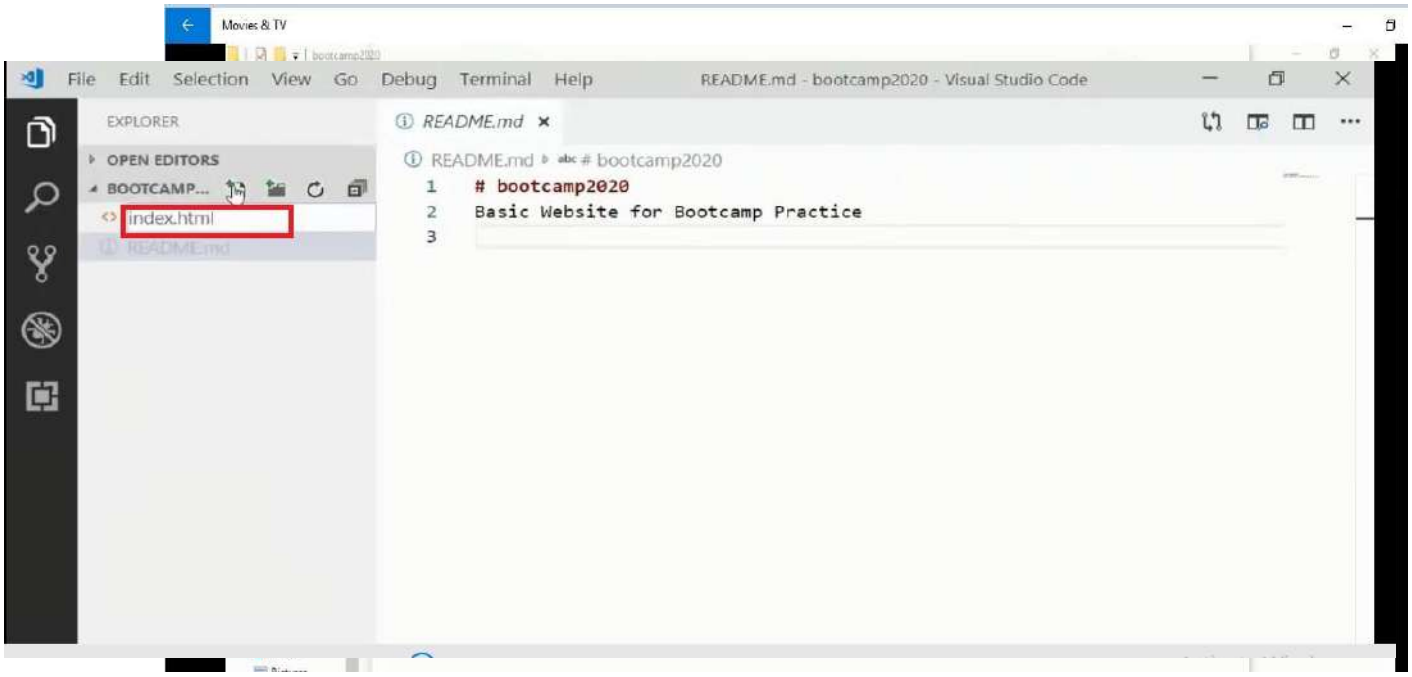
❖ Once you have installed visual studio, then you can open the repository via multiple options.

1. open with code option by right-clicking
2. 2nd option is to go to the terminal by writing "cmd" and open folder and write "code." in it. it will open visual studio

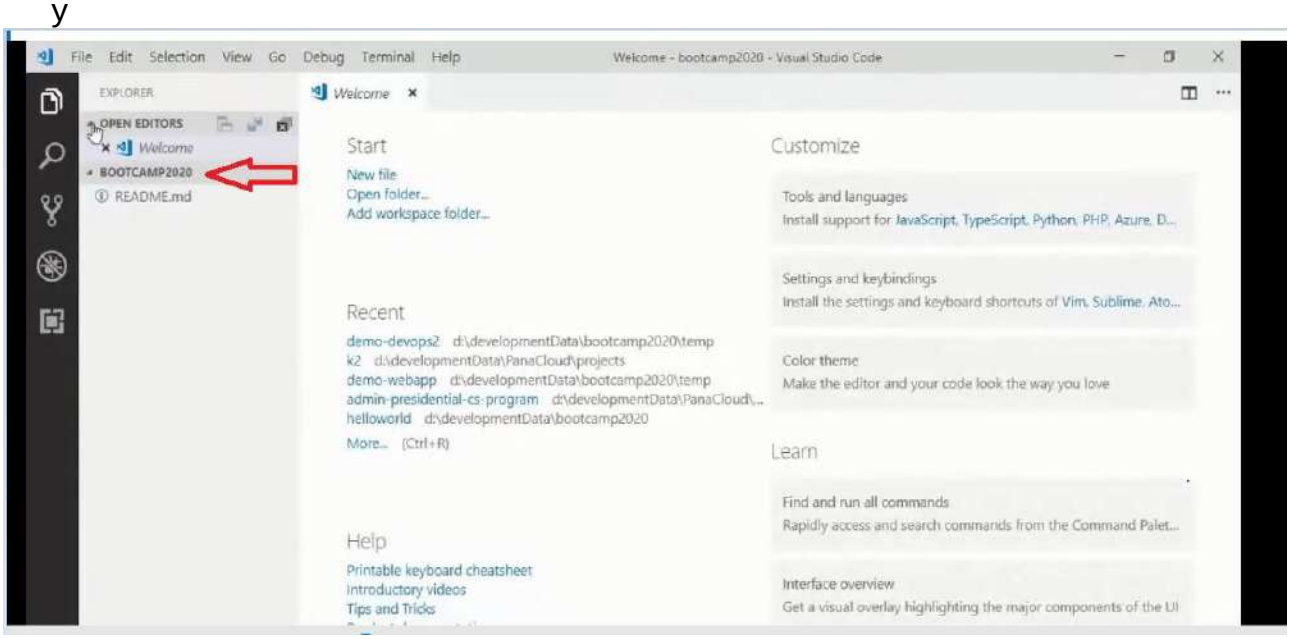




- ***I have selected the 2nd option***

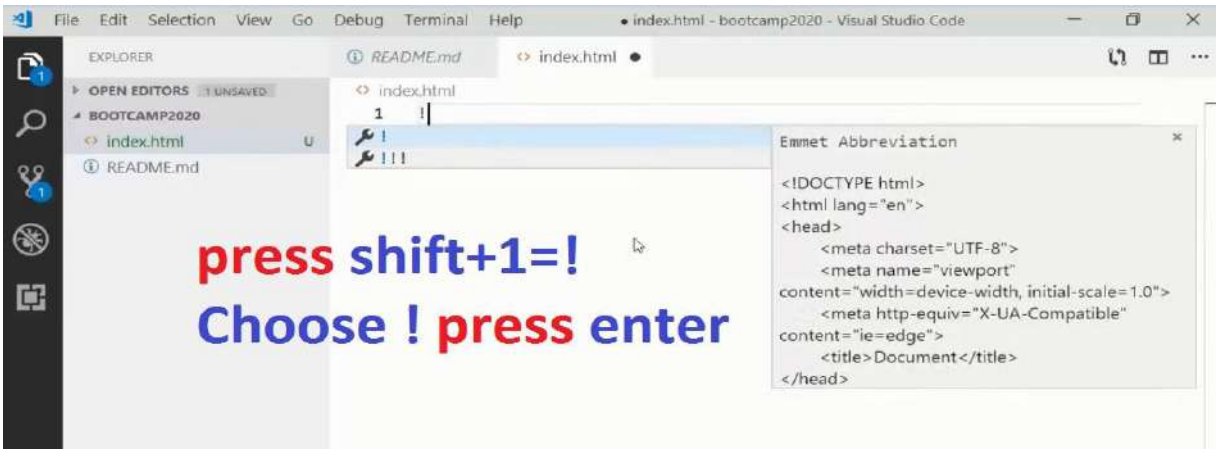


- folder is empty



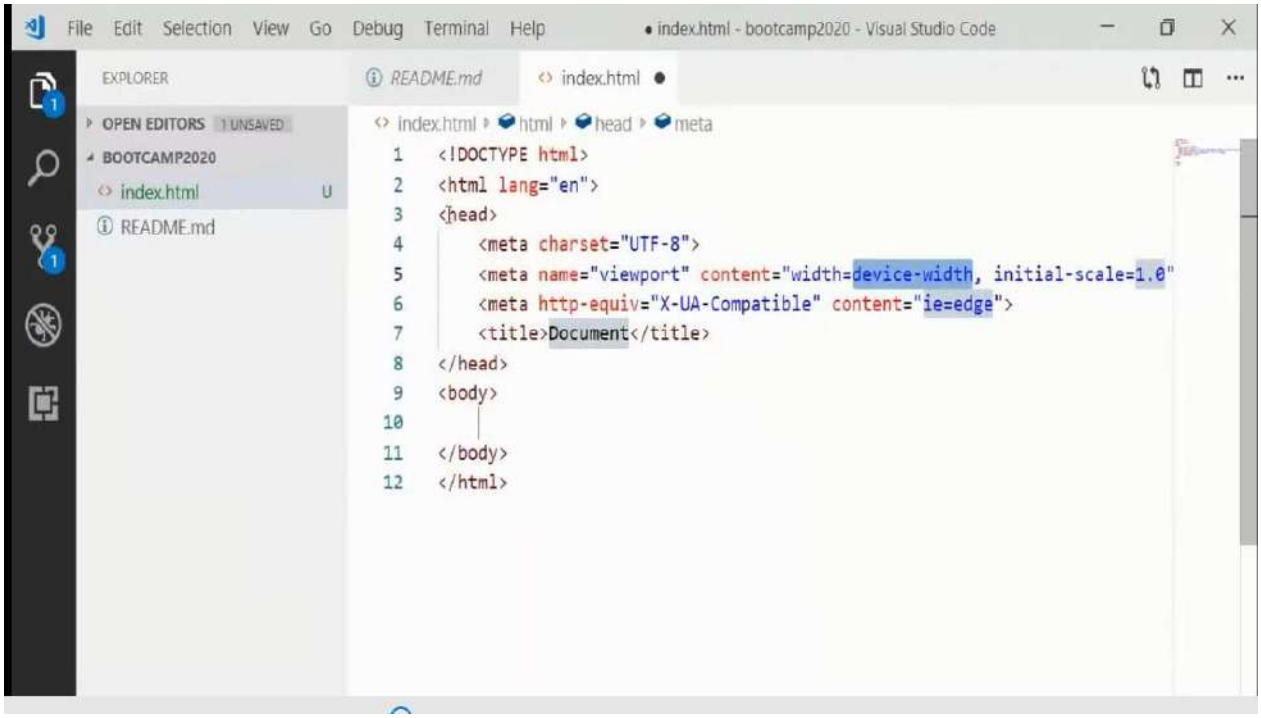
- **for creating Html file just create a new file i.e"index.html" from boot camp2020 folder**

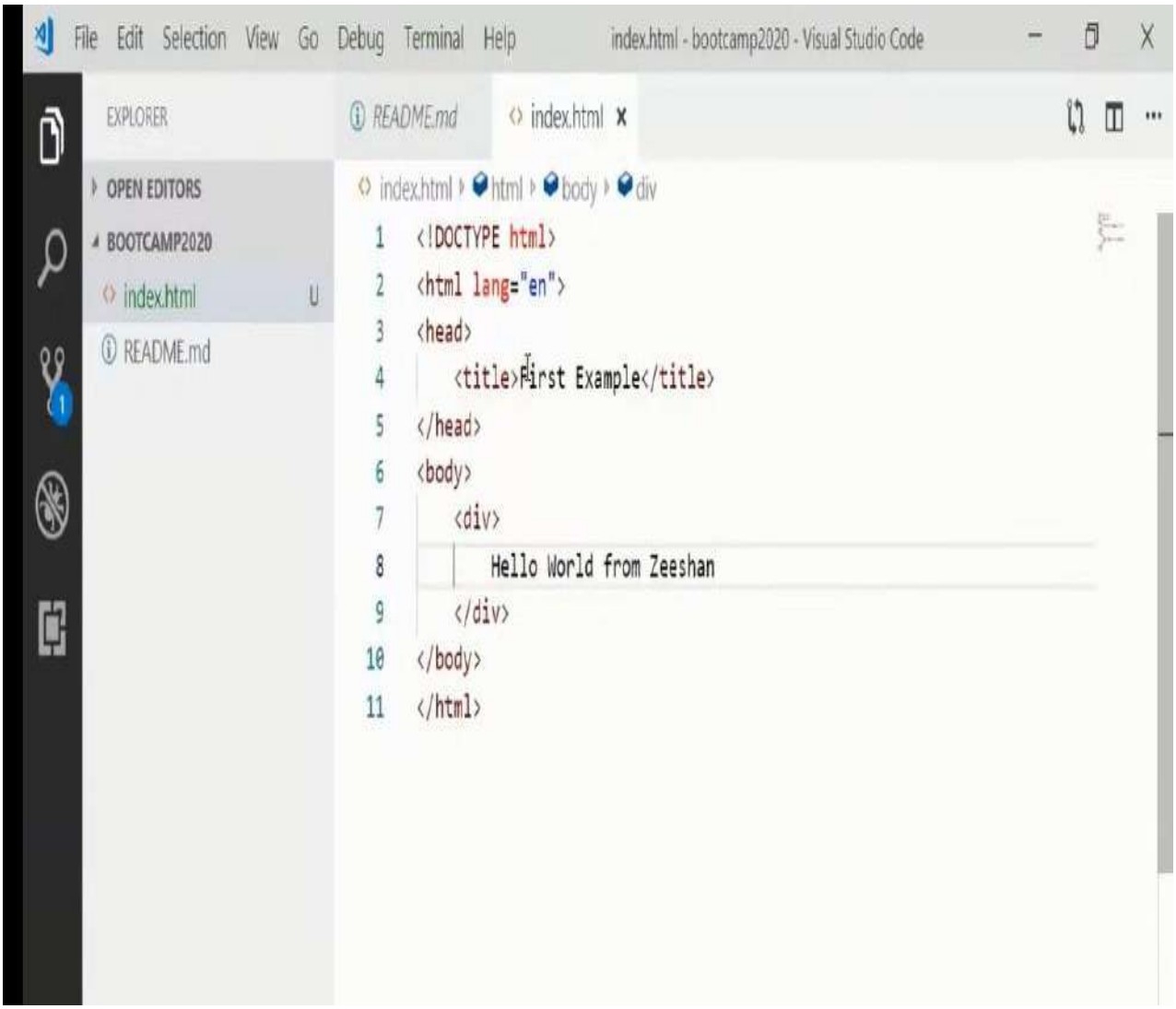
➢ Now to get basic Html structure press "**shift+1=!**" and press **enter.**



Now if you want to write something i.e." Hello World from Zeeshan "
Then in body tag enter the open and close tag of div *i.e.*

```
<body>
    <div>
  "Hello World From Zeeshan "
  </div>

  </body>
```

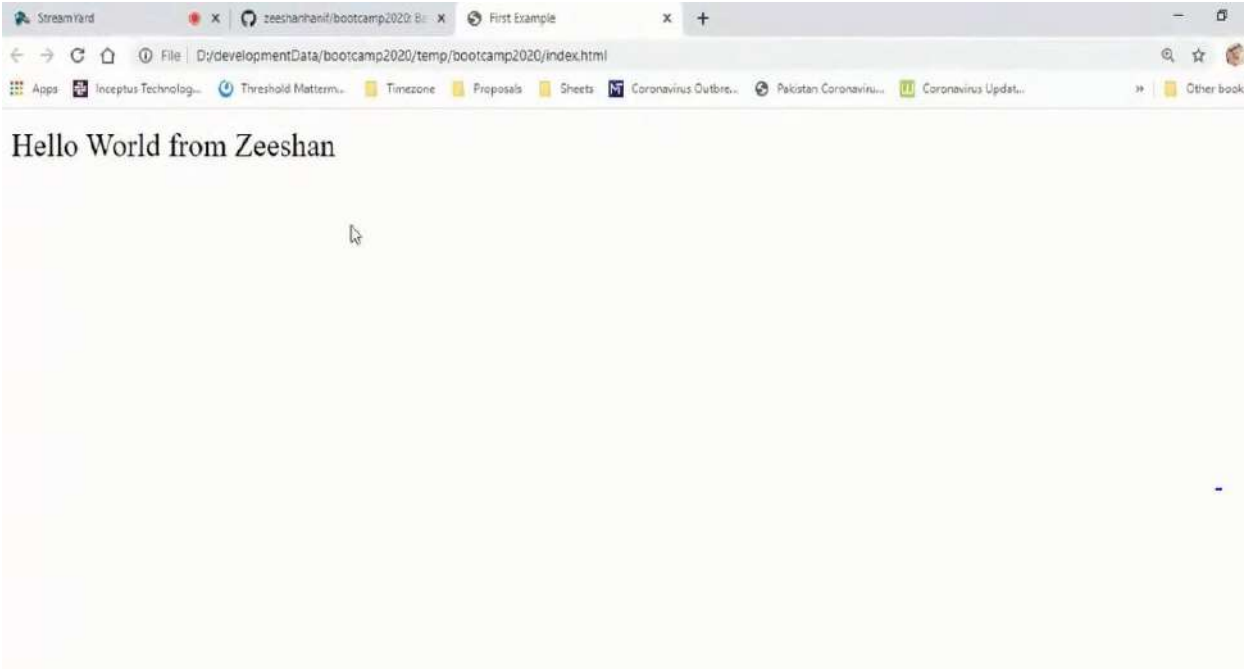



File Edit Selection View Go Debug Terminal Help
index.html - bootcamp2020 - Visual Studio Code
EXPLORER
OPEN EDITORS
BOOTCAMP2020
index.html
README.md
index.html > html > body > div
<!DOCTYPE html>
<html lang="en">
<head>
<title>First Example</title>
</head>
<body>

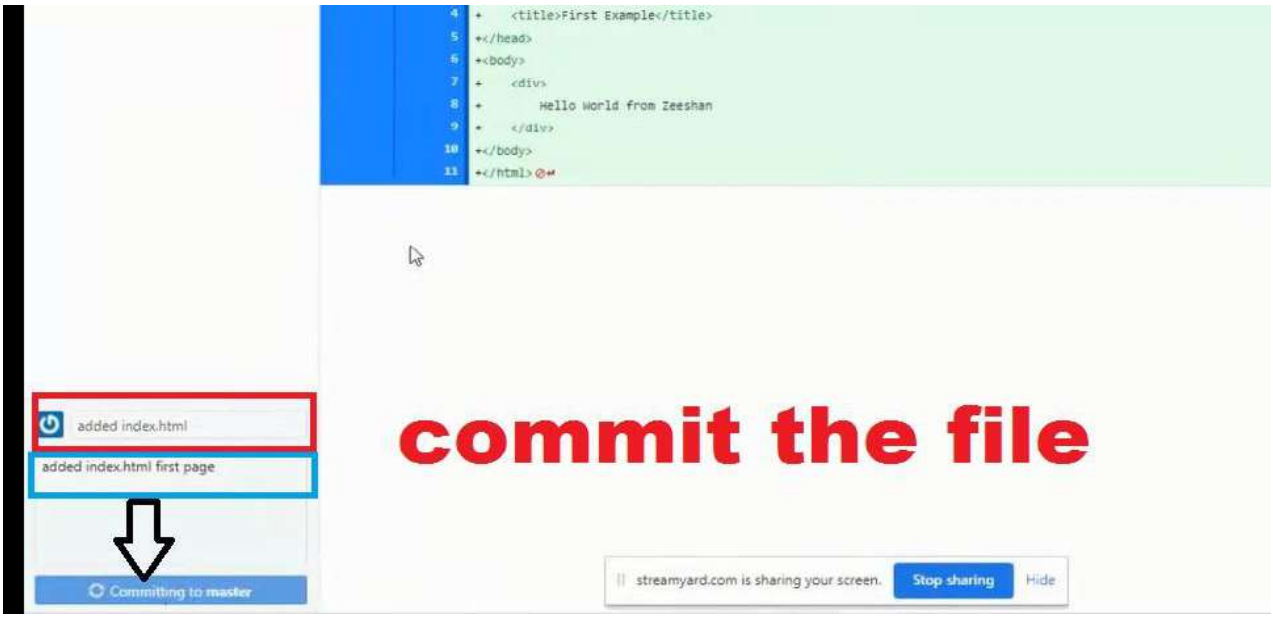
Hello World from Zeeshan

</body>
</html>

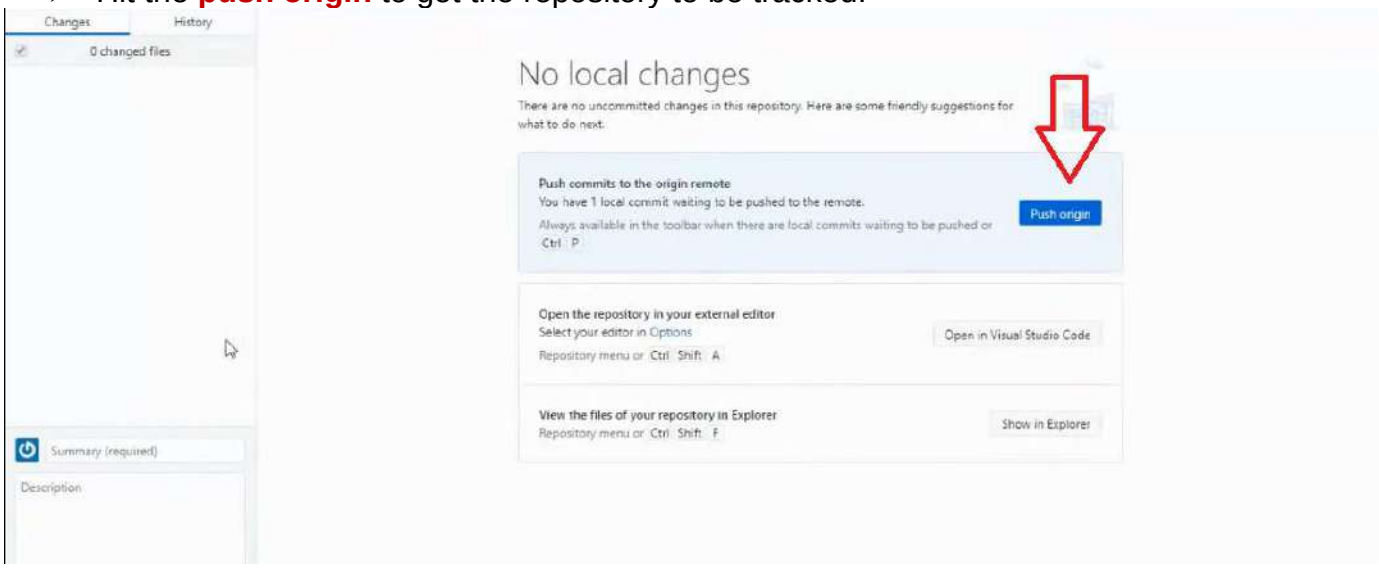
- **Save the file and open it either with internet explorer or chrome and view it.**



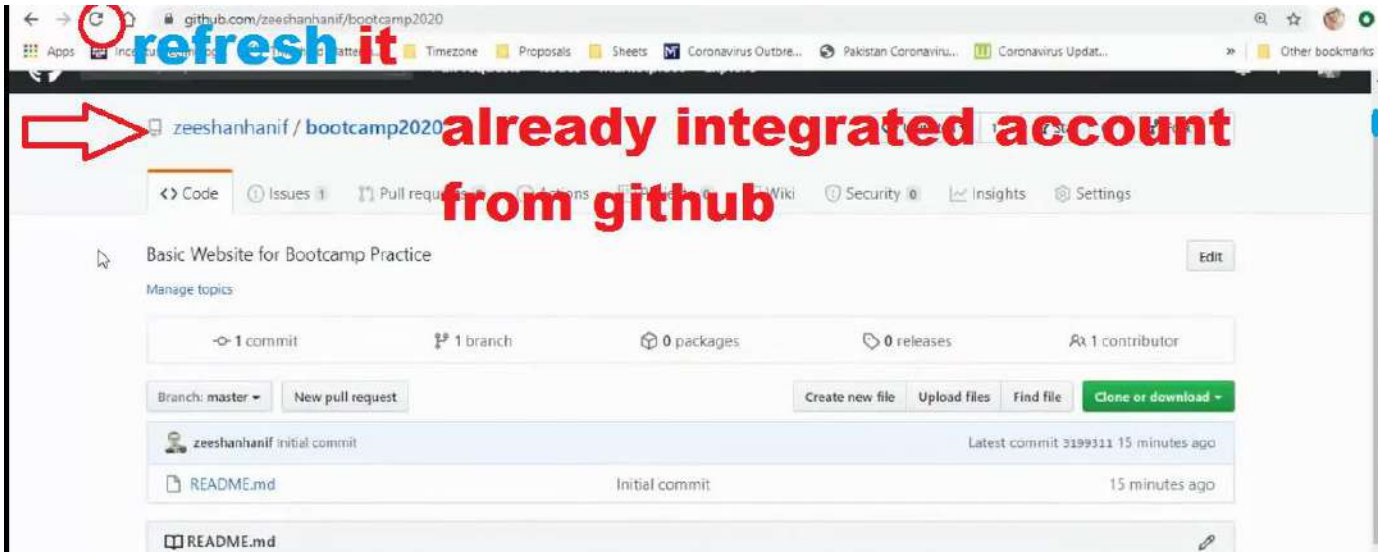
- **Now commit the file to get it to reposit (**it means to register any change with the git you have made. It will track changes or versioning
  - only if you commit a file change)



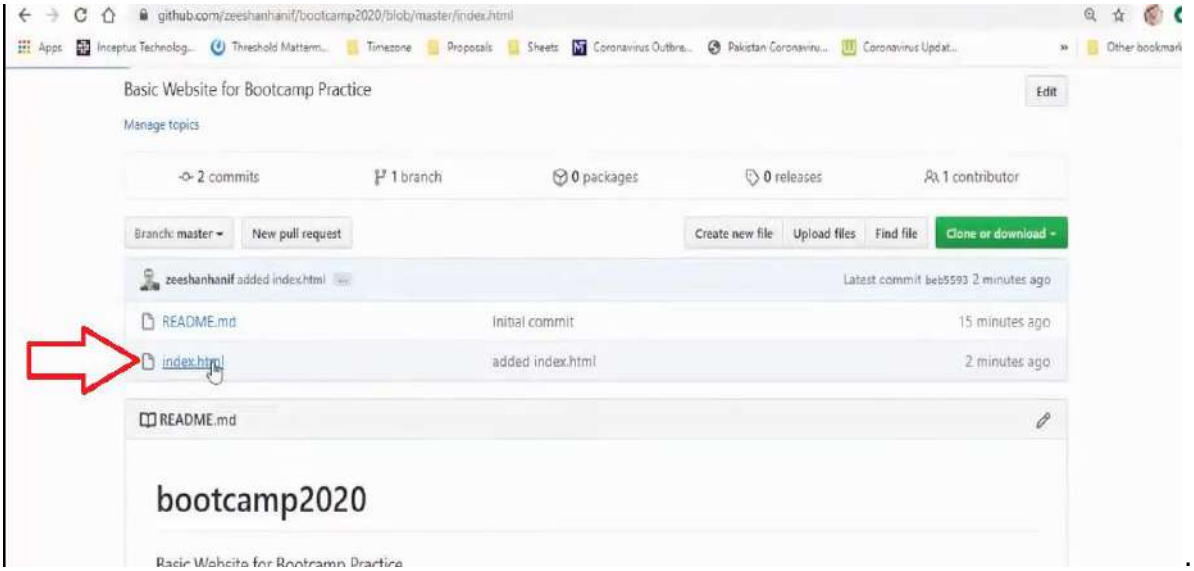
- Hit the **push origin** to get the repository to be tracked.

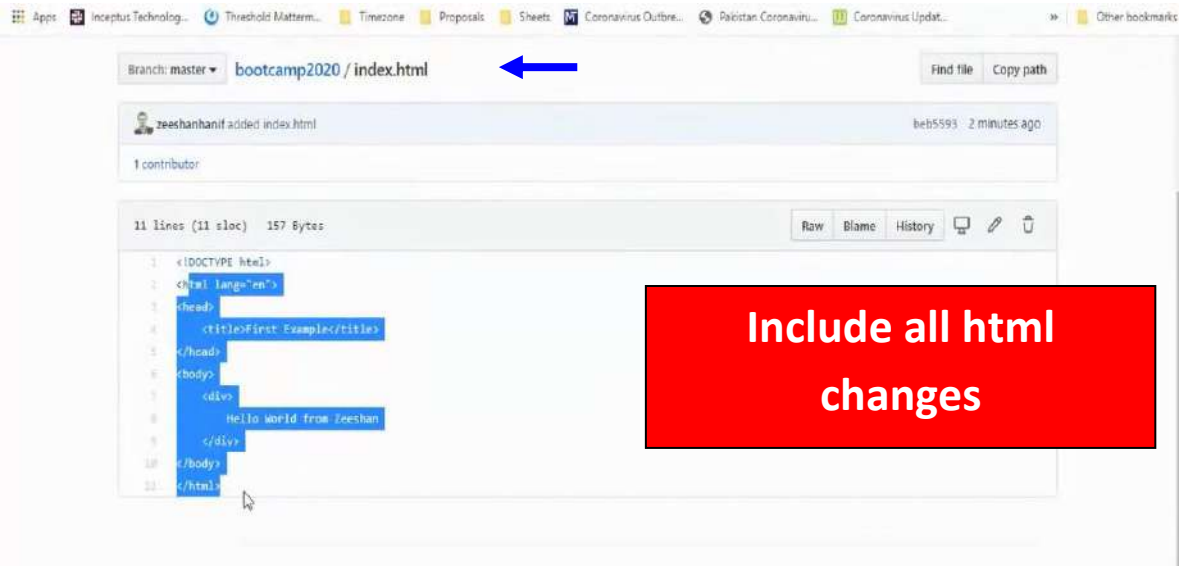


- Since the UI tool **(desktop GitHub)** here used is already integrated with the GitHub repository (i.e Zeshanhabib account), that's why it isn't asking user id and password otherwise it will ask about user id and password.
- Refresh the repository.



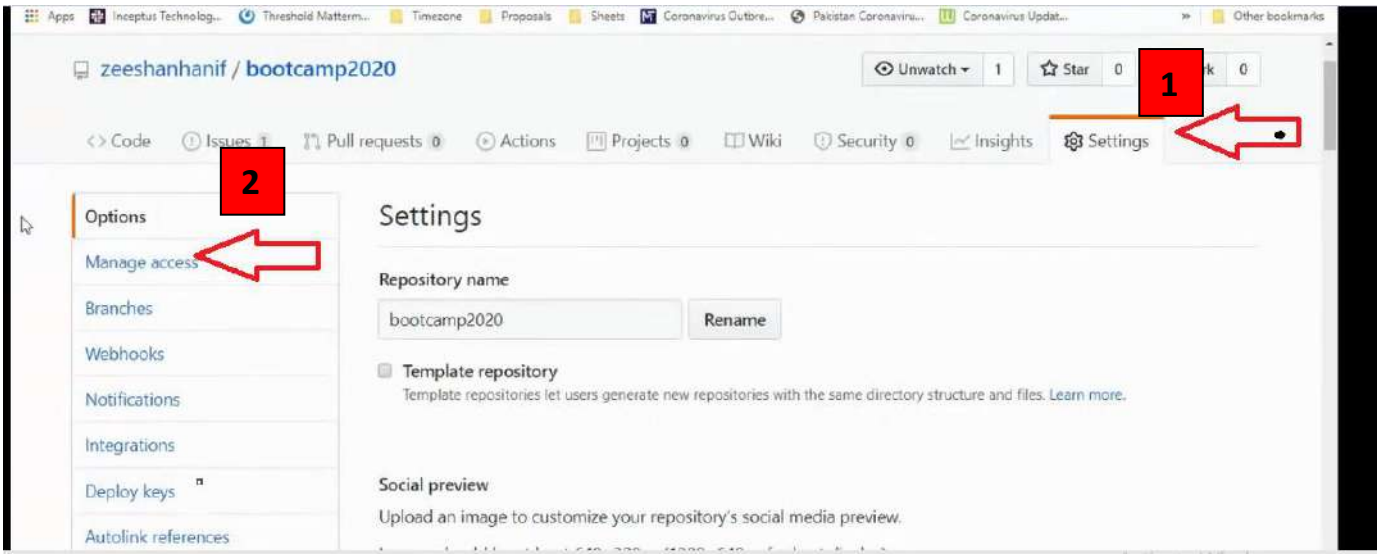
- After refreshing "**index.html**" will be shown in the repository and it includes all changes that are made.



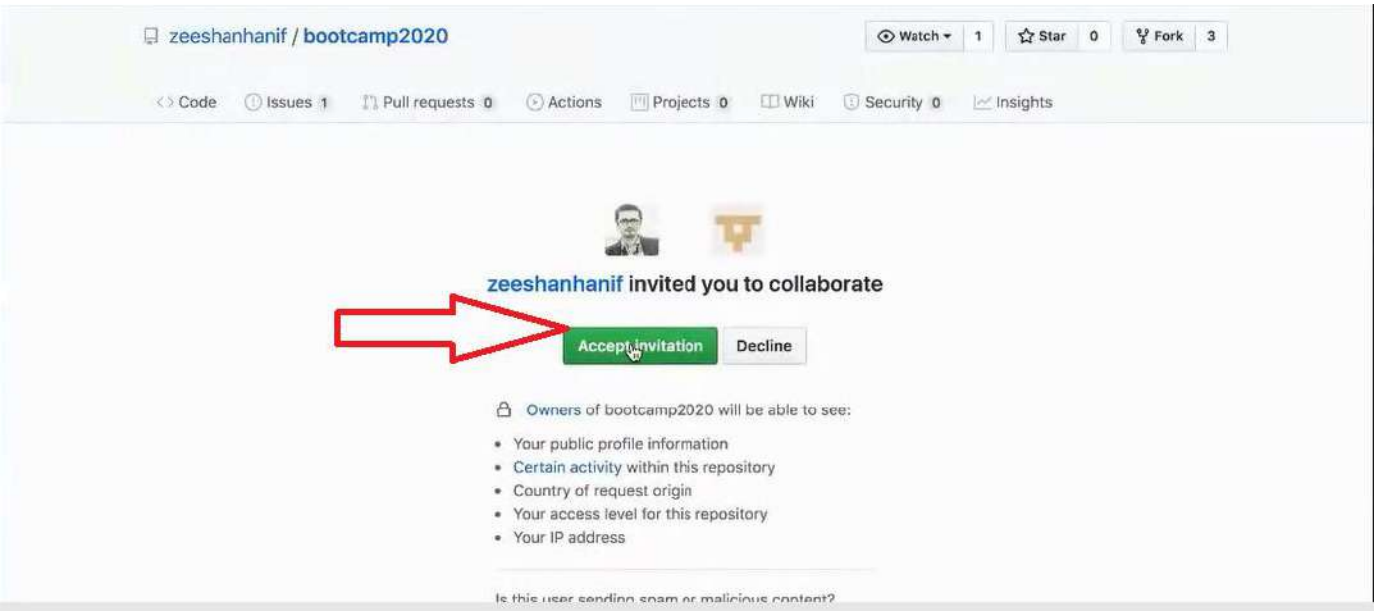


3. How To Allow Anyone To Change Code On GitHub?

- Go to setting
- Manage access
- Enter either username or email of the person you want to collaborate with **(i.e Adil-innovation-lab)**



After accepting the invitation, the other party (**i.e Adil**) will be able to make changes in the code of Html.

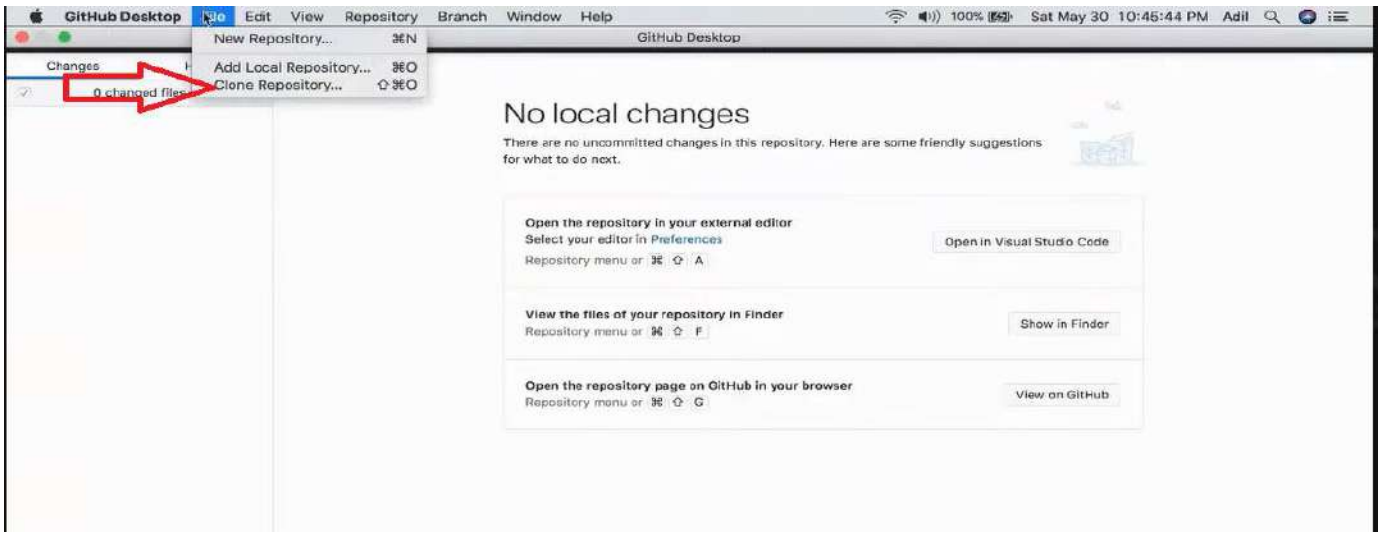


- since the initial repository is made by zeeshahanif with html code "**Helo world by Zeshan**" now another committer says Adil will change this html code to"**Helo world by Adilaltaf**"

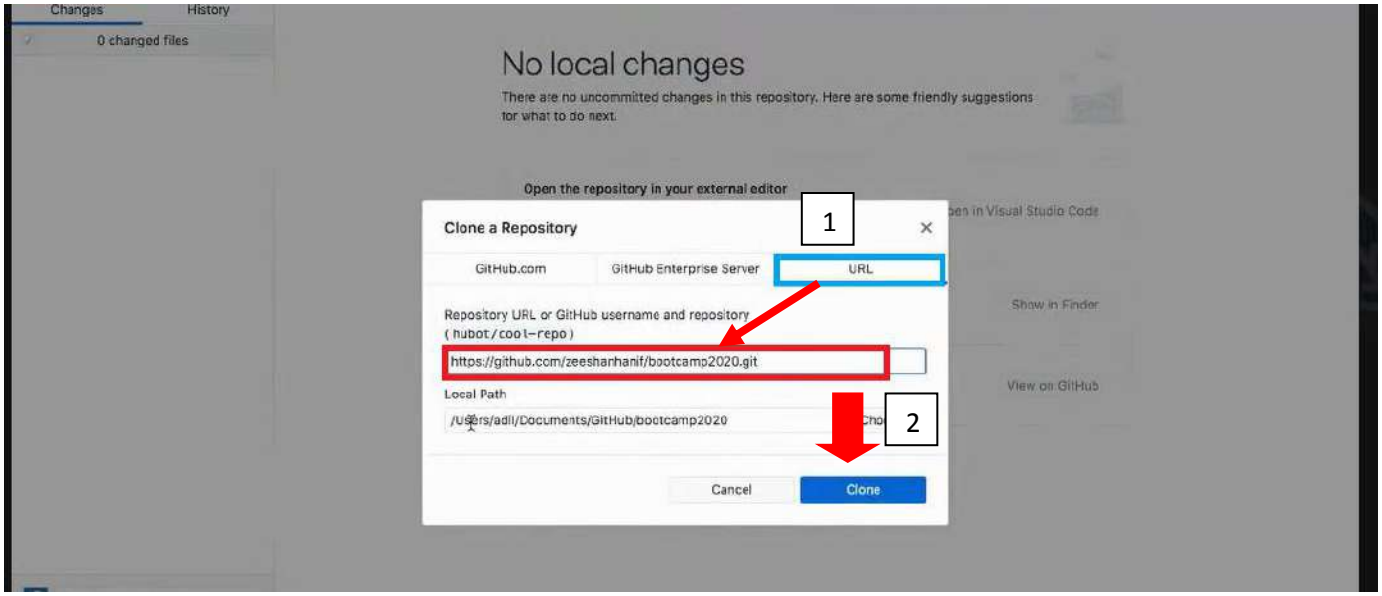
let's look how 2nd committer will change the html code

Adilaltaf will follow up on the following steps:

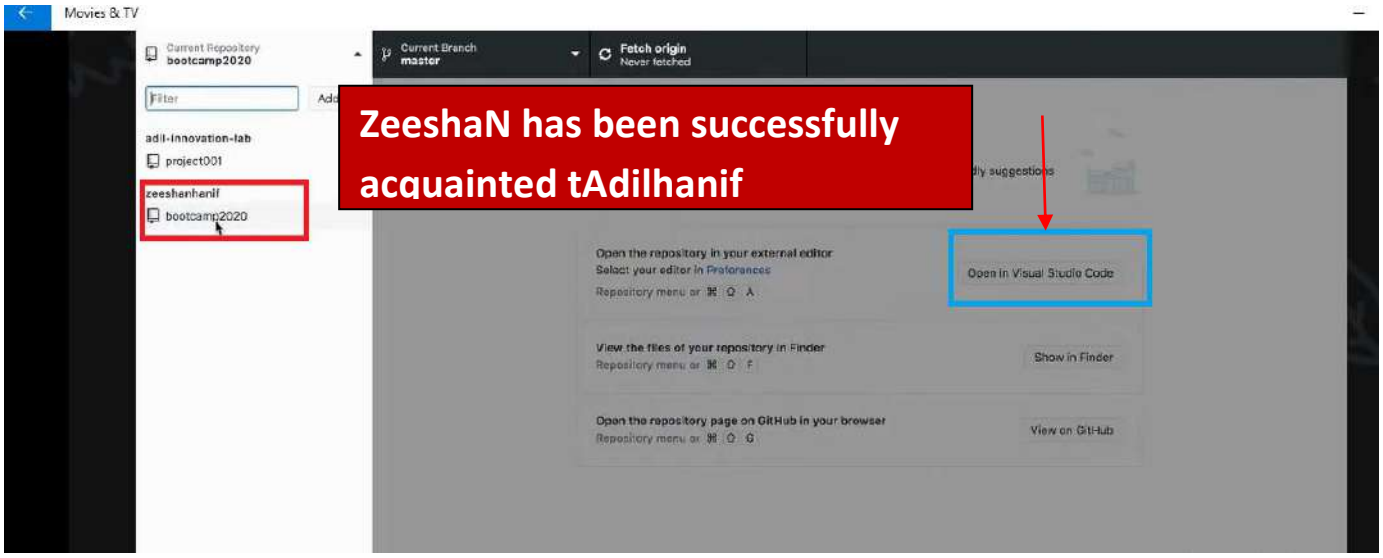
- Go to **desktop GitHub.**
- click to file at the top menu and select "**clone repository**"



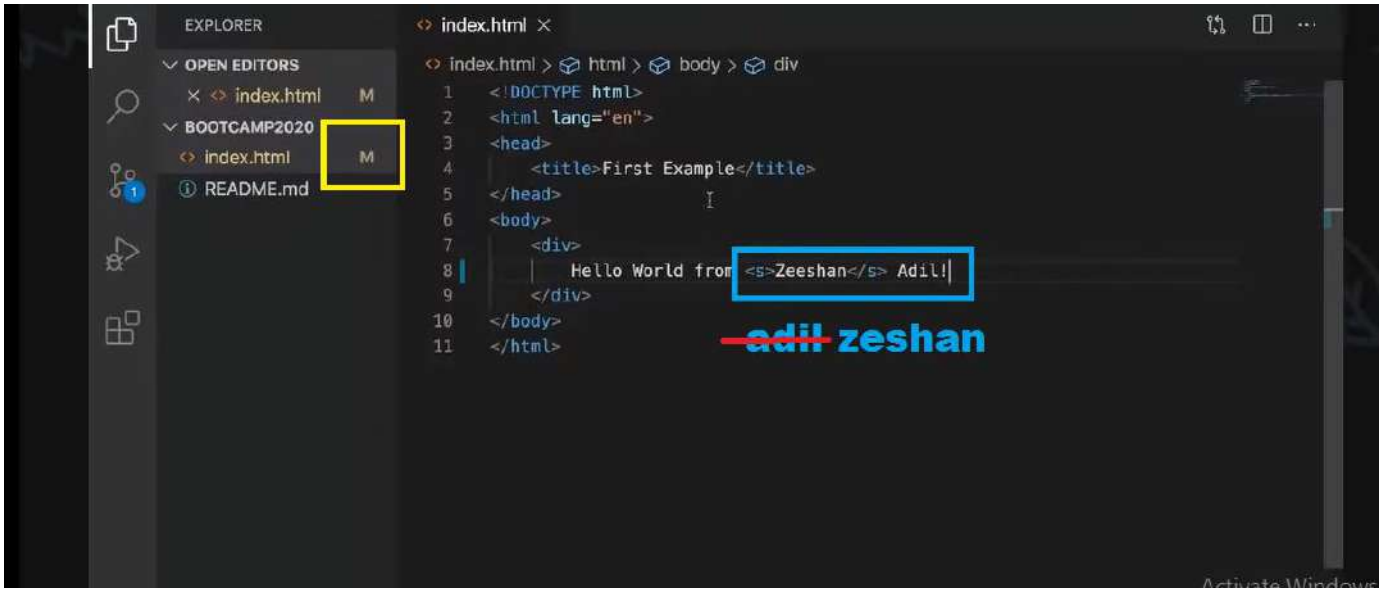
- Copy the URL from the Zeshanhanif Bootcamp repository and paste it in the **URL repository section.**
- Click to clone now.



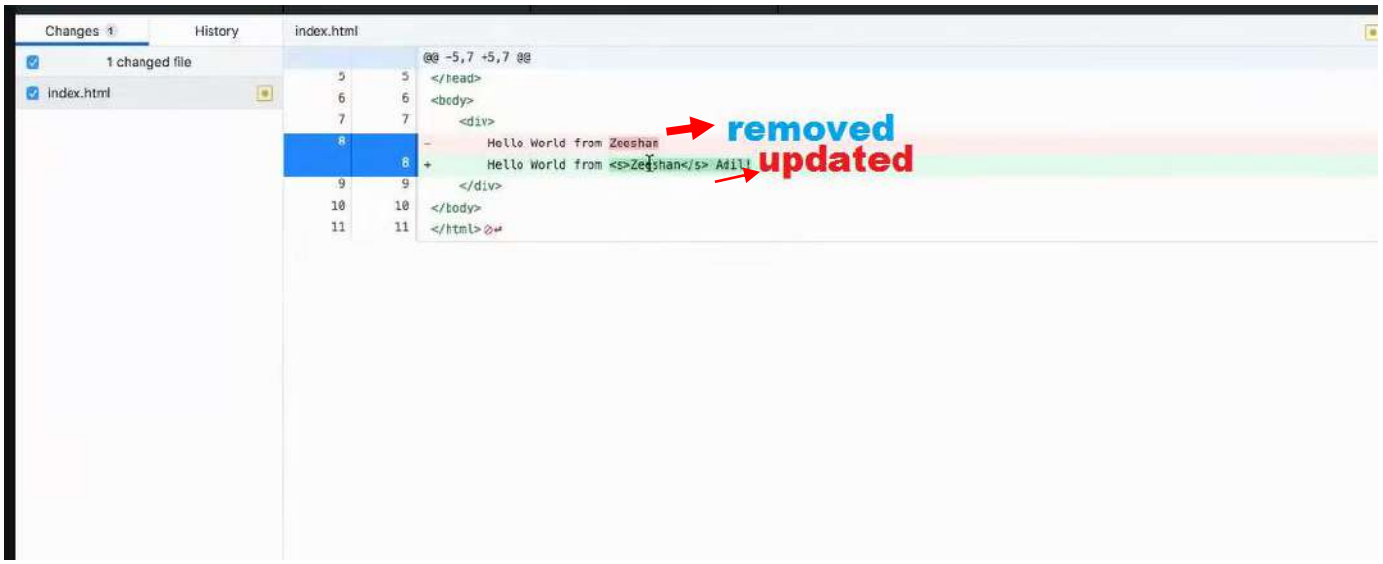
- Zeeshanhanif has been successfully acquainted with Adilaltaf
- Now click to open in visual studio code



- Go to body section of html in between and replace Adilaltaf with zeeshanhanif by using tag <s></s> as shown
- click to **M(modify) in.**

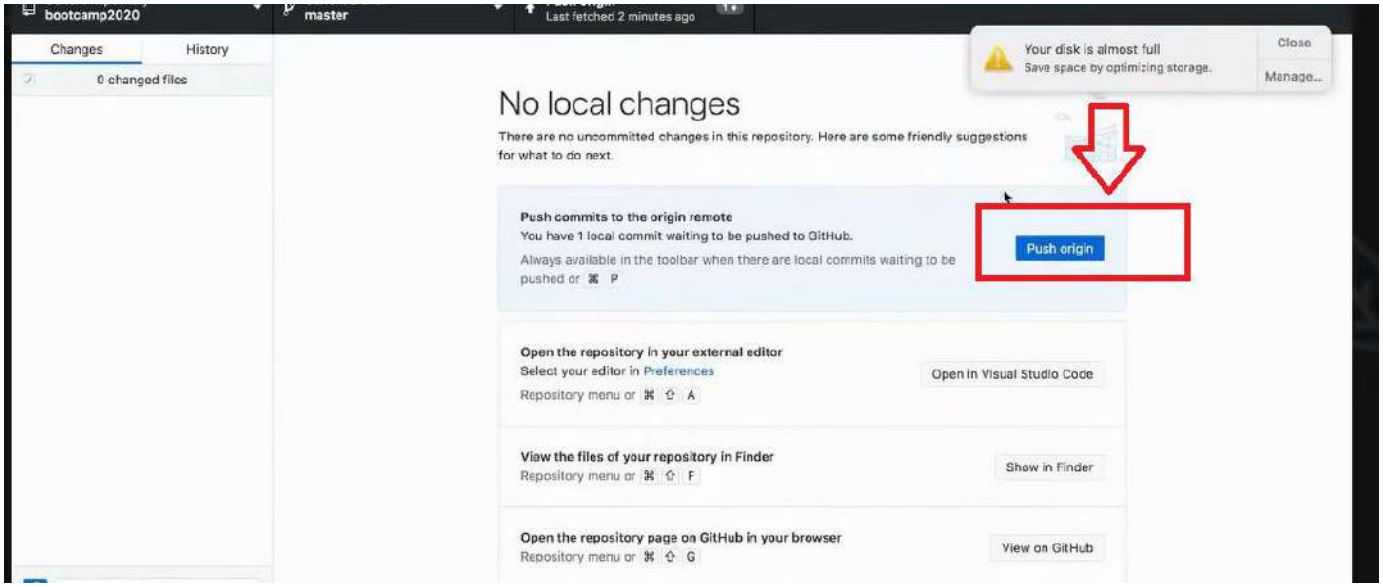


- ✓ Look into modifying **index.html**, Adilaltaf has been removed and Zeeshan has been updated



- ➢ Furthermore update index.html commit with "**no Zeshan, Its Hello world from Adil**"
- ➢ push origin and reload repository page, open up "**updated index.html**"
- ➢ Html code has been successfully updated in the repository section.

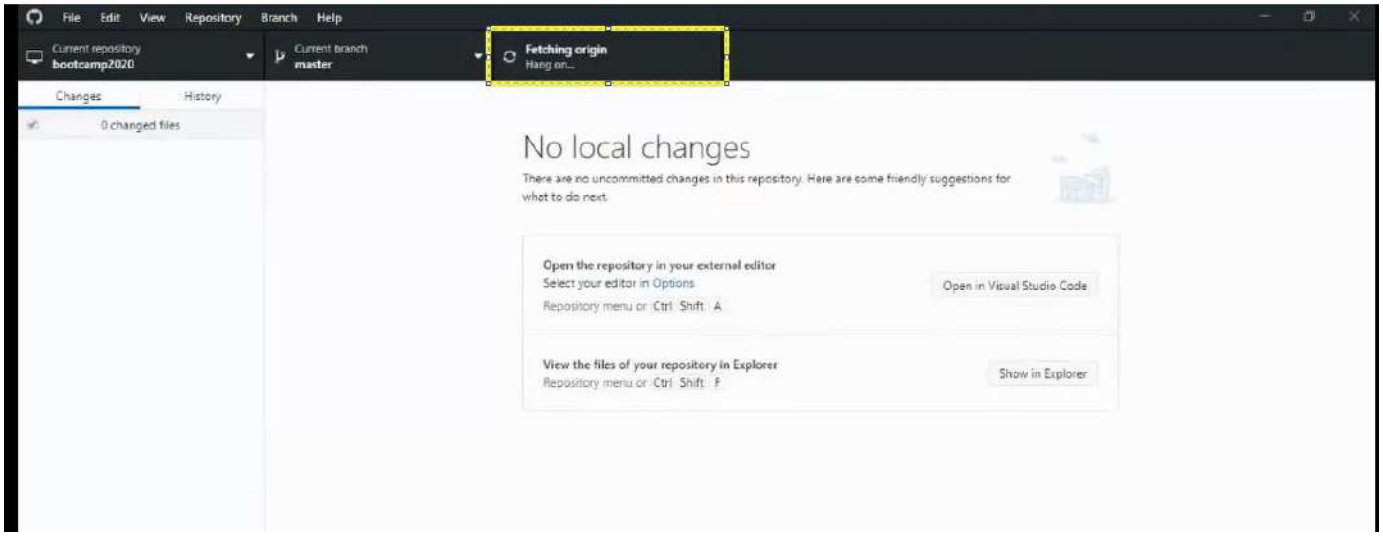
Hello World from Zeeshan
Hello World from <s>Zeeshan</s> Adil!
updating index.html commit
Updated Index.html
No Zeeshan, it's Hello World from Adil

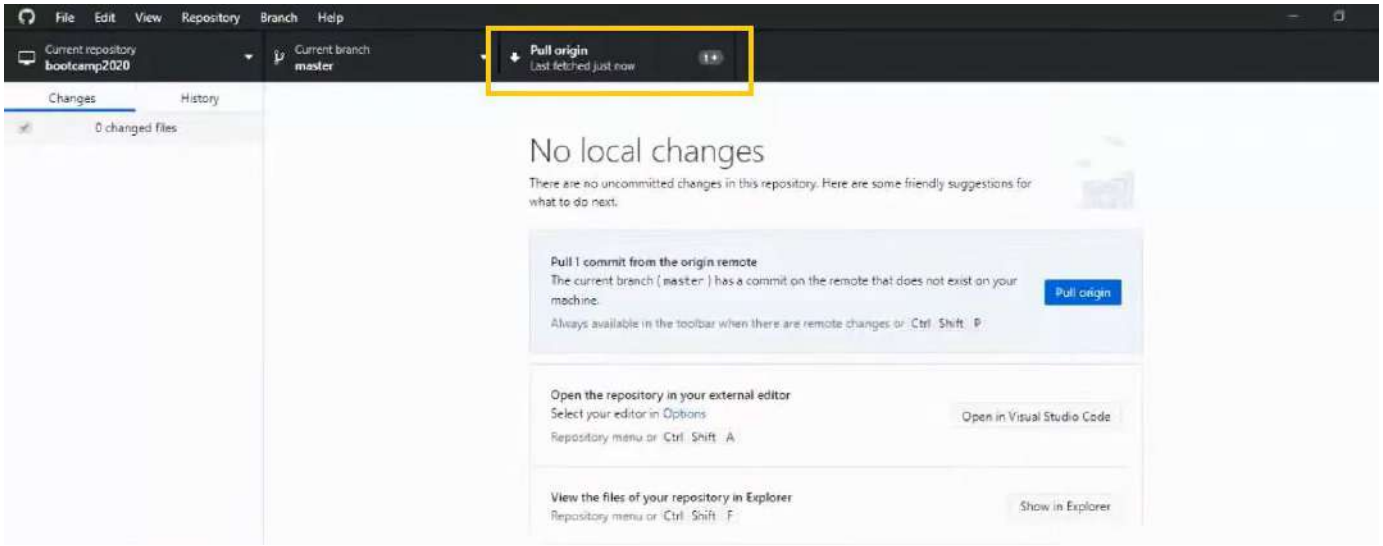

bootcamp2020
master
Last fetched 2 minutes ago
Changes
History
0 changed files
Your disk is almost full
Save space by optimizing storage.
Close
Manage...
No local changes
There are no uncommitted changes in this repository. Here are some friendly suggestions for what to do next.
Push commits to the origin remote
You have 1 local commit waiting to be pushed to GitHub.
Always available in the toolbar when there are local commits waiting to be pushed or ⌘ P
Push origin
Open the repository in your external editor
Select your editor in Preferences
Repository menu or ⌘ ⇧ A
Open in Visual Studio Code
View the files of your repository in Finder
Repository menu or ⌘ ⇧ F
Show in Finder
Open the repository page on GitHub in your browser
Repository menu or ⌘ ⇧ G
View on GitHub

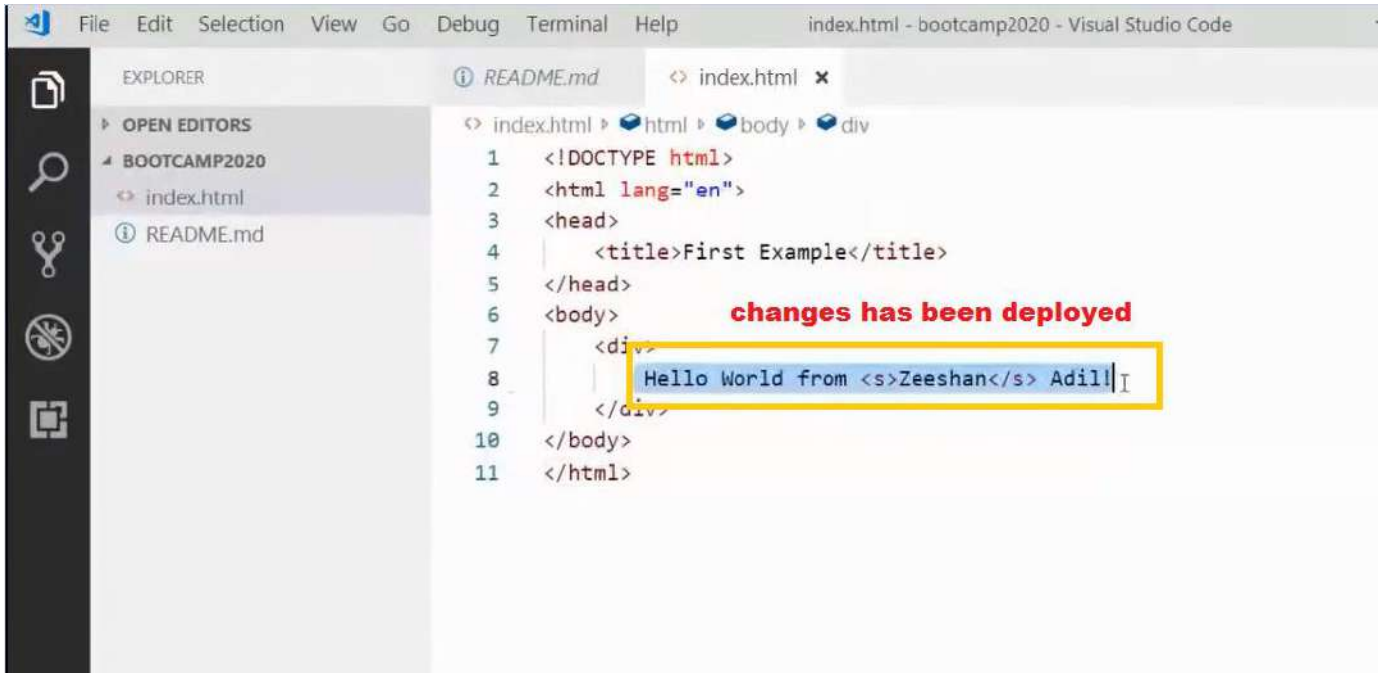
Note:

To keep change on the repository 1st committer needs to

- Go to his GitHub desktop.
- Click to fetch origin.
- Click to pull origin.
- View it in the Bootcamp repository; it will deploy changes that are made by 2nd committer.



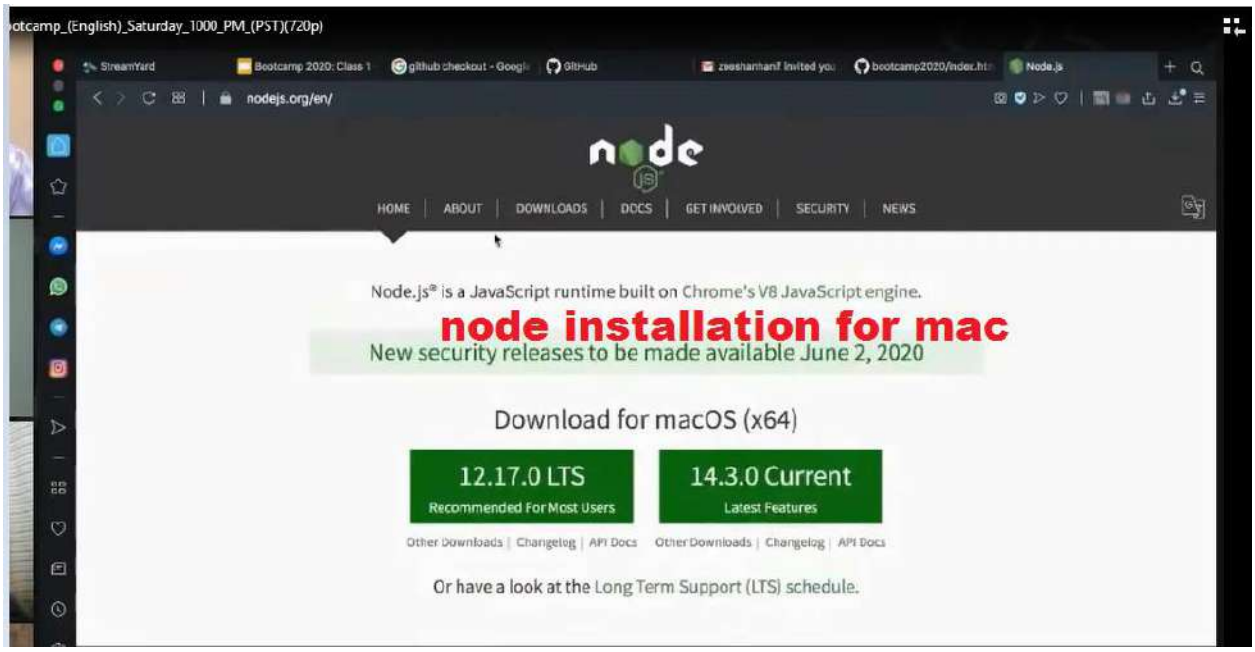




❖ **To publish this deployed change use surge. Let us go into deep of this surge.**

3.1. Use Of Surge With Github Repository

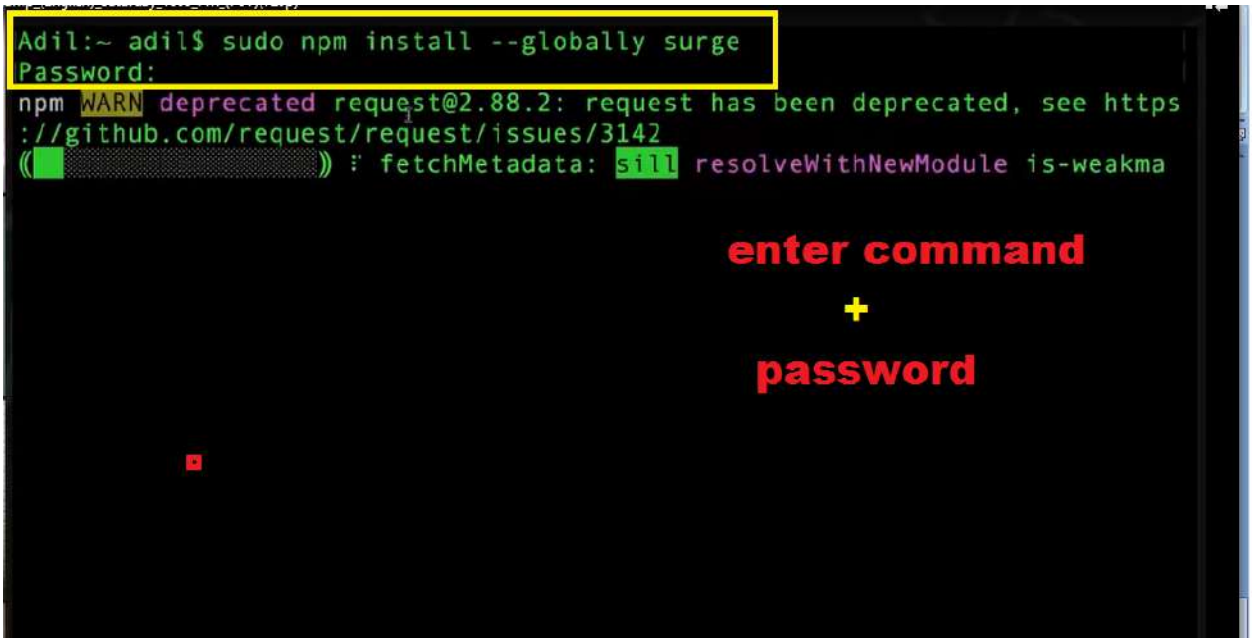
- ✓ Surge Allow users to access static front and application .surge is a hosting platform, that publishes our websites free of cost, where users can access our content, by integrating surge with GitHub action and workflows we can configure GitHub repository to automatically deploy our website for free surge hosting, whenever there new push code to GitHub repository
- ✓ Install node for using the surge in GitHub repository either for MAC or window
- ✓ There is the installation of the node on MAC



- ➢ After installing “node” from “nodejs.org”
- ➢ Go to the command prompt
- ➢ open new window with profile<click to Homebrew

- ❖ **verify the following command and exit command.**

- **Next step is to install surge**
- Use command-line installation using the node package manager
- Use **sudo** command for admin access < **npm(node package manager)< --- globally surge** **(for public access)**
- **Enter the password and install surge online**

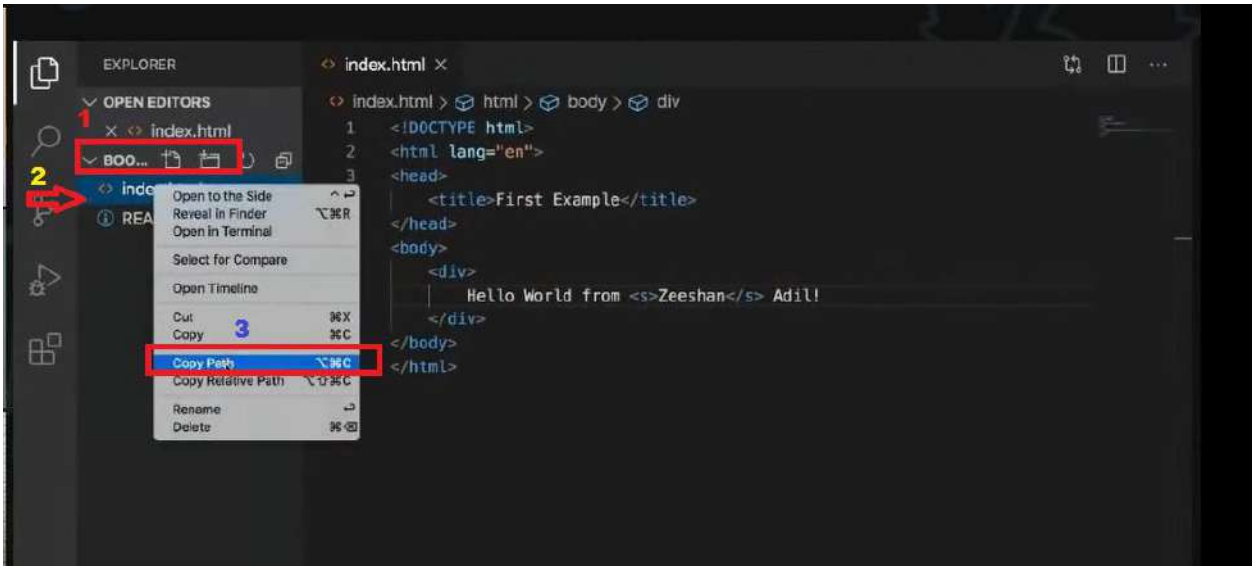


```
Password:
npm WARN deprecated request@2.88.2: request has been deprecated, see https
://github.com/request/request/issues/3142
npm WARN saveError ENOENT: no such file or directory, open '/Users/adil/pa
ckage.json'
npm notice created a lockfile as package-lock.json. You should commit this
 file.
npm WARN enoent ENOENT: no such file or directory, open '/Users/adil/packa
ge.json'
npm WARN adil No description
npm WARN adil No repository field.
npm WARN adil No README data
npm WARN adil No license field.
+ surge@0.21.3
added 164 packages from 115 contributors and audited 164 packages in 22.61
9s

30 packages are looking for funding
  run `npm fund` for details
```

surge installation

3.2. Creation Of Account On Surge

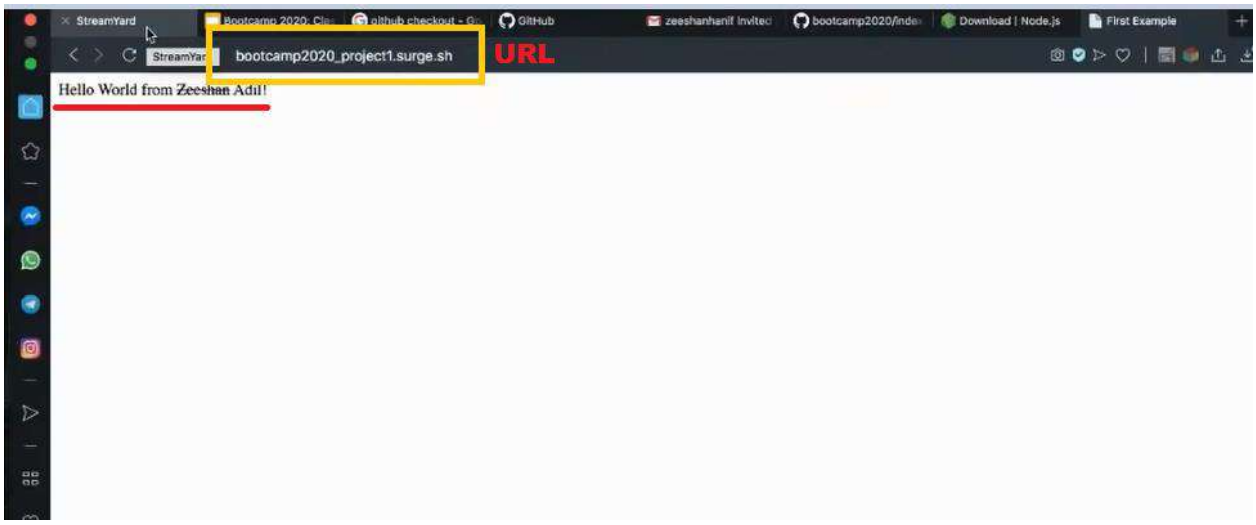
- Create an account on the surge and generate a security token to show up locally
- To Deploy website write Surge command<location of Bootcamp to get the path "copy path"<enter URL**(bootcamp2020-project1.surge.sh)**
- Press enter



- **NOW website has been published successfully on the provided URL**

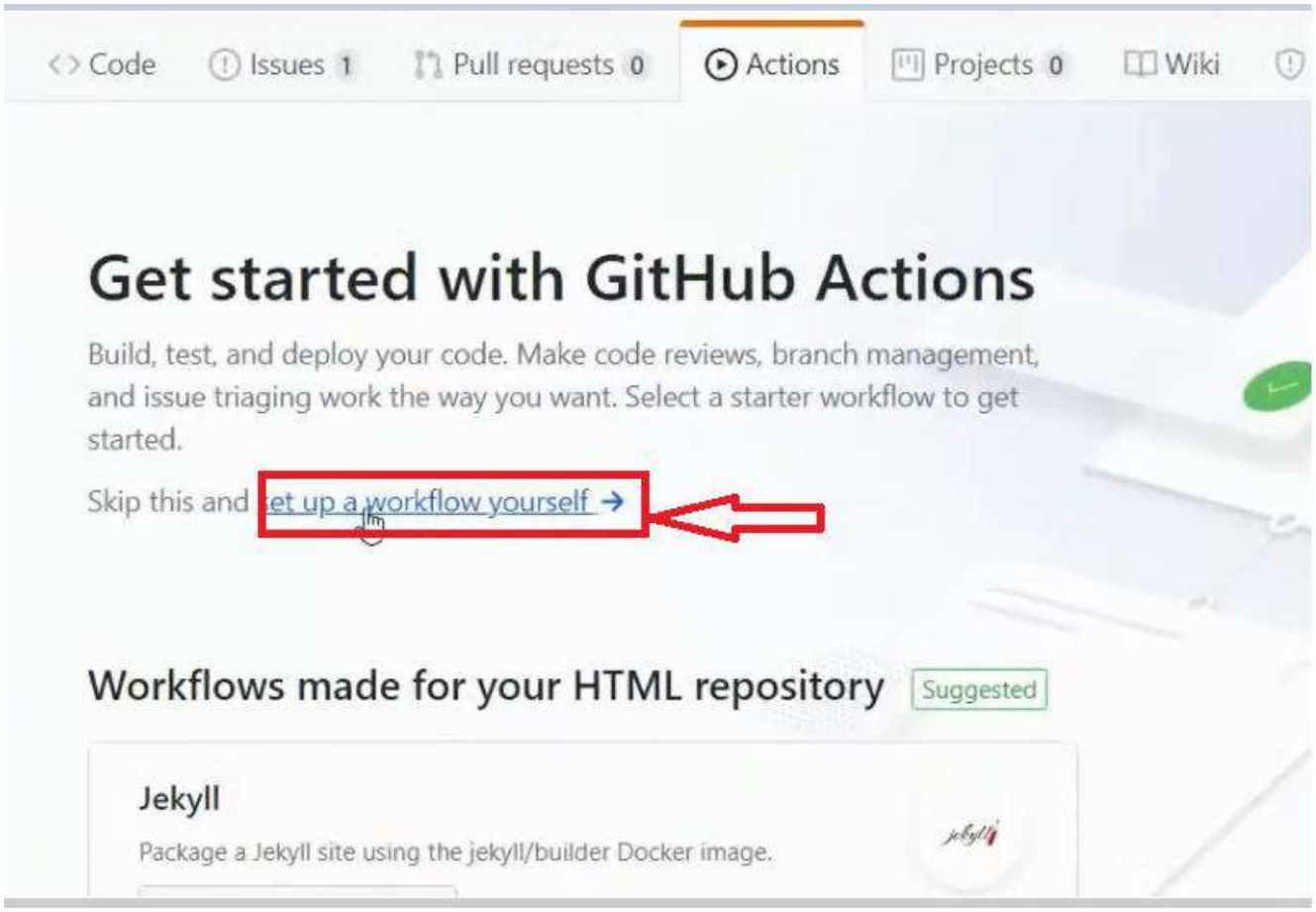
```
domain: bootcamp2020.surge.sh

  Aborted - you do not have permission to publish to bootcamp2020.surge.s
h
                    account creation

Adil:~ adil$ surge /Users/adil/Documents/GitHub/bootcamp2020/ bootcamp2020
_project1.surge.sh

  Running as khanadilaltaf@gmail.com (Student)

      project: /Users/adil/Documents/GitHub/bootcamp2020/
       domain: bootcamp2020_project1.surge.sh
       upload: [] 100% eta: 0.0s (2 files, 221 bytes)
          CDN: [====================] 100%
           IP: 45.55.110.124
```

- Copy this URL from any browser open it. it will show “hello world ~~from~~ Zeeshan Adil”
- NOW anyone can copy this URL in the browser and can view it publically.



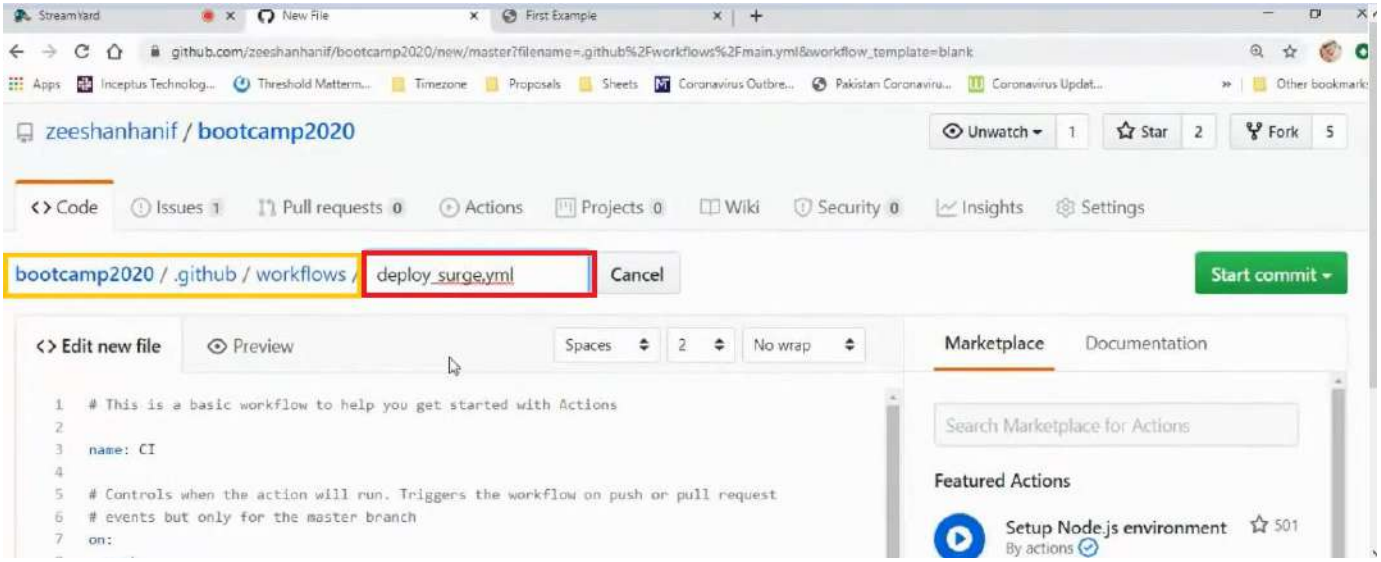
- Go to GitHub.
- There are multiple options, click to “Action”.
- Click to “set up a workflow for yourself”.
- It will create **“bootcamp2020/github/yml**” to configure deploy changes.
- Name file i.e to **deploy surge.yml.**

adil-innovation-lab has been added as a collaborator on the repository.
zeeshanhanif / bootcamp2020
Unwatch 1
Star 0
Fork 2
Code
Issues 1
Pull requests 0
Actions
Projects 0
Wiki
Security 0
Insights
Settings
Options
Manage access
Branches
Webhooks
Notifications
Who has access
PUBLIC REPOSITORY
This repository is public and visible to anyone.
Manage
DIRECT ACCESS
1 has access to this repository. 1 invitation.

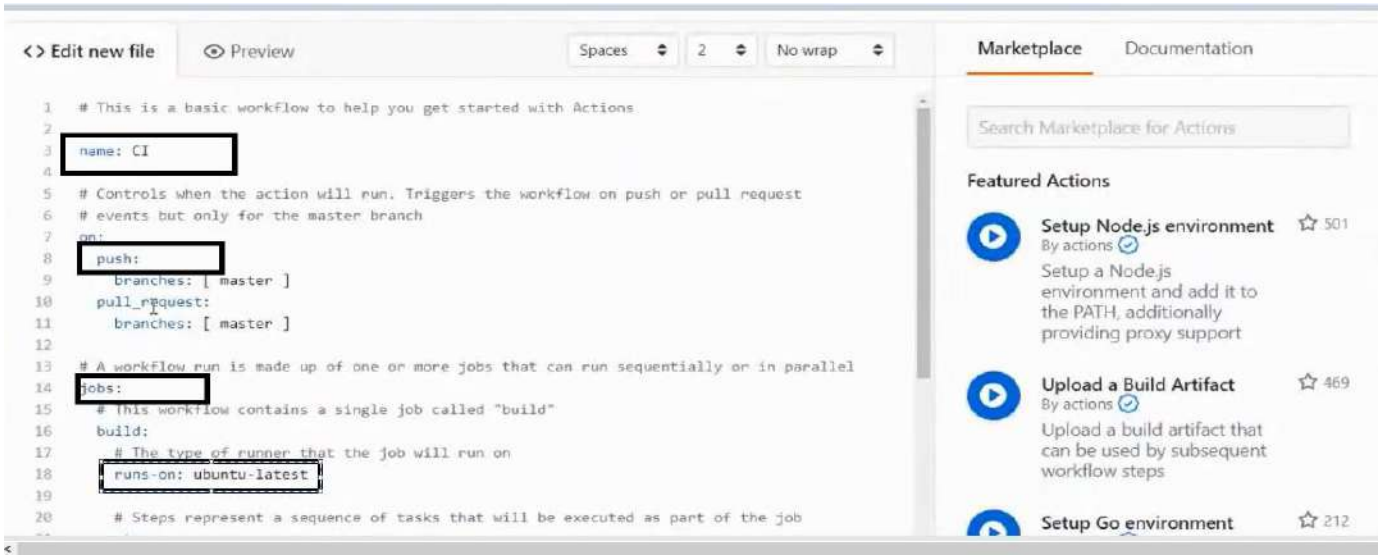

Code
Issues 1
Pull requests 0
Actions
Projects 0
Wiki
Get started with GitHub Actions
Build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way you want. Select a starter workflow to get started.
Skip this and set up a workflow yourself →
Workflows made for your HTML repository
Suggested
Jekyll
Package a Jekyll site using the jekyll/builder Docker image.

4. Automatic Deployment Of Code on GitHub Repository

Let us deploy automatic change from "Hello world zeeshanhanif" to "Hello world from Hira khan"



- To deploy changes this file will show the different declarative option
  - Name(enter name you want)
  - Push(push the action you want to run i.e master and when you want to run, it will trigger that action on push)
  - Jobs (it results in the execution of push action i.e execution of code)
    - ✓ Build (build code of action)
    - ✓ Runs –ubuntu(create VMWare virtual machine ubuntu to run GitHub server on this virtual machine according to manual changes that had been made to your Bootcamp file)



- **So all changes or code will first execute in that virtual machine and then it will be deployed on Github server**

(install nodeJs and surge as given)

- **Take the following steps to perform your job**

Steps:

#checks out your repository under $GITHUB-WORKSTATION, so your job can access it

Use the following declaration ;

Uses:action/checksout@v2

#runs a single command using run shell

name: install NodeJs

uses:action/setup-node@v2-beta

with :

node version:12

name ;install surge

run: npm install –g surge

name :run surge

run:surge ./ bootcampewapp.surge.st – token ${{secrets.SURGE_TOKEN}}

Note "$ {{ }}." will help the GitHub server to take the token from the deployer when he uses this surge operation for changes

Bootcampwapp.surge.sh is used for the authentication of the user in virtual machine ubuntu.

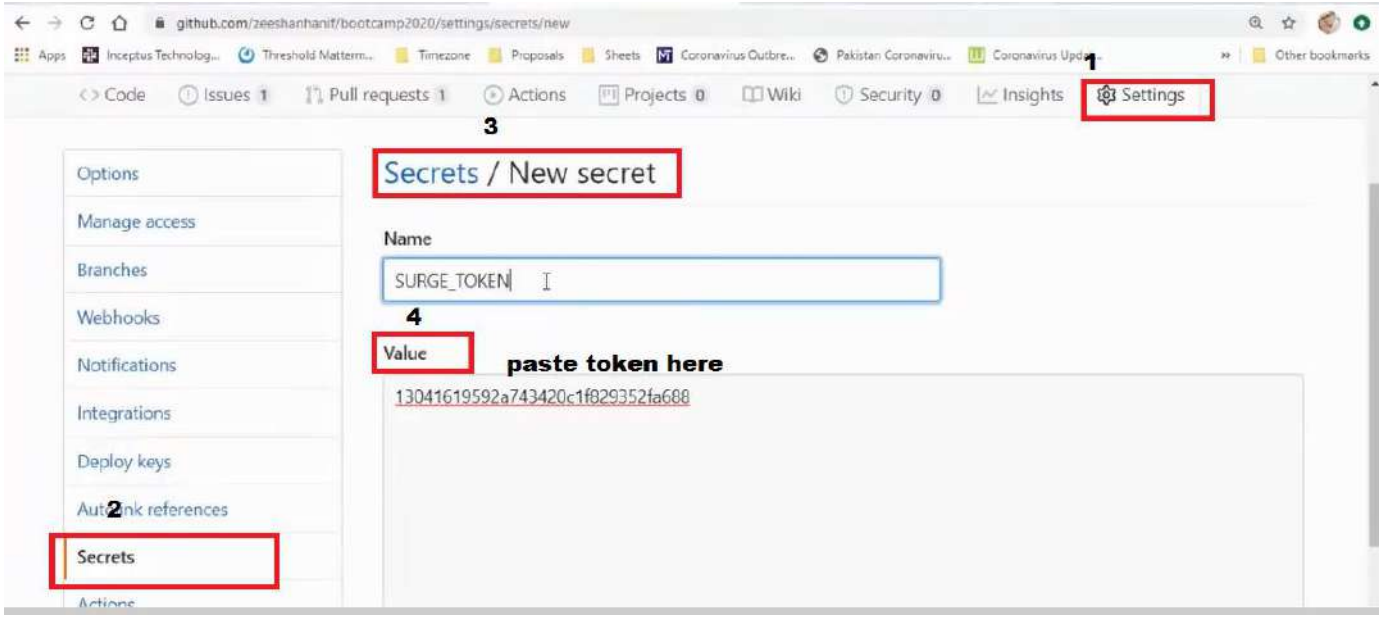
Edit new file | Preview | Spaces 2 No wrap

```
16      runs-on: ubuntu-latest
17
18      # Steps represent a sequence of tasks that will be executed as part of the job
19      steps:
20      # Checks-out your repository under $GITHUB_WORKSPACE, so your job can access it
21      - uses: actions/checkout@v2
22
23      # Runs a single command using the runners shell
24      - name: Install NodeJS
25        uses: actions/setup-node@v2-beta
26        with:
27          node-version: 12
28      - name: Install Surge
29        run: npm install -g surge
30      - name: Run Surge
31        run: surge ./ bootcampnewapp.surge.sh --token ${{secret.SURGE_TOKEN}}
32
33
```

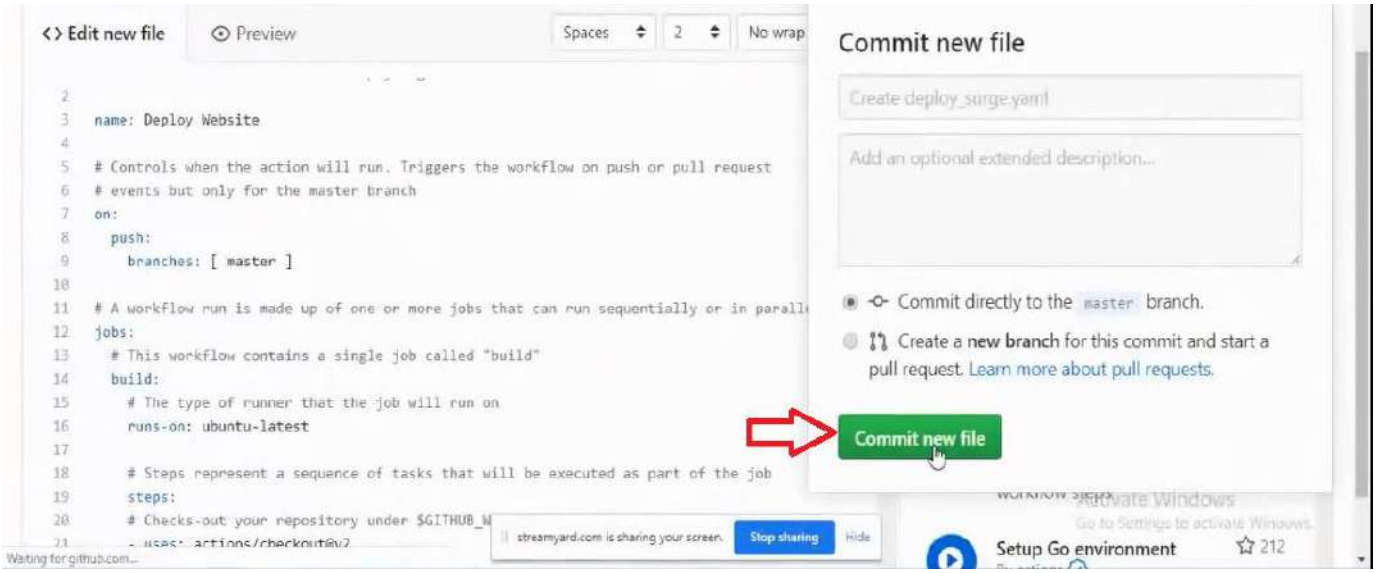
token

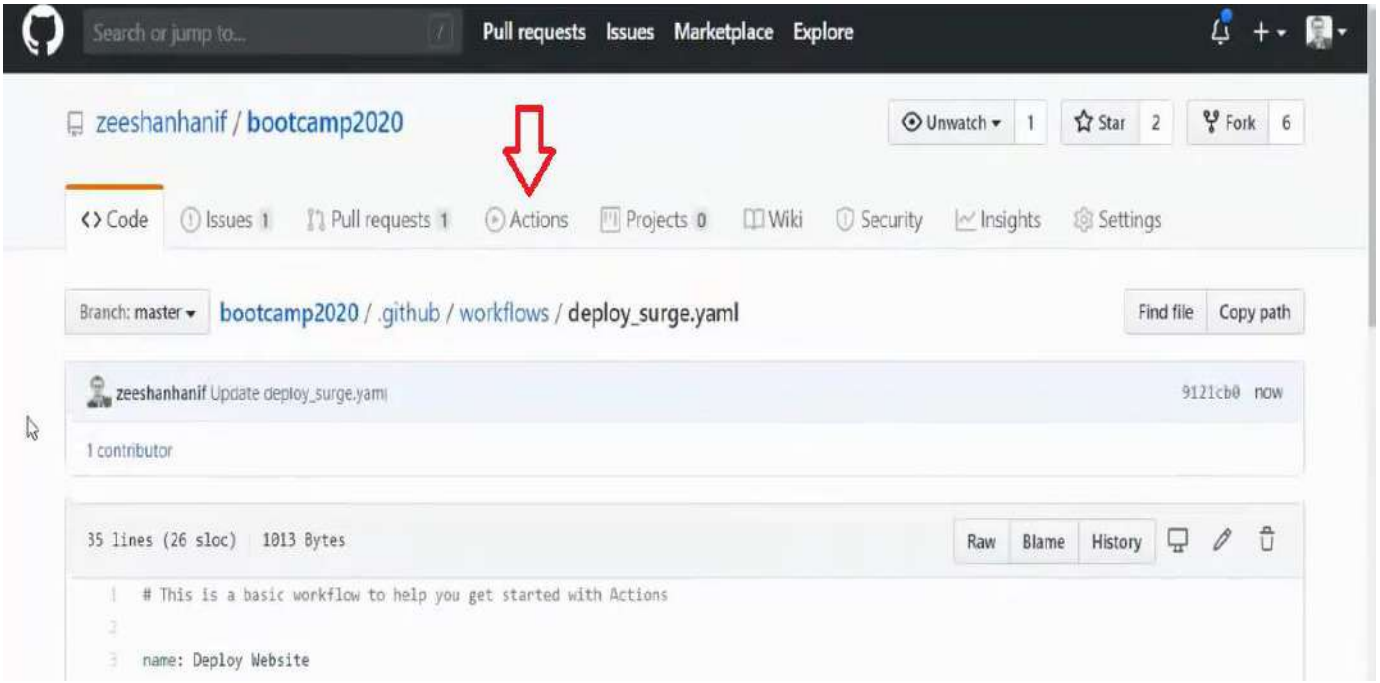
NOTE

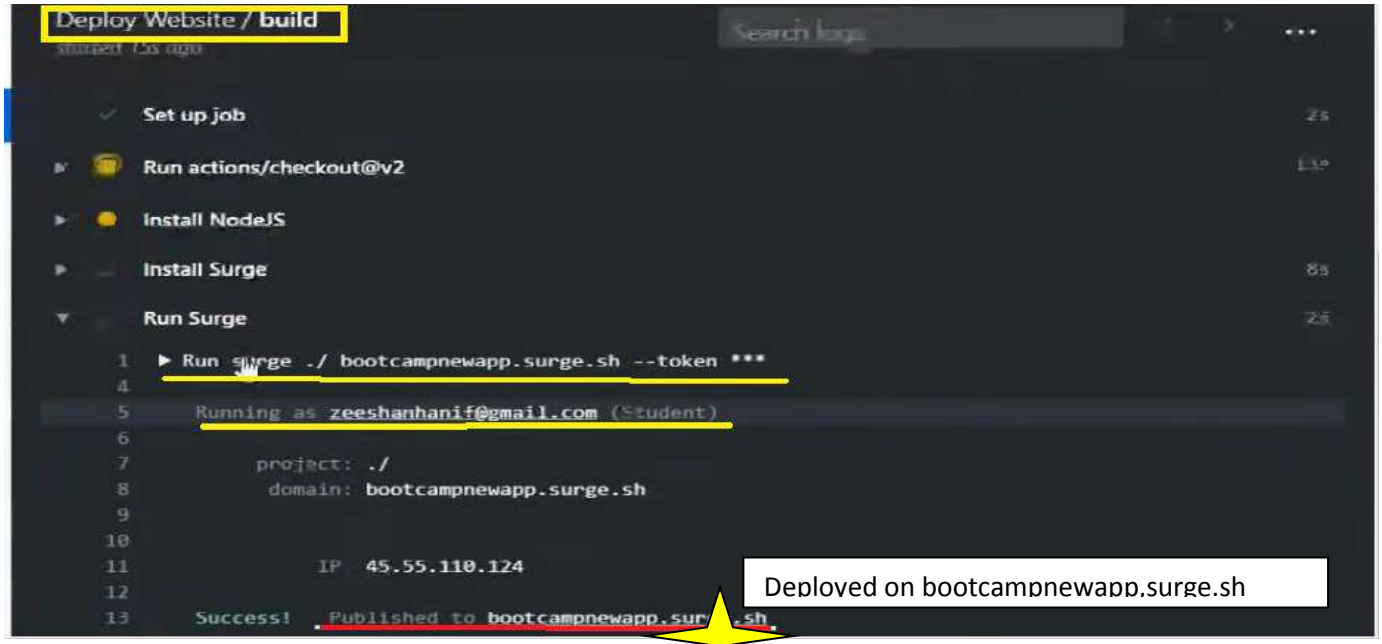
- For the purpose to run surge you have to take the token from the GitHub terminal
  - GO to GitHub desktop.
  - Copy token.
  - Go to the GitHub repository.
  - Click on the setting.
  - Go to secrets.
  - Name secrets name and value.



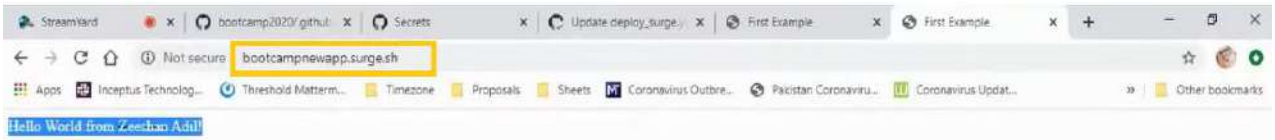
- Next click to commit the new file
- Choose action
- Deployment of the website is running as shown with the user-id



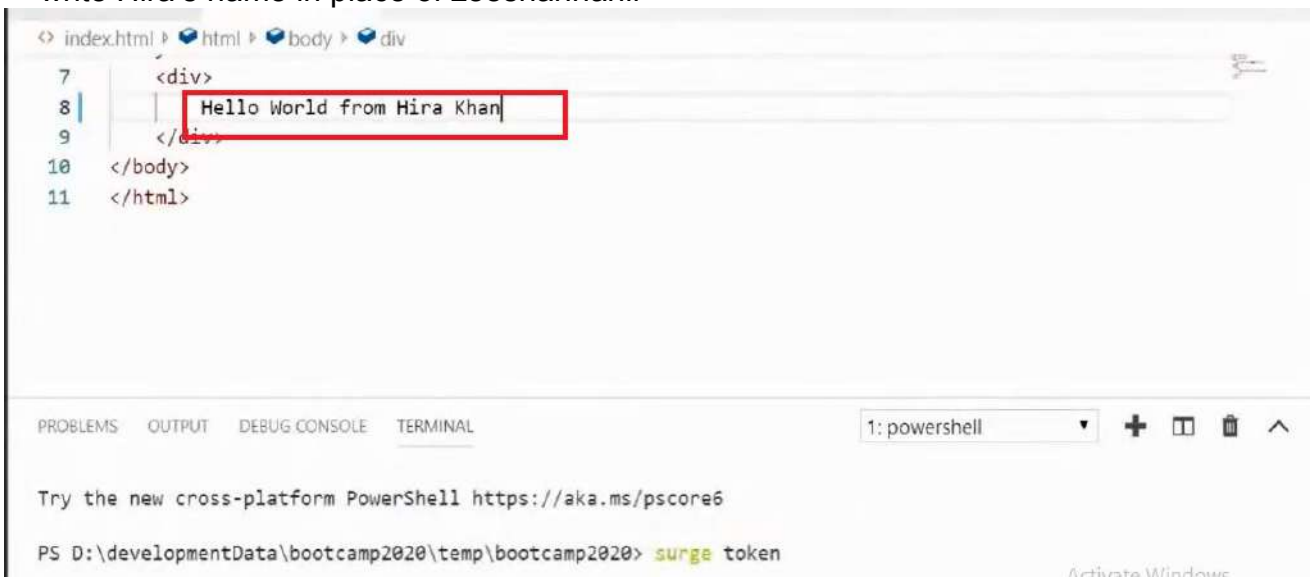




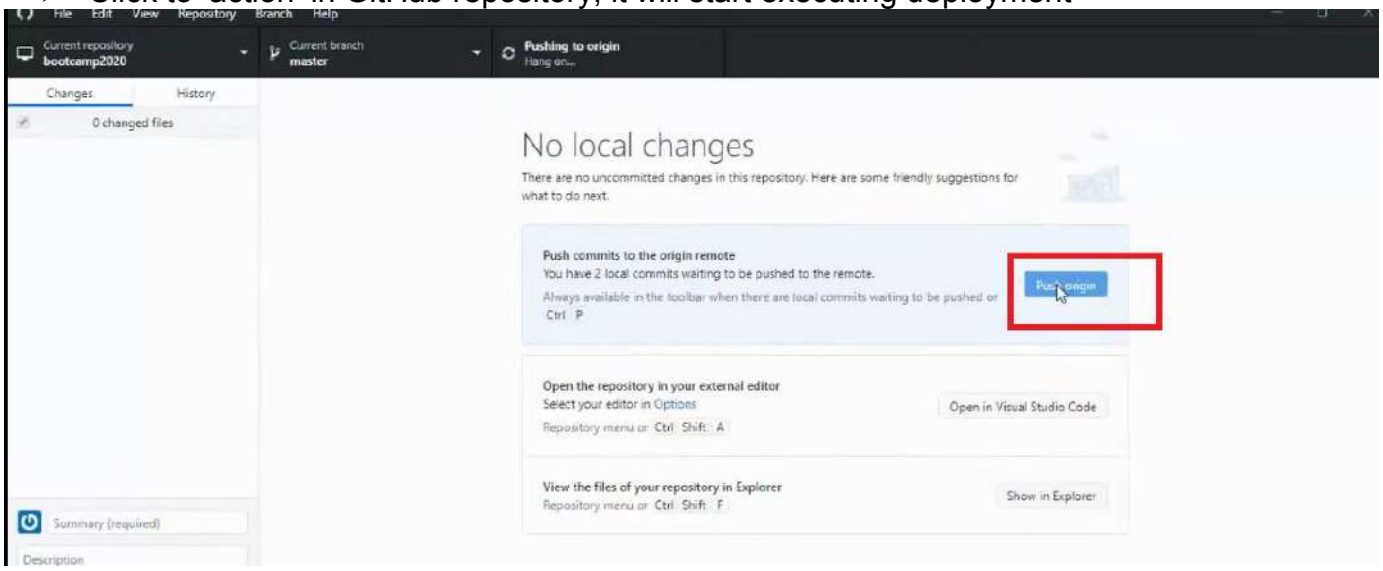
- Open up in browser “bootcampnewapp,surge.sh “
- You will see it as



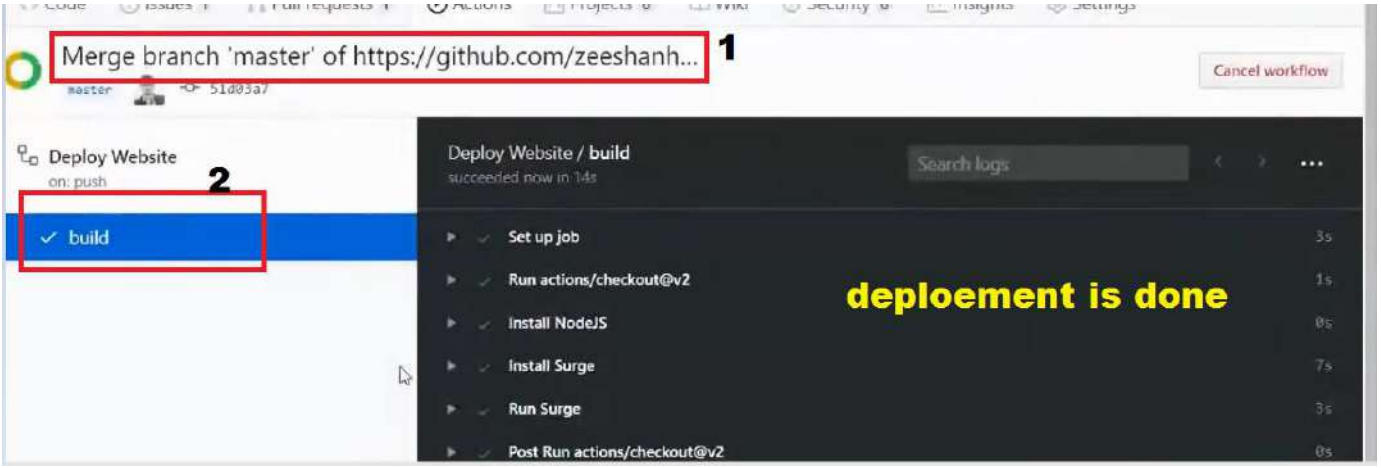
- Now to replace Hira Khan name in place of Zeeshanhanif you go to GitHub desktop and write Hira’s name in place of zeeshanhanif



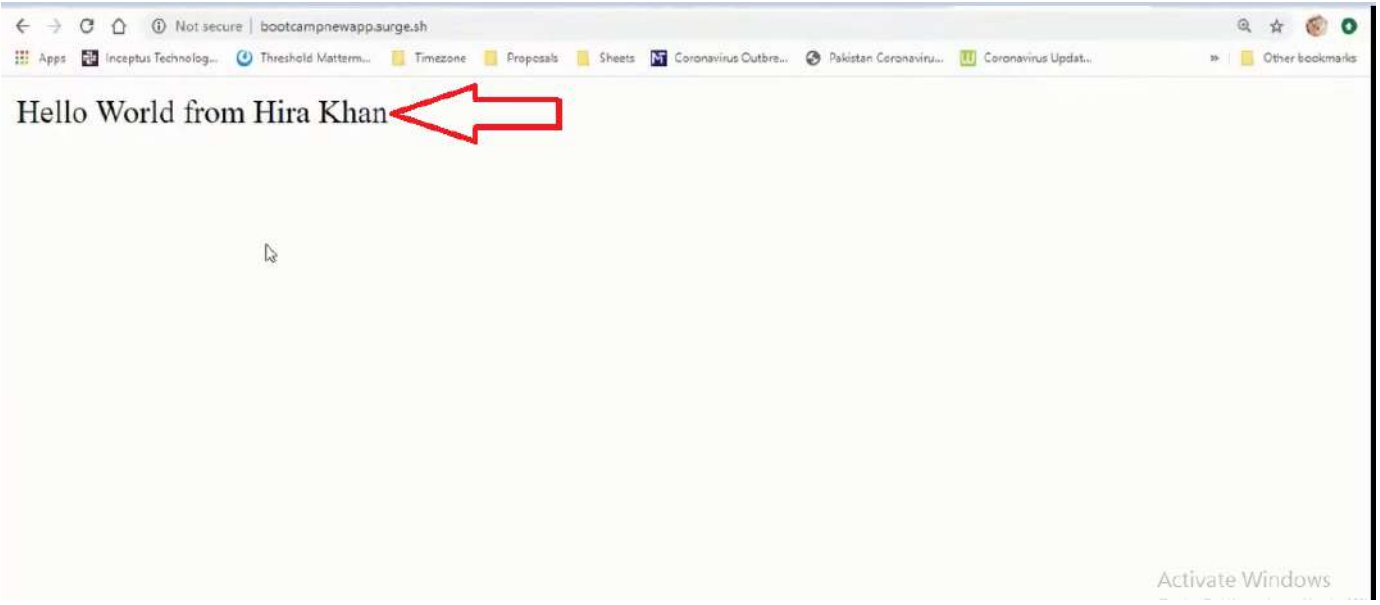
- Push origin to deploy changes
- Click to ‘action’ in GitHub repository, it will start executing deployment



- Click to merge branch master
- Click to build
- Deployment is done



- **Refresh the browser and see changes has been made by GitHub action**



Result

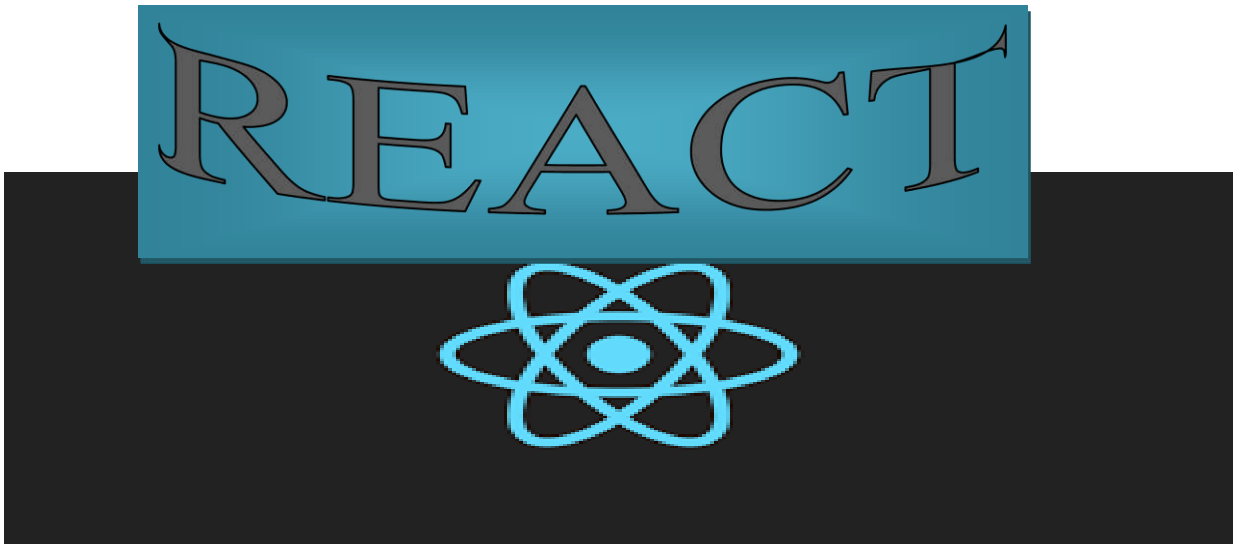
Github repository in Bootcamp plays an important rule

- It provides free hosting services for any website and show it publically
- It helps in tracking of any changes made by team members
- It keeps all existing records
- Instead of writing codes again and again making changes, githup repository allows you to make automated changes from GitHub action services
- Hence it is cheaper and time-saving

Session 2: React Base App

REACT BASE APLLICATION

Development of React Base Application using react



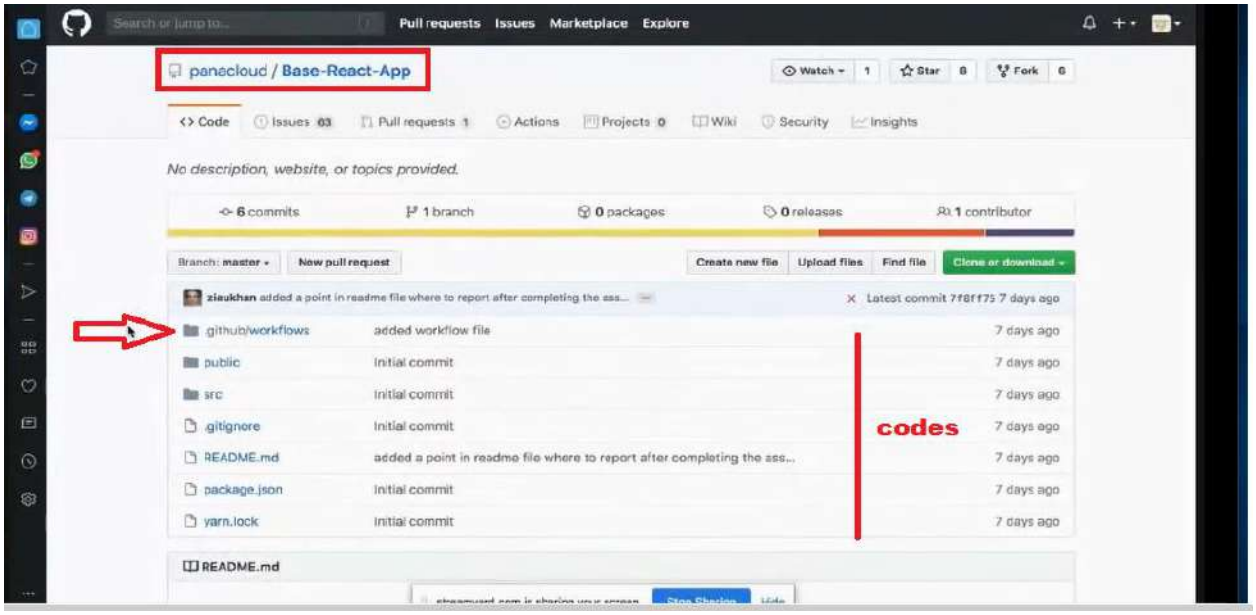
1. **<u>What Is React?</u>**

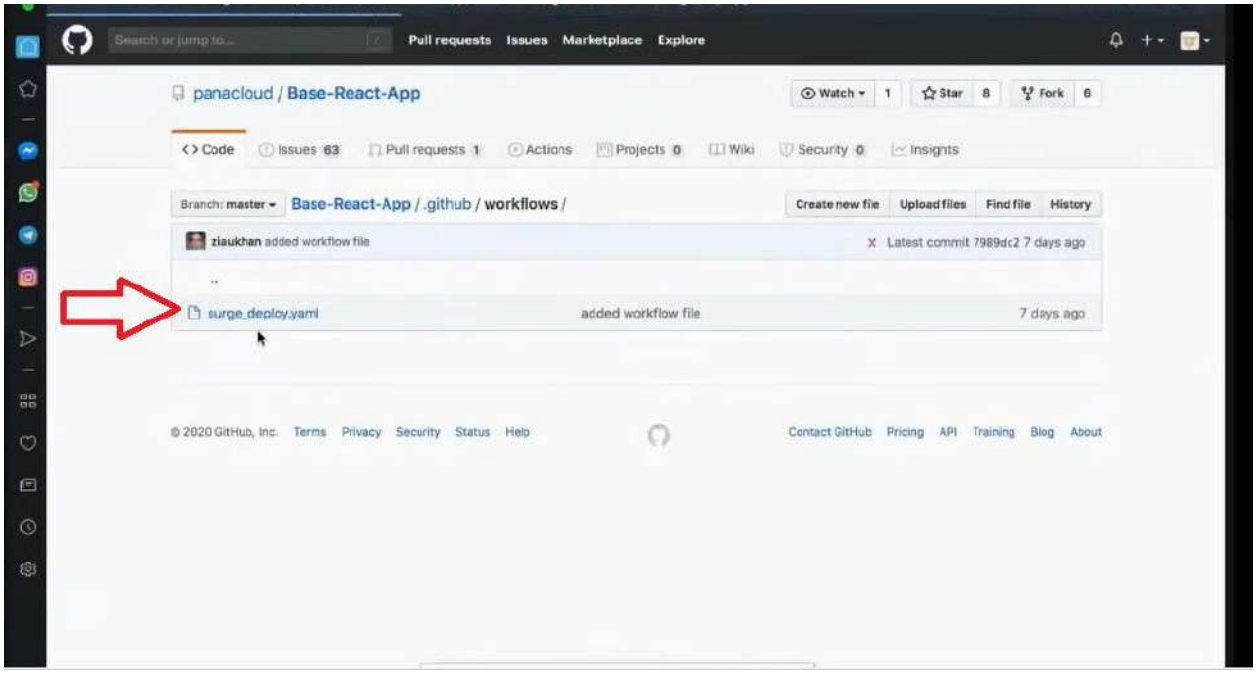
React is a declarative, efficient, and flexible JavaScript library for building user interfaces. It lets you compose complex UIs from small and isolated pieces of code called “components”.

It is maintained by Facebook and a community of individual developers and companies. React can be used as a base in the development of single-page or mobile applications. However, React is only concerned with rendering data to the DOM and so creating React applications usually requires the use of additional libraries for state management and routing. Redux and React Router

1.1. Development Of React Base Application

- Open the react base app GitHub repository, it contains different codes for the deployment of react base application.
- Now click to the workflow.
- Open surge-deploy.yaml file.





- Configure a workflow in surge-deploy.yaml file to deploy an application that is running on react
- workflow needed to configure with basic react application is:
  For that purpose set up the machine (it should contain basic yaml file for workflows)

This includes three major components

1. **Name of workflow**

```
name: deploy website
```

2. **Event** that causes the workflow to execute

```
On:[push]
```

3. **Jobs**: that call ubuntu-latest to run that is deploying on surge;

```
Jobs:
    build:
     runs-on:ubntu latest
    name:deploying to
```

Steps To Deploy The Application To Surge

- Check out the code on GitHub repository using an action/GitHub checkout action

```
-uses:action/checkout@v2
```

- Installing node using the setup node action; install node-version:12

```
-name:install Node.js
-uses:actions/setups-
with:
        node version:12
```

- Install Yarn to build the react application, to build the dependencies, to get the application ready for the run time environment.

```
-name:prepare for yarn installation
```

- This is the command to the workflow for installing yarn, this is curl statement
- Event for yarn installation is; Run:sudo apt update $$ sudo apt install yarn

```
run:curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg|sudo apt-key add- echo"debt
https://dl.yarnpkg.com/debian/stable main"  |sudo tee/etc/apt/source.list.d/yarn

    -name :install yarn

     run:sudo apt && sudo apt install yarn

    -name: build react app
```

- Install surge on your system
- Run deployment on the surge using the same token as we have done before in 1st tutorial.

```
-name: Run surge

      run:surge ./build bootcamp-react-ziakhan.surge.sh --token${{ secrets. SURGE-TOKEN}}
```

- Create a folder "project003 by using the "npx" syntax create-react-app in the main GitHub folder.

Use this Command `npx create_react-app project003`

```
Adil:GitHub adil$ npx create-react-app project003
npx: installed 98 in 12.84s
```

- Enter ctrl, react- app is started creating

```
Adil:.Trash adil$ cd /Users/adil/Documents/GitHub
Adil:GitHub adil$ clear

Adil:GitHub adil$ npx create-react-app project003
npx: installed 98 in 12.84s
```

creating react app

```
Creating a new React app in /Users/adil/Documents/GitHub/project003.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...

yarn add v1.19.1
[1/4] 🔍  Resolving packages...
[2/4] 🚚  Fetching packages...
[3/4] 🔗  Linking dependencies...
warning "react-scripts > @typescript-eslint/eslint-plugin > tsutils@3.17.1" has
 unmet peer dependency "typescript@>=2.8.0 || >= 3.2.0-dev || >= 3.3.0-dev || >
= 3.4.0-dev || >= 3.5.0-dev || >= 3.6.0-dev || >= 3.6.0-beta || >= 3.7.0-dev ||
 >= 3.7.0-beta".
```

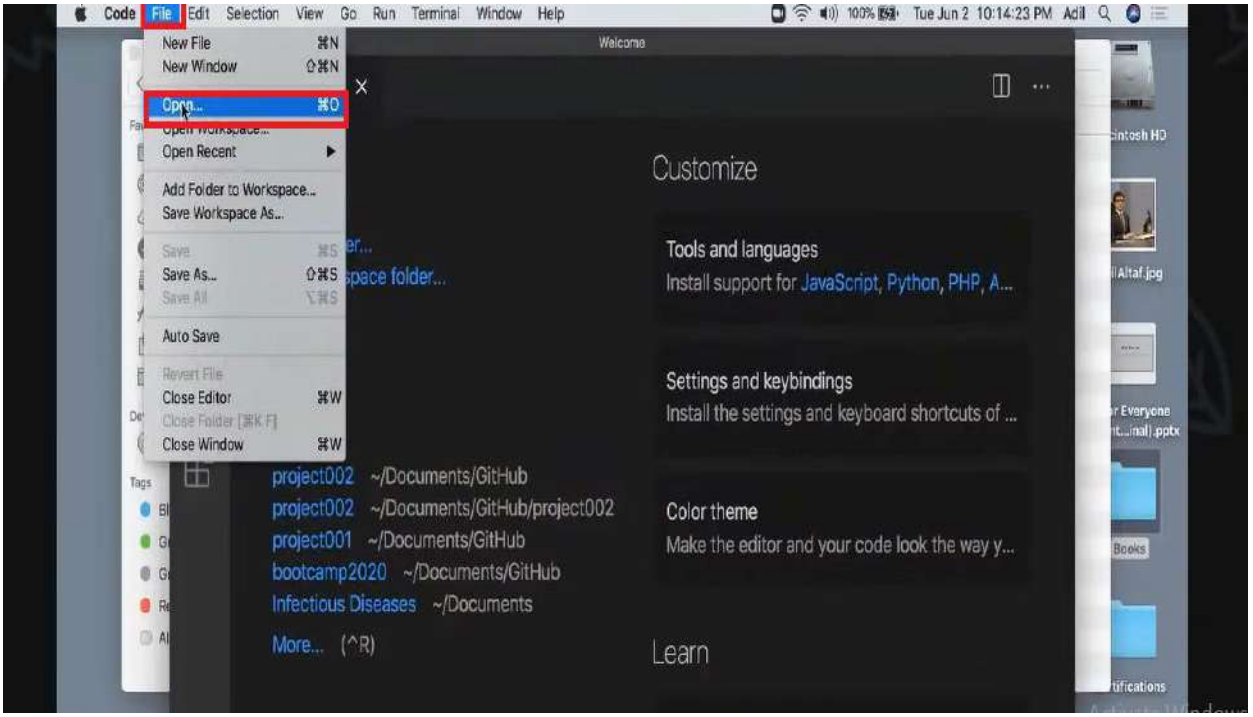
❖ **Success in creating git commit**

```
Created git commit.

Success! Created project003 at /Users/adil/Documents/GitHub/project003
Inside that directory, you can run several commands:

  yarn start
    Starts the development server.

  yarn build
    Bundles the app into static files for production.

  yarn test
    Starts the test runner.

  yarn eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd project003
```

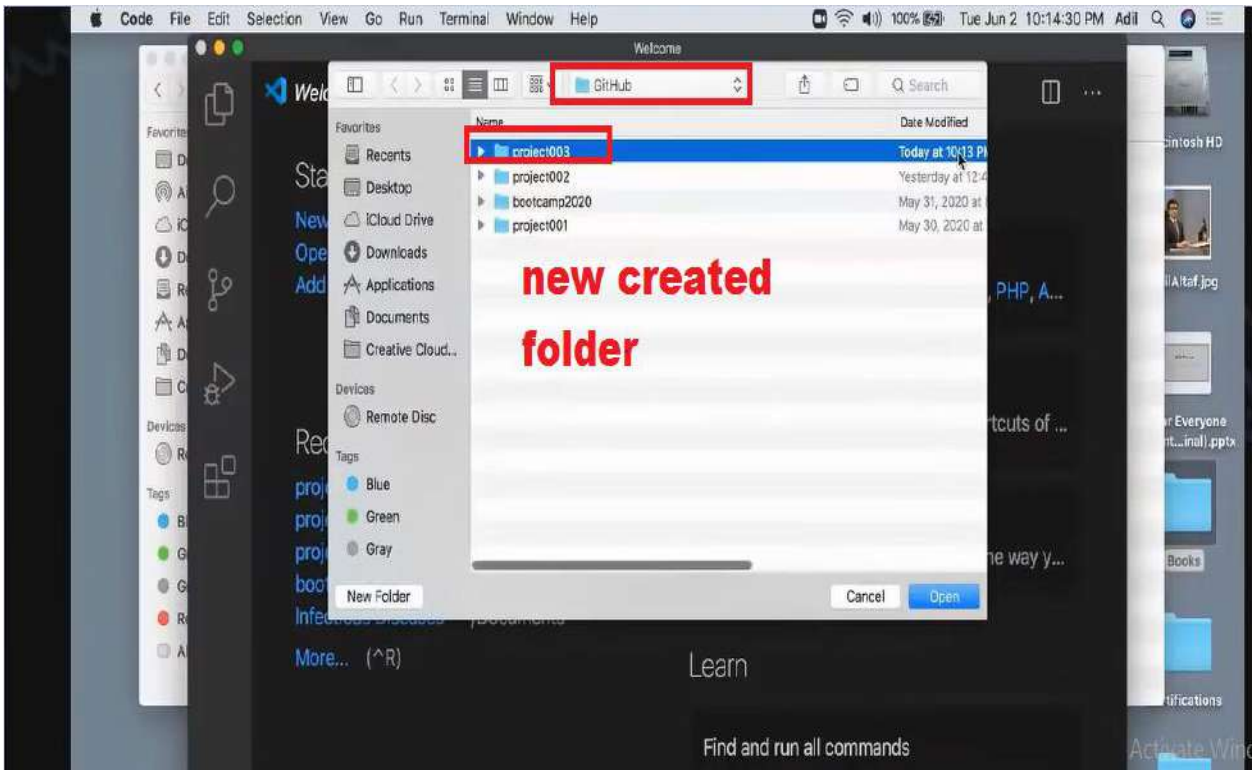
successfully created a git

Activate Windows

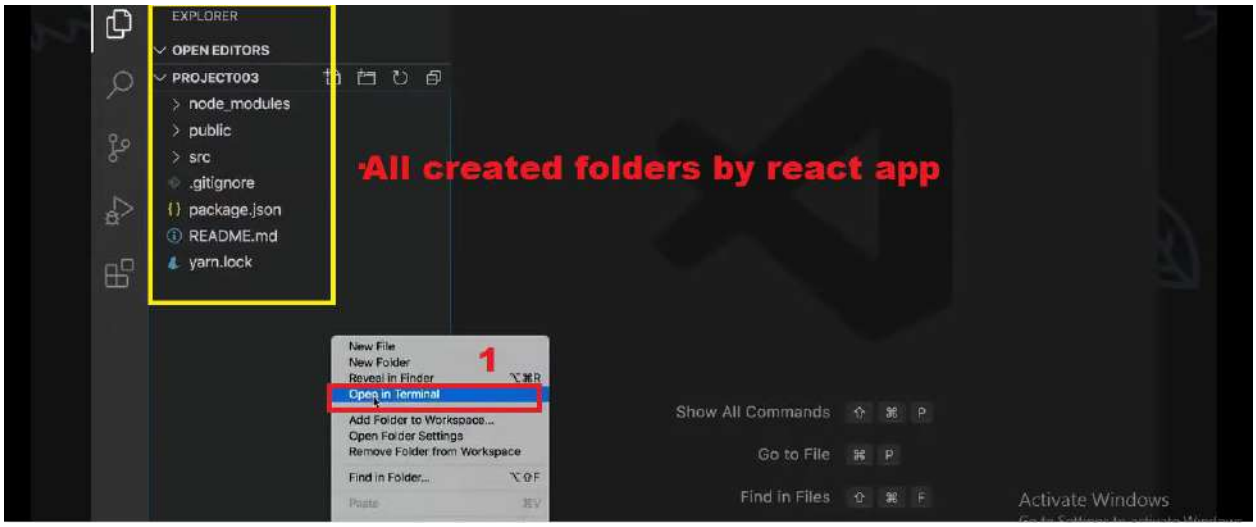
- Open in visual studio
- Click to file < Click to open the newly created folder.



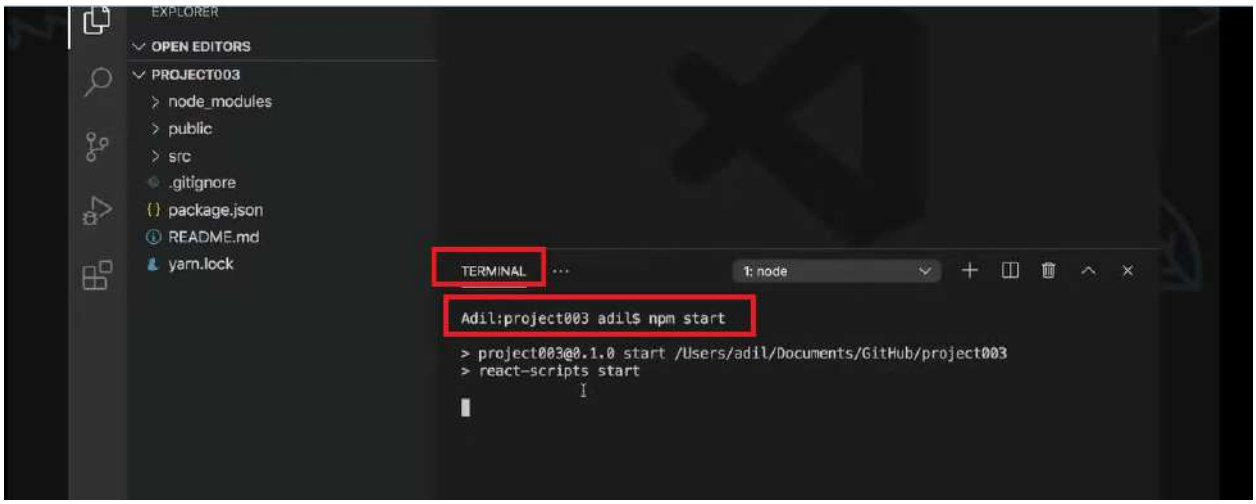
- open up project003



- All created folders are shown there, now click to right and go to “open in terminal”

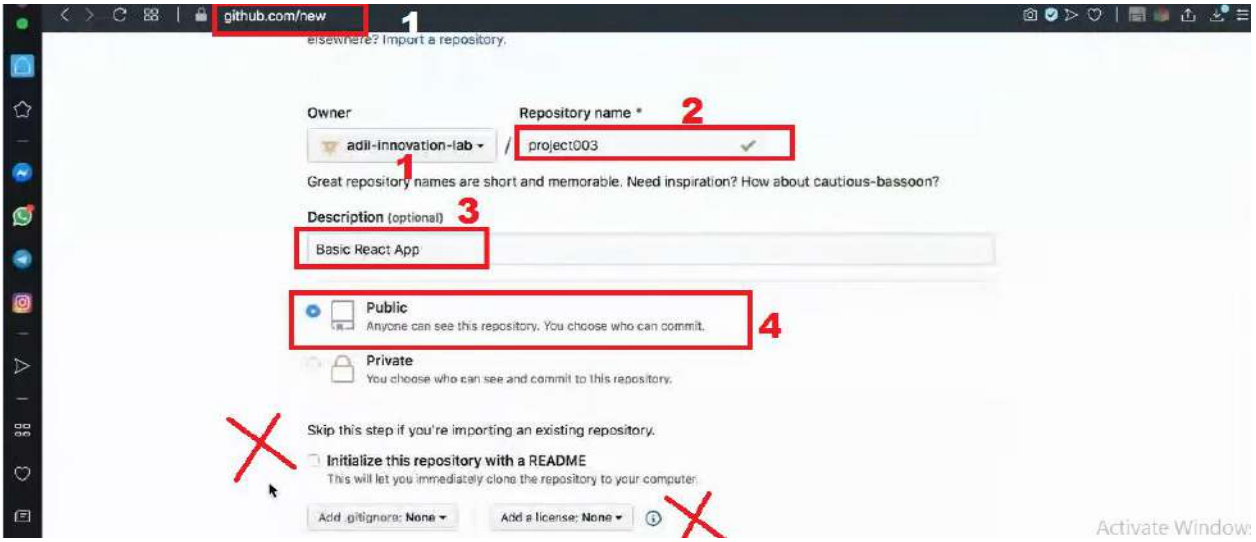


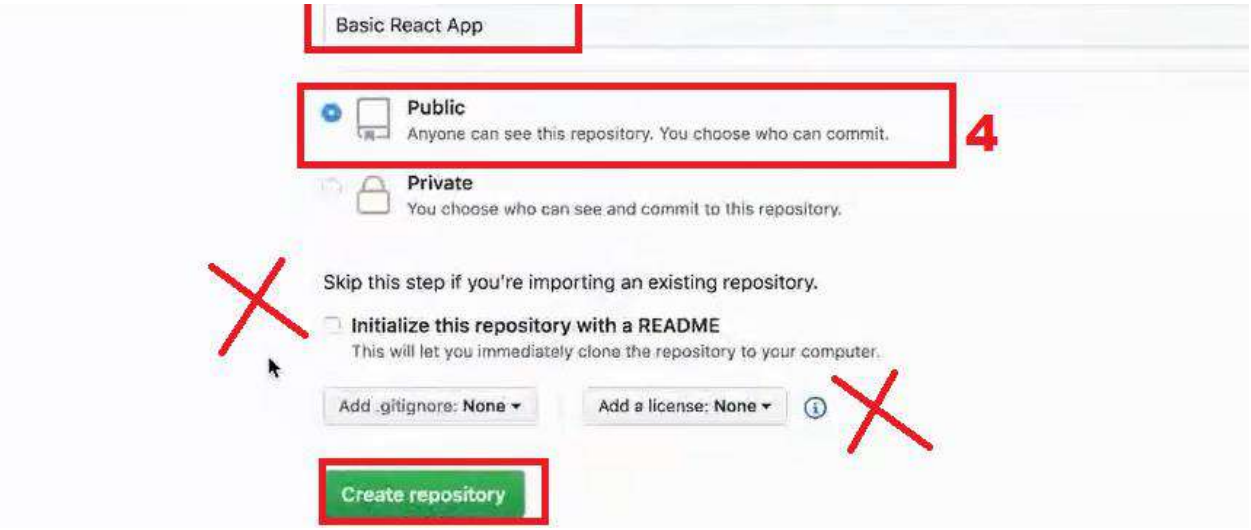
- Now enter the npm command in the terminal to run this.



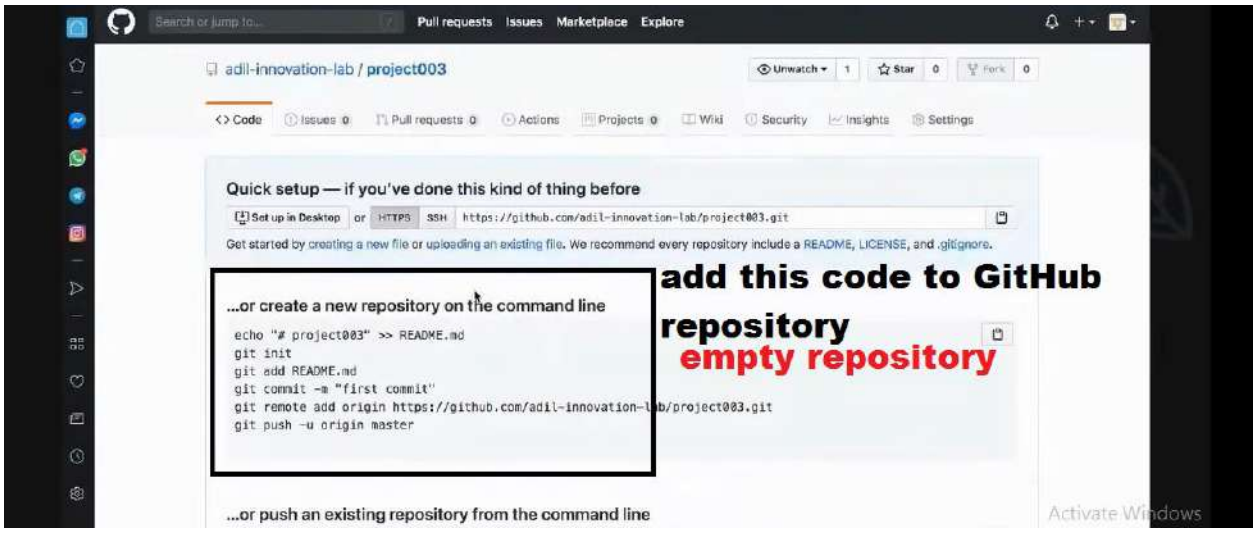
- React application has been successfully created on localhost

- Now go to GitHub.com to open in a browser
- Click to create a new repository
- Enter the new repository name
- Write description
- Click to the public to make it view globally
- No need of authentication as we have opened it in our local host already
- Now click to create a repository





- Now a new repository is opened; add its code to our GitHub repository terminal



- Add git by the following command; git init

- Add git directory; get it add .

```
Adil:project003 adil$ git add .
```

- Add git commit; git commit -m “initial Commit”

```
Adil:project003 adil$ git commit -m "Initial Commit"
On branch master
nothing to commit, working tree clean
```

-

- Add remote repository where we want to push code, locate the origin by writing the URL where this repository is in our device

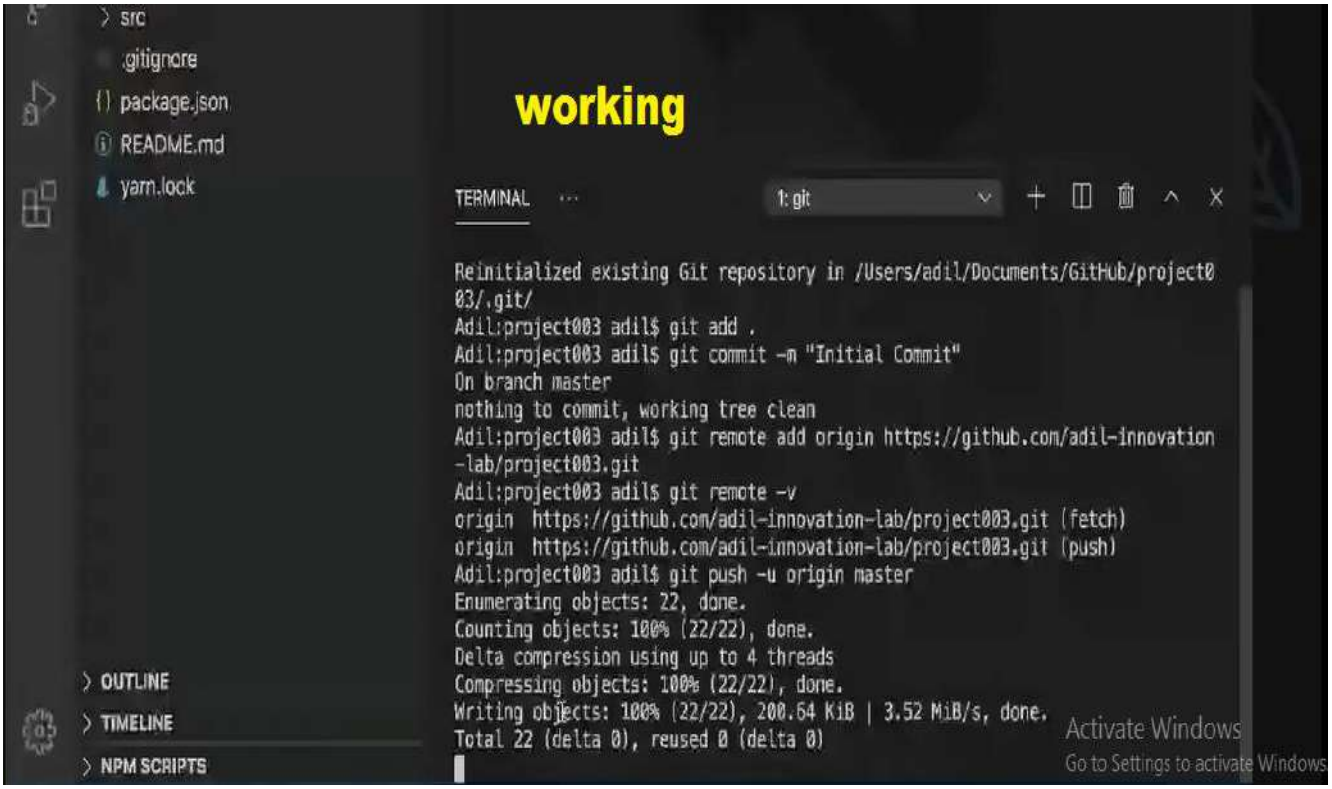
```
Adil:project003 adil$ git remote add origin https://github.com/adil-innovation
-lab/project003.git
```

- verify as to whether this is a valid URL; git remote –v
- do the git push into the origin from the master branch.

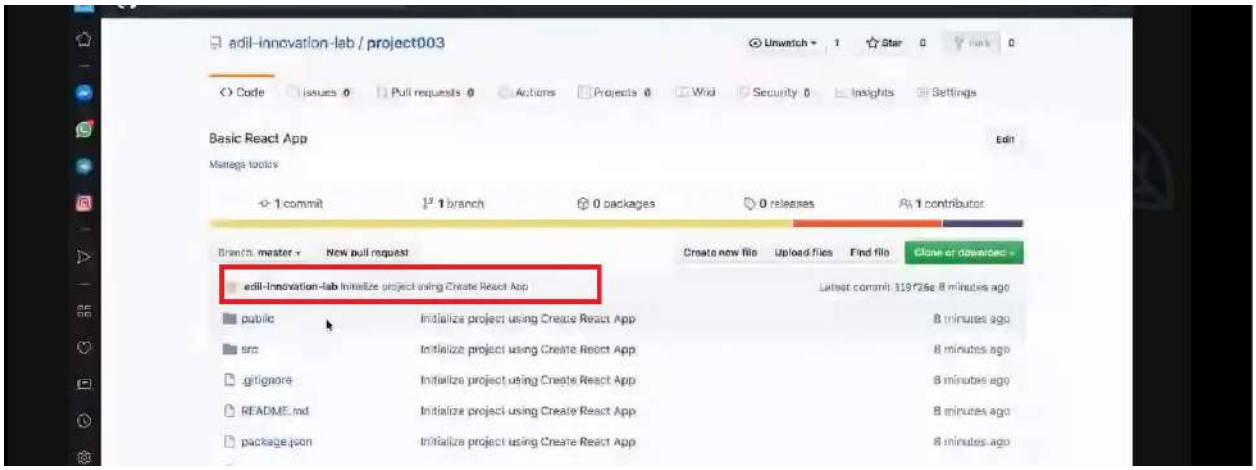
```
Adil:project003 adil$ git remote -v
origin  https://github.com/adil-innovation-lab/project003.git (fetch)
origin  https://github.com/adil-innovation-lab/project003.git (push)
```

```
Adil:project003 adil$ git push -u origin master
```

- It is working



- Go to Github repository which you have created, refresh the page; code has been successfully pushed into the repository



- Click to action to create a workflow
- Create the following workflow:

Name of workflow

```
name: React App Deploy To Surge
```

Event that causes the workflow to execute

```
On:[push]
```

Jobs: that call ubuntu-latest to run that is deploying on surge;

```
Jobs:
    build:
     runs-on:ubntu latest
    name:deploying to
```

Steps To Deploy The Application To Surge

- Check out the code on GitHub repository using an action/GitHub checkout action

```
-uses:action/checkout@v2
```

- Installing node using the setup node action; install node-version:12

```
-name:install Node.js
-uses:actions/setups-
with:
        node version:12
```

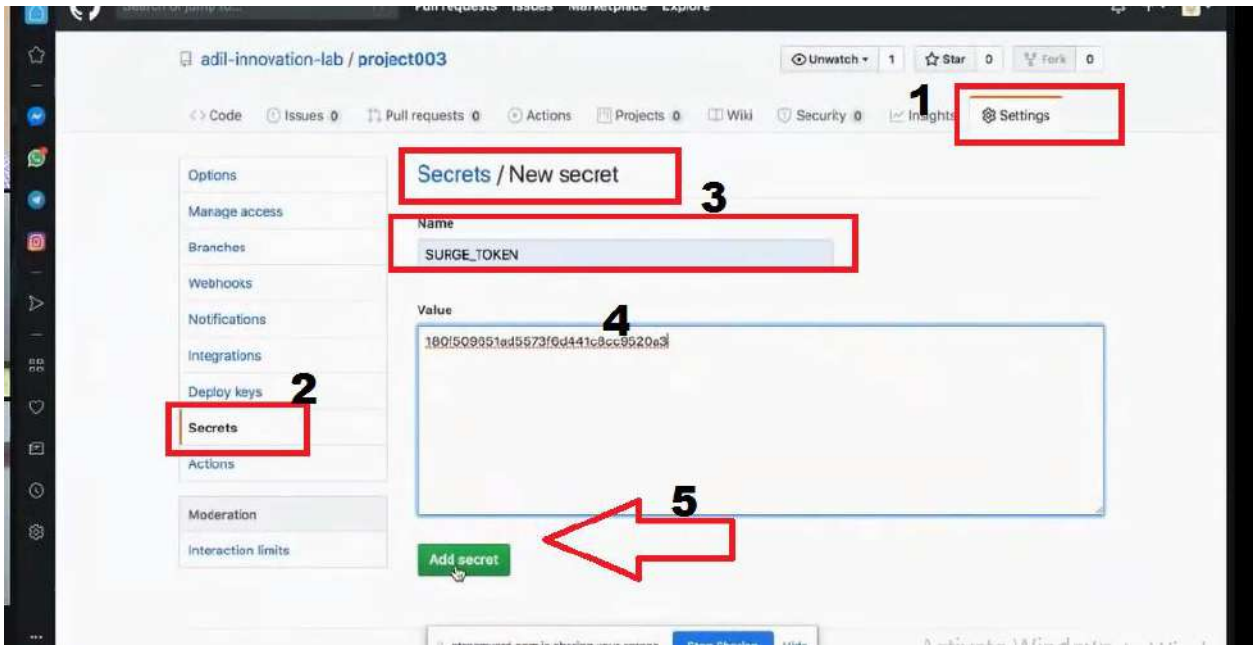
- Install Yarn to build the react application, to build the dependencies, to get the application ready for the run time environment.

```
-name:prepare for yarn installation
```

- Now enter these name, event and actions to install yarn, build react app and yarn build as shown

```
-uses:borales/actions-yarn@2.0.0

 with:

    cmd:install # will run 'yarn install' command

-name: Build React app

 run:yarn build
```

- Deploy surge, run to call the folder where the code is ./build and copy URL where is our directory
- For copying, URL follow following steps
  - Now go to setting
  - Click to secrets
  - Enter secrets name
  - Enter value i.e (take URL from the terminal and copy-paste in value section)
  - Now hit the button press to add secrets.



- run deployment on the surge using the same token as we have done before in 1st tutorial.

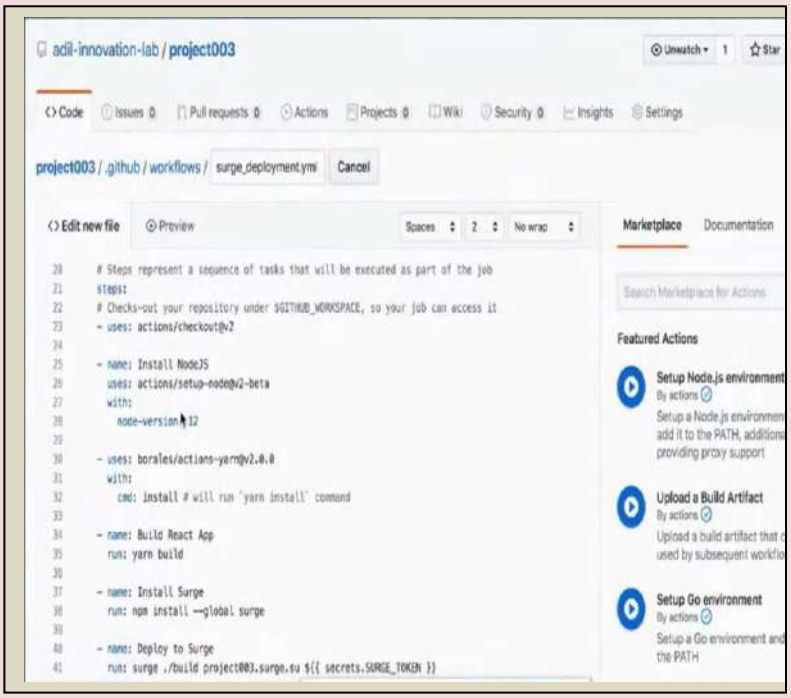
```
-name: Deploy to surge
 run:surge ./build project003.surge.sh --token ${{ secrets. SURGE-TOKEN}}
```

name:React App Deploy To Surge

on:[push]

jobs:

workflow



build:

runs-on:ubntu latest

name:deploying to surge

steps:

-uses:action/checkout@v2

-name:install Node.js

-uses:actions/setups-node@v2-beta

with:

node version:12

-uses:borales/actions-yarn@2.0.0

with:

cmd:install # will run 'yarn install' command

-name: Build React app

run:yarn build

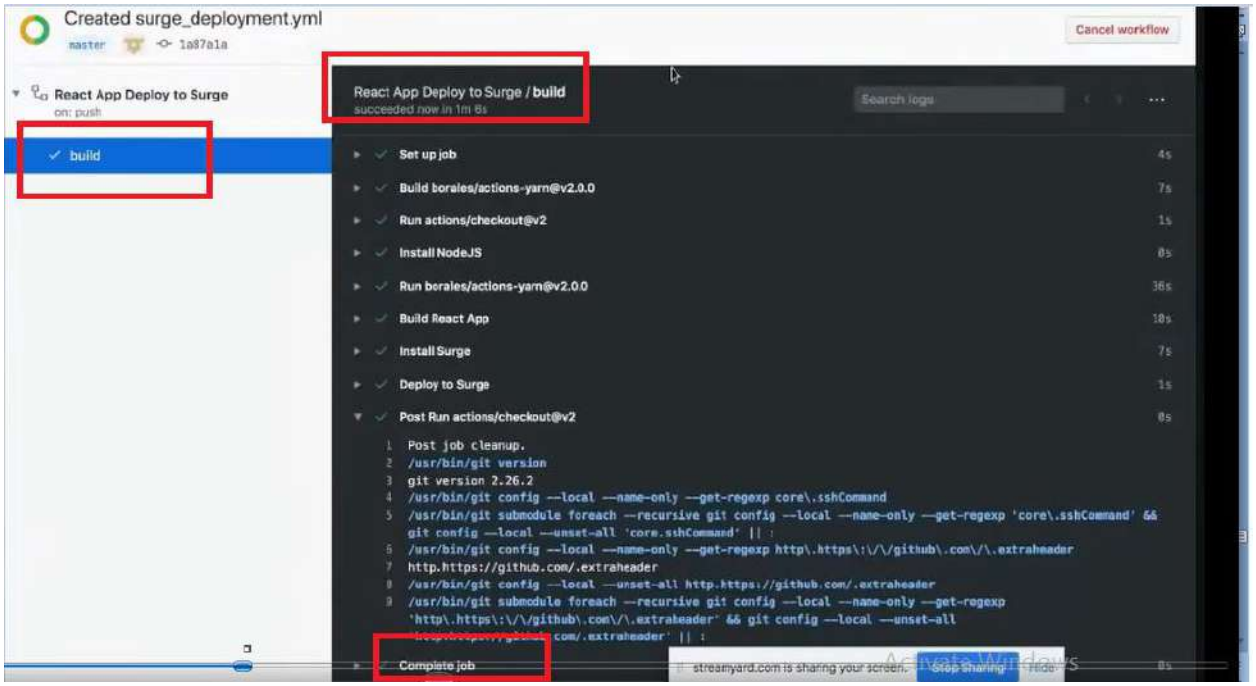
-name: install surge

run:npm install- global surge

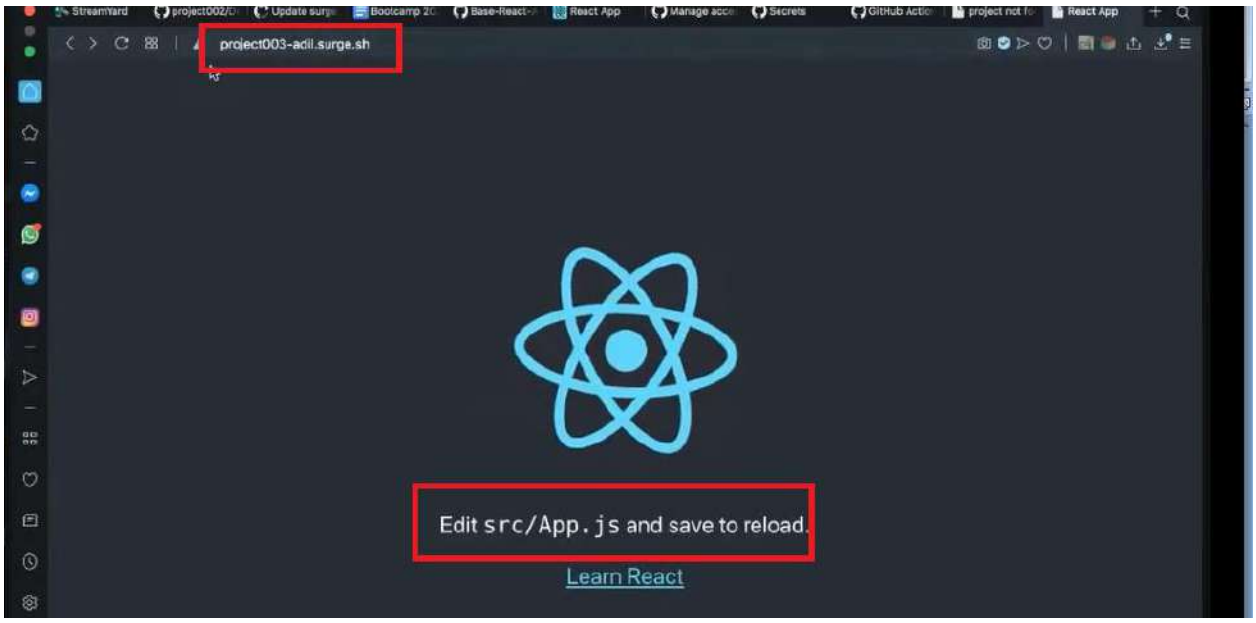
-name: Deploy to surge

run:surge ./build project003.surge.sh--token ${{ secrets. SURGE-TOKEN}}

- Now after completing the workflow
- Click to action in GitHub Repository
- Go to build option
- Checkout all installation of the surge, yarn all has been completed



- **Now go to the browser, enter URL "project003.surge.sh", finally it's been done**



Why Do We Need React

1.2. Why React?

. **1. Simplicity**

ReactJS is very easy to grasp right away. Its component-based approach, well-defined lifecycle, and use of just plain JavaScript let us learn to React very easily and quickly, build a professional web (and mobile applications), and support it. It uses a special syntax called JSX which allows you to mix HTML with JavaScript. It is not so compulsory; Developer can write in plain JavaScript but JSX is much easier to use.

2. Easy To Learn

Anyone with a basic previous knowledge in programming can easily understand React while Angular and Ember are referred to as 'Domain-specific Language', implying that it is difficult to learn them. To react, you just need basic knowledge of CSS and HTML.

3. Native Approach

React can be used to create mobile applications (React Native). And React is a diehard fan of reusability, meaning extensive code reusability is supported. So at the same time, we can make IOS, Android, and Web applications.

4. Data BindingReact uses one-way data binding and an application architecture called Flux controls the flow of data to components through one control point – the dispatcher. It's easier to debug self-contained components of large ReactJS apps.

5. Performance

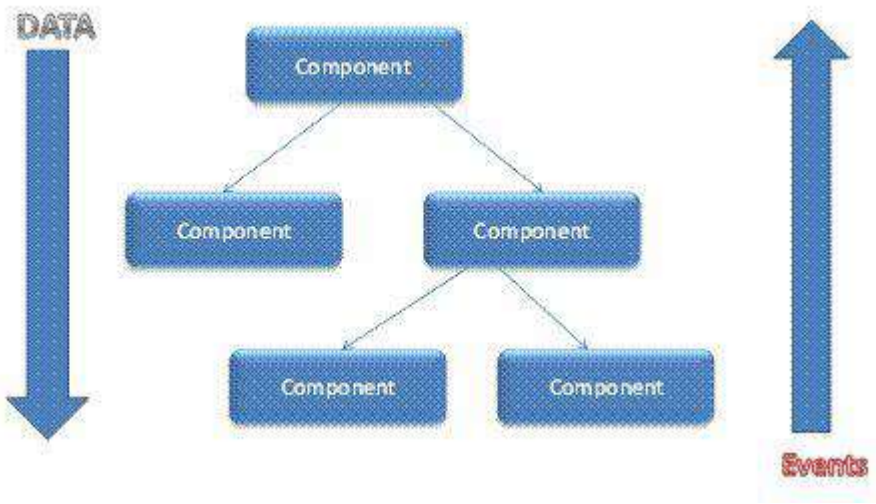
React does not offer any concept of a built-in container for dependency. You can use Browserify, Require JS, EcmaScript 6 modules which we can use via Babel, ReactJS-di to inject dependencies automatically.

6. Testability

ReactJS applications are super easy to test. React views can be treated as functions of the state, so we can manipulate with the state we pass to the ReactJS view and take a look at the output and triggered actions, events, functions, etc.

7. Single-Way Data Flow

In React, a set of immutable values are passed to the components renderer as properties in its HTML tags. The component cannot directly modify any properties but can pass a call back function with the help of which we can do modifications. This complete process is known as “properties flow down; actions flow up”.



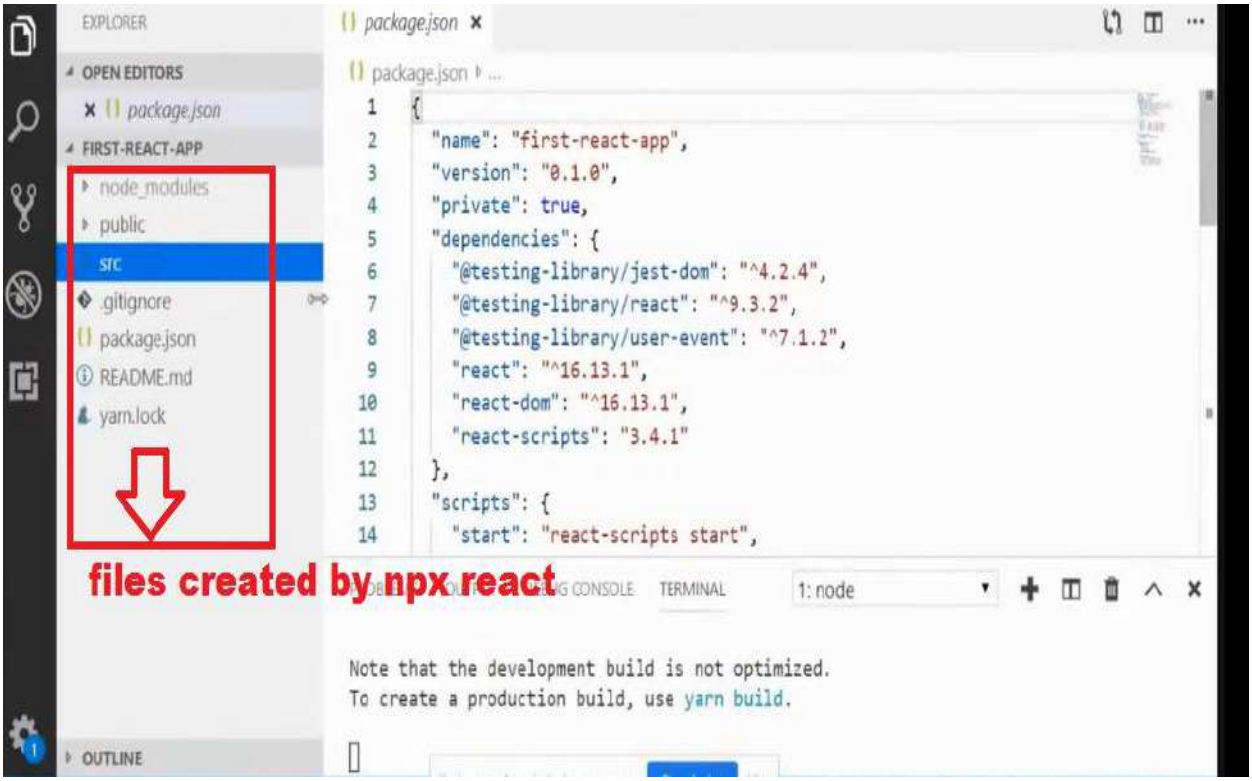
.

Session 3: React_Tutorial

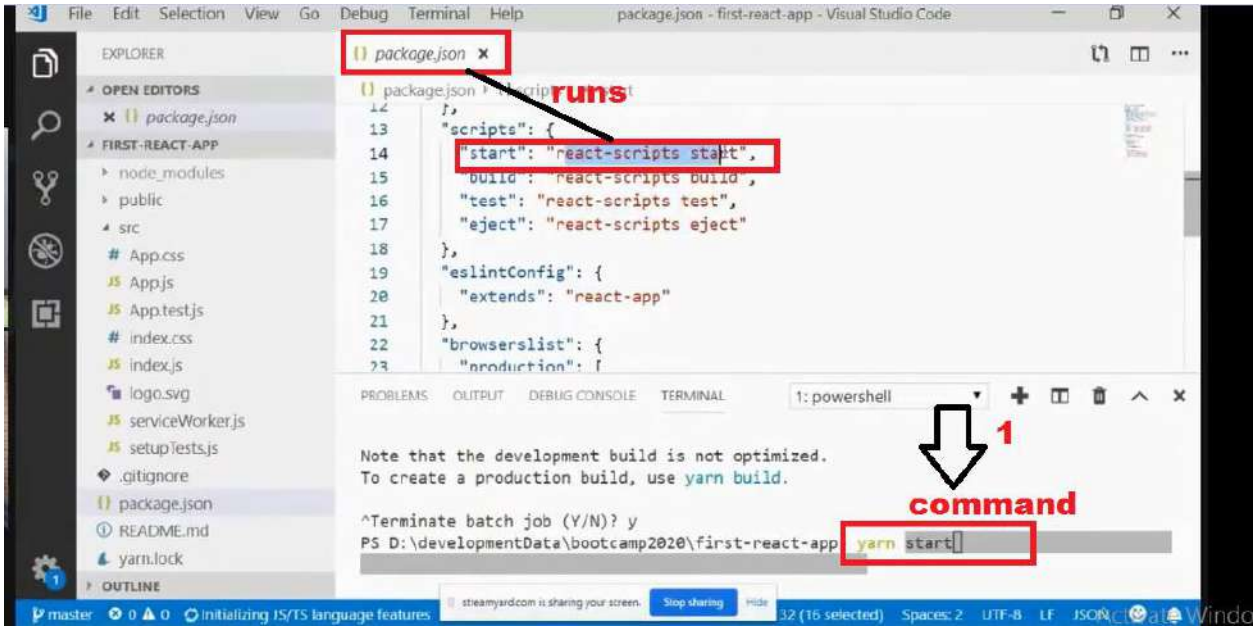
Learning React:

1.1 Simple “Hello World To React “

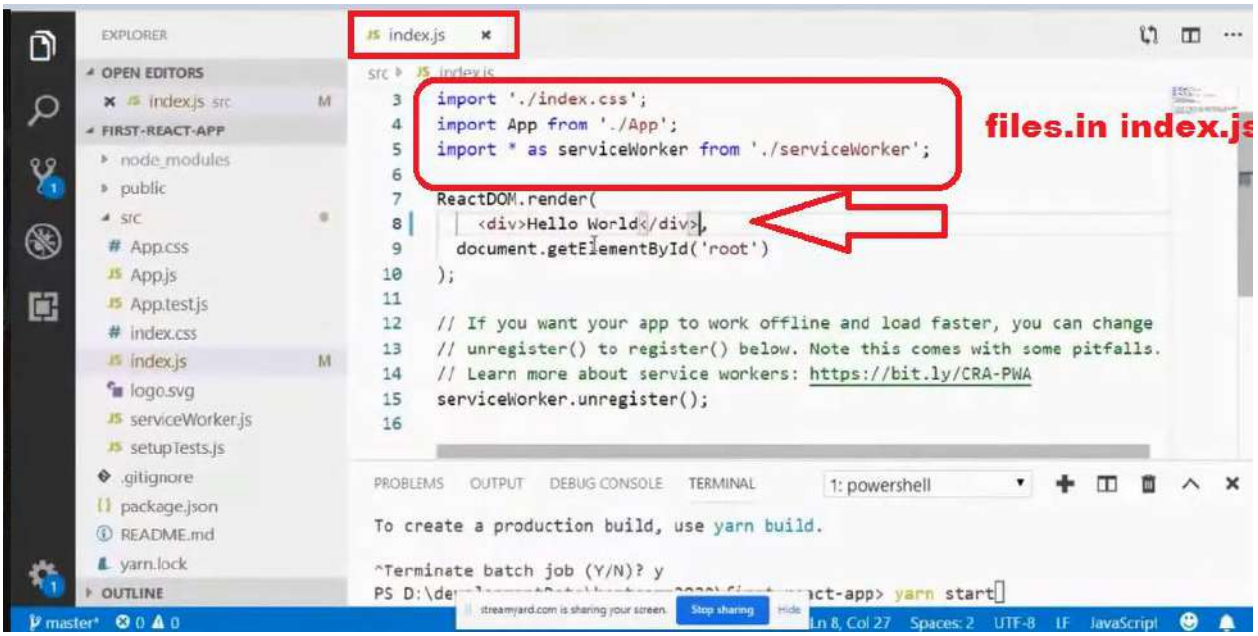
- whenever we create npx react project app it creates following files
  - node.modules
  - public
  - src
  - {}
  - README.md
  - Yarn. Lock



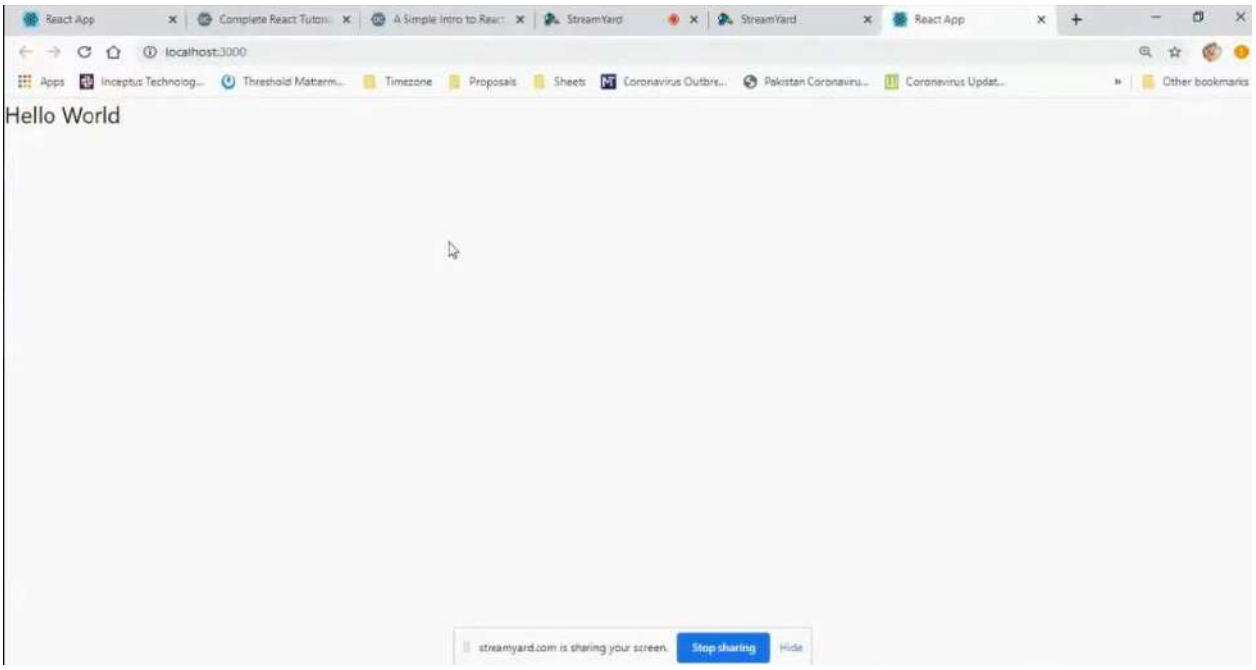
- In order to starrt the application enter this command yarn start
  - (this commands comes from ({}package.json)
  - { }package.json runs **"react_scripts start**"



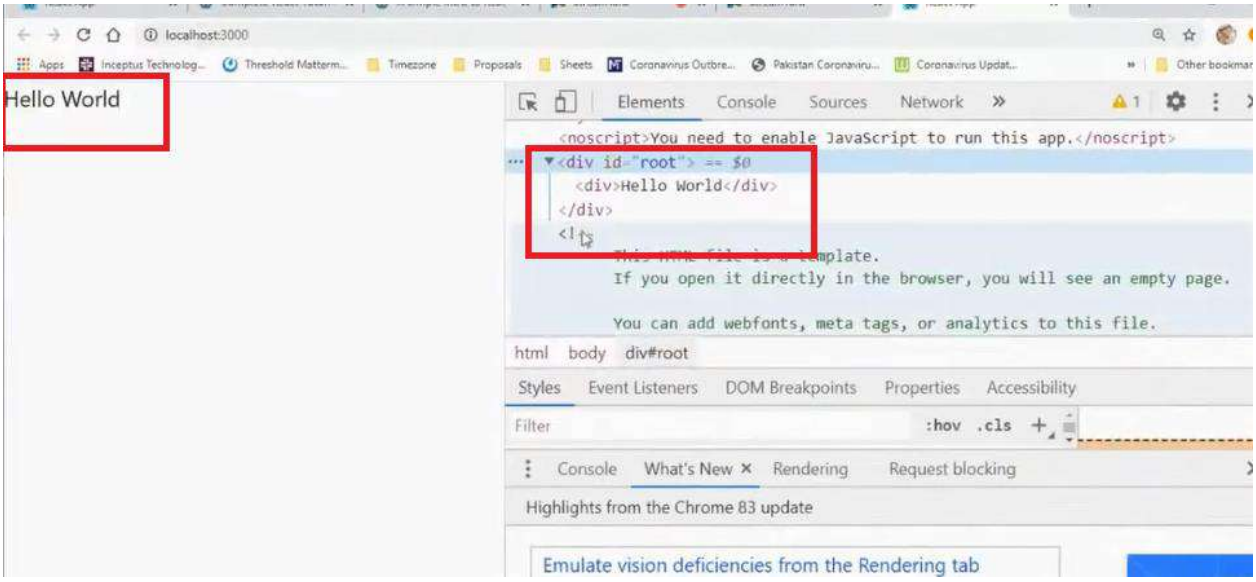
- As soon as' yarn start' commands execute it moves towards index.js file
  - this index.js includes different files in it i.e ./index.css,./app,./serviceworker etc
  - whatever you write in ReactDOM.render; that will be displayed on the screen
- ReactDOM.render requires from Html.index file:
  - div elements and id
  - Html elements and id
  - It takes div elements and call it by its root
- so create a in ReactDOM.render and write "Hello world" in it



- Hit the button press to run yarn start command and look into browser; “Hello world” would have been displayed over the screen



- Whatever we are going to write into ReactDOM.render , that is being added in html and being displayed over our browser

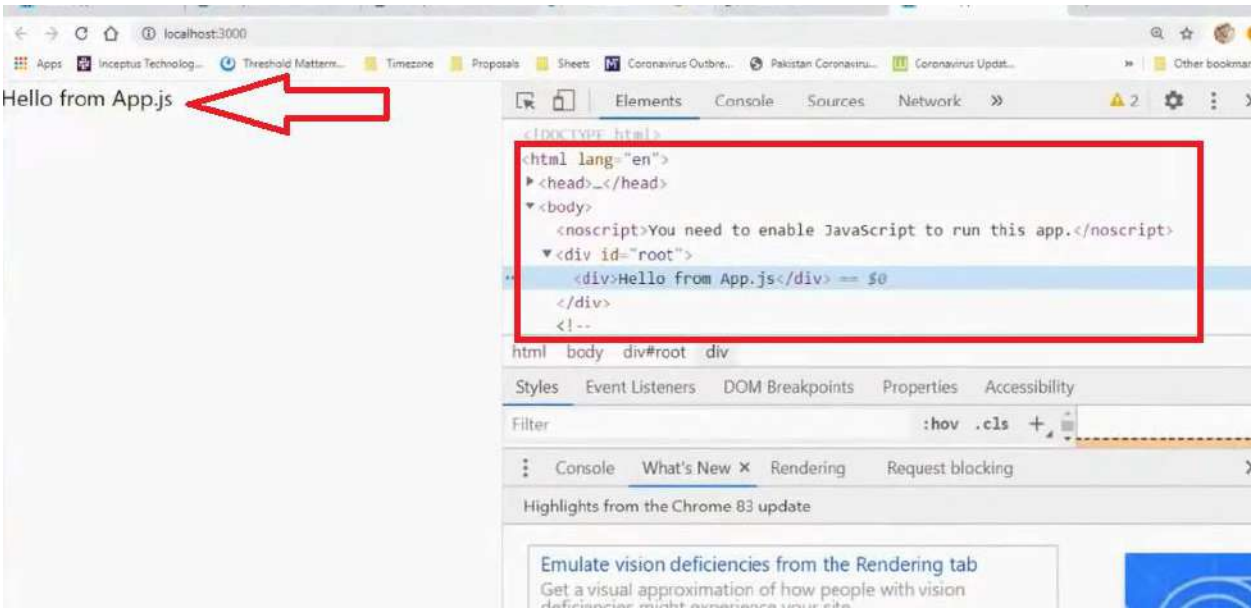


1.2. Adding Properties To Simple Hello world

- **build first coomponent wit app.js**

- its basic structure is

```
import React from 'react';

import './App.css';

function App() {

 return   <div>hello from App.js</div>

   }

export default App;
```



Now if you want to add kind of attribute in react (i.e you want to add name zia khan) then you will have add **name="zia"** in the div section **of ReactDOM.render in index.js file**

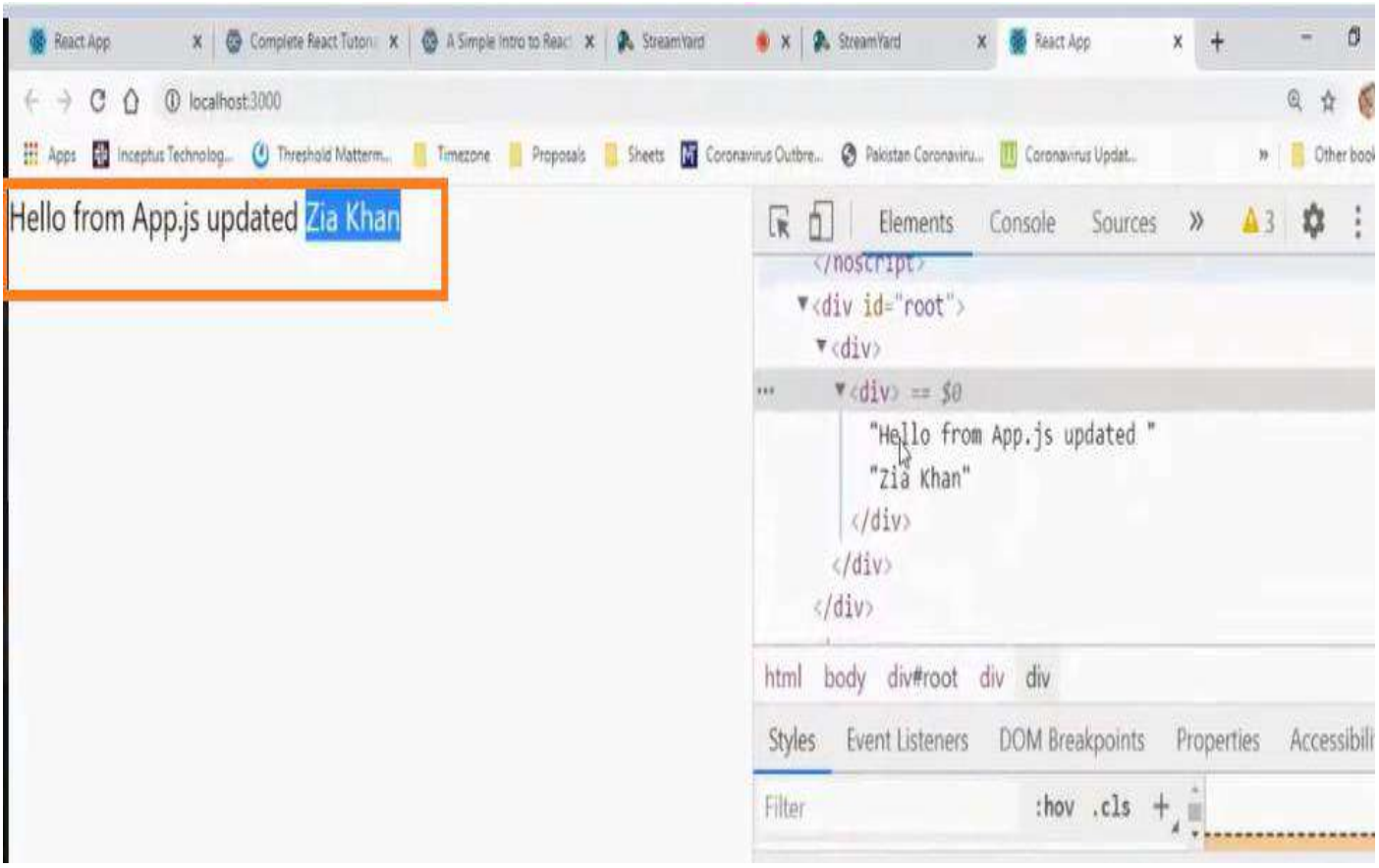
This will only executes if you add probs paramet is declared in **Function app [props]** in App.js file

App.js file

```
import './App.css'/

function App(props) {

  <div>

      Hello from App.js updated {props name}

  </div>

}

export default App;
```

Index.js file

```
ReactDom.render(

<div><App name="zia khan" /></div>

document.getElementByid('root')

);
```



- **You can add as many attributes as you want only you have to declare the probs in App.js file and have to write the attribute name=its value in inverted commas or in curly brackets that's it**

FOR example if you want to add properties of age with name then follow up following code

App.js file

```
import './App.css';

function App(props){

  <div>

      Hello from App.js updated {props.
name} Age={props.age}

  </div>

}

export default App;
```

Index.js file

```
ReactDom.render(

<div><App name="zia khan" age="25"
/></div>

document.getElementByid('root')

);
```

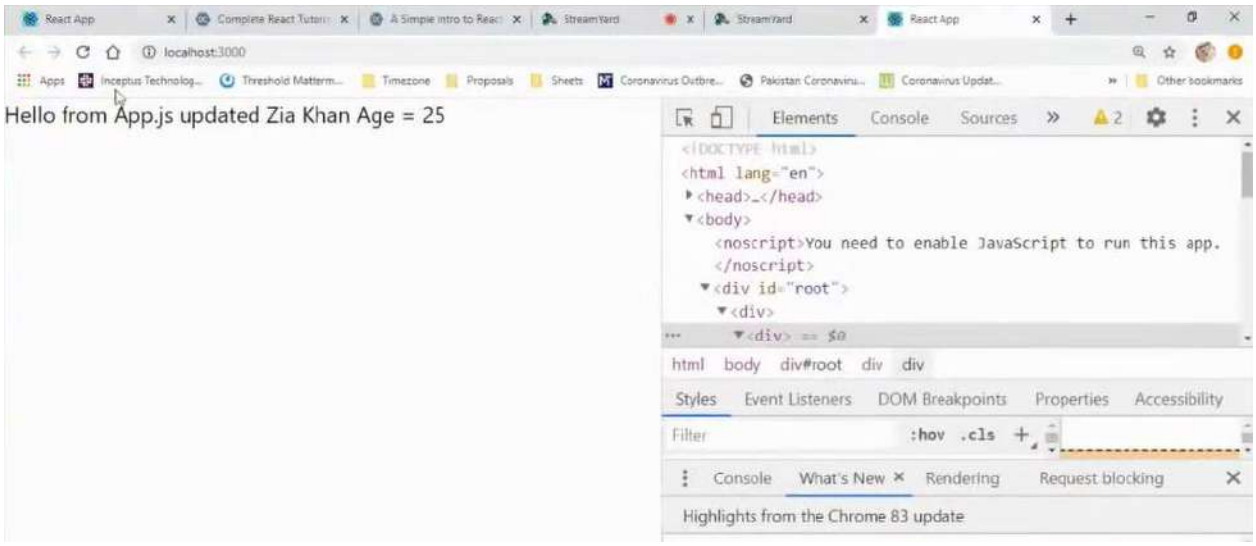
App.js file

```
import './App.css';

function App(props) {
  return <div>
          Hello from App.js updated {props.name} Age = {props.age}
         </div>
}

export default App;
```

Index.js file

```
ReactDOM.render(
  <div><App name="Zia Khan" age="25" /></div>,
  document.getElementById('root')
);

/*
ReactDOM.render(
  React.createElement('div',null,React.createElement('span',null,"Hello i
  document.getElementById('root')
);
```

After save changes ,refresh you browser page you will get edited of age with name



2.1. Different ways of writing a function

plain function

```
function Hi({name}){
return =<div Hello {nmae}!</div>;
}
```

A constant Holding An Anonymous funtion

```
cons Hi =function({name}){
return <div>Hello {name}!</div>:
}
```

Turning Function into an arrow

```
Const Hi=({name})=>{
return <div>Hello {name}!</div>;
}
```

removing the space

```
const Hi=({name})=>(
<div> Hello {name}!</div>
)
```

skipping Paranthesis

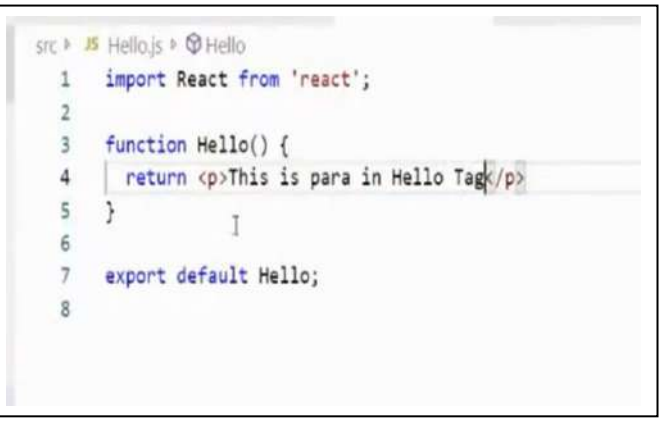
```
const Hi =({name}) =<div> Hello {name}!</div>
```

Create another component Hello.js

3. (Tree Creation)

In this we need to create a function and export it

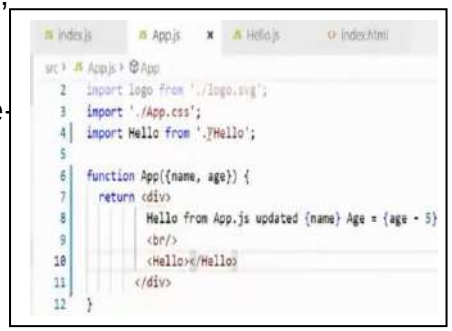
```
import React from 'react';

function Hello() {

 return <p>This is para in Hello Tag</p>

}

export default Hello;
```

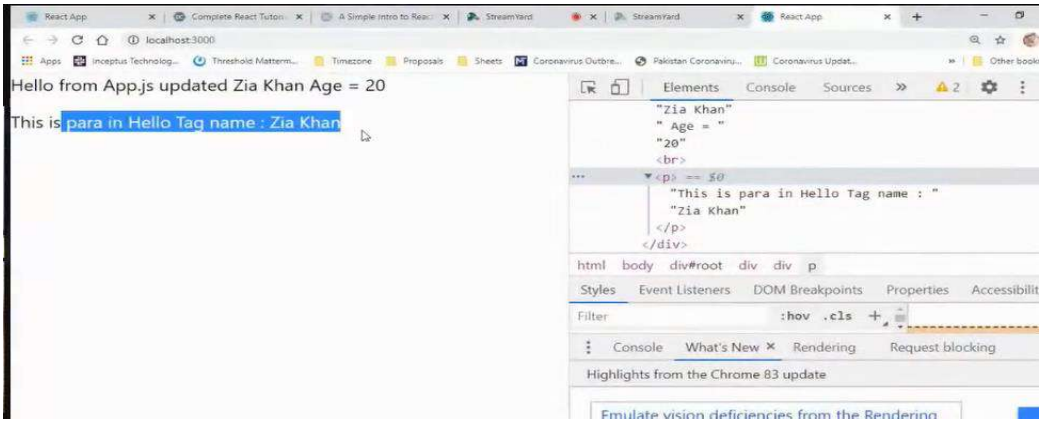


3. Tree Working

- **Now in app.js use this hello.js component for that purpose import Hello from '/Hello';**

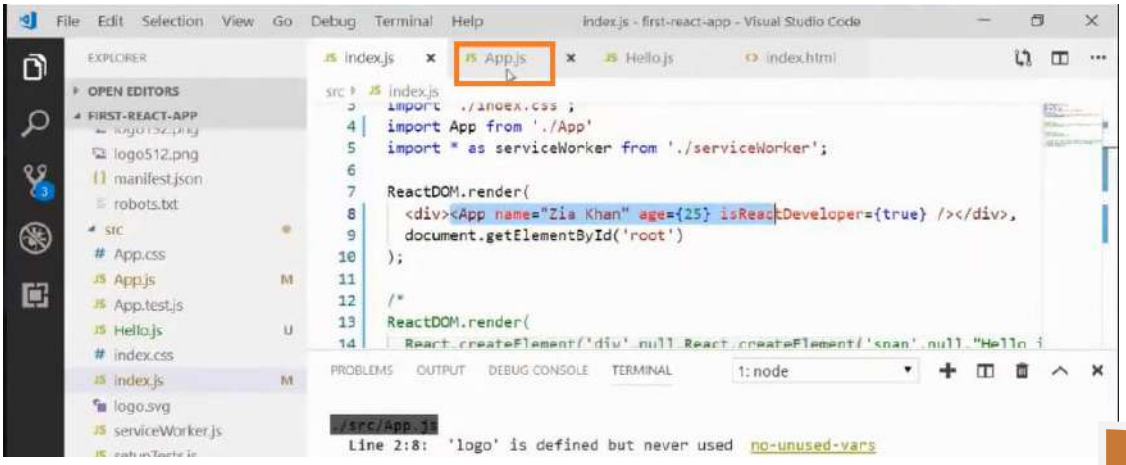
```
 function App({name, age}) {
return <div>
        hello from App.js updated{name}age={age
<br/>
<Hello></Hello>
</div>
}
```





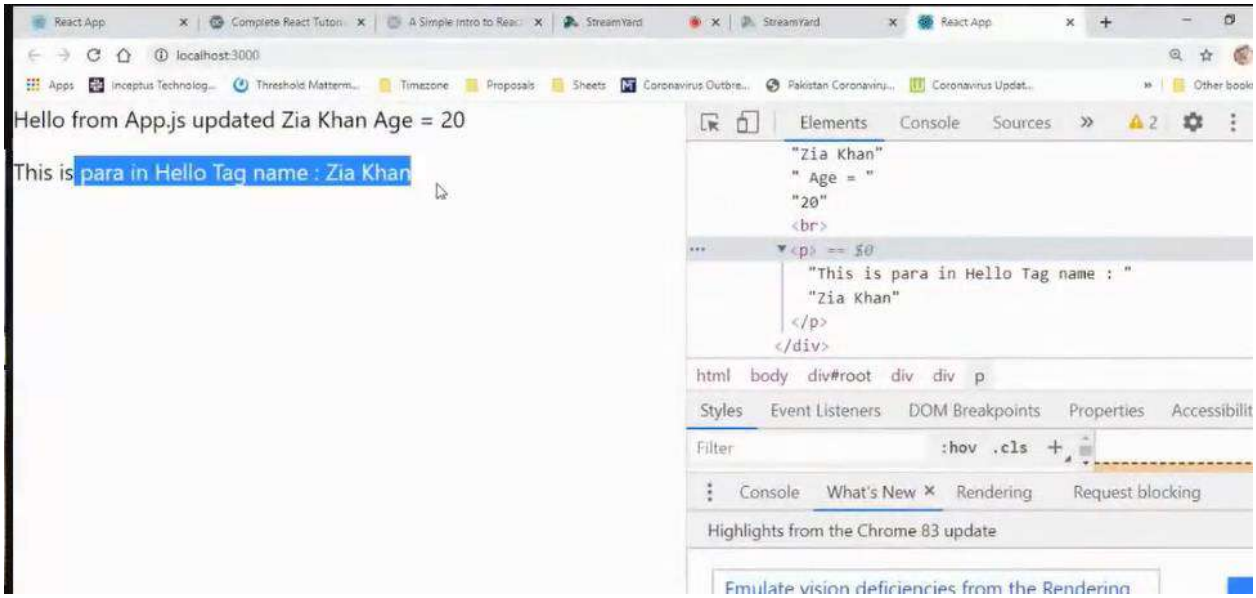
3.1 This is how a tree work.

- **app.js starts devloping strusture , it recives the div that contain app tag , this app tag include the hello tag that will be converted into p tag**



app.js starts devloping strusture , it recives the div that contain app

this app tag include the hello tag that will be converted into p tag



4. CSS Style implementation

- create another component hello.css as (give class name as".myname"

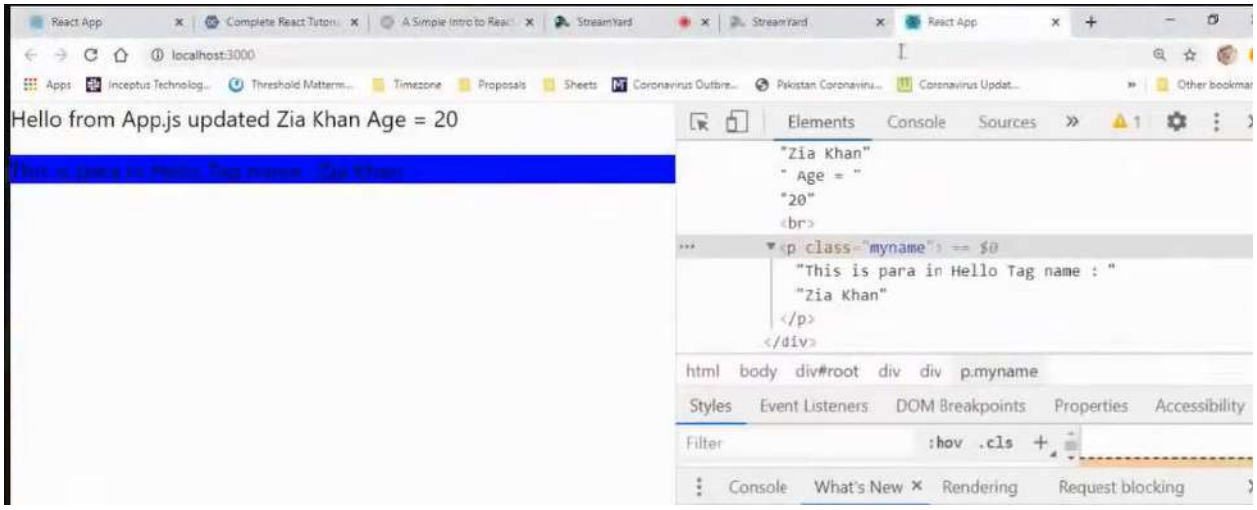
```
.myname{
   background-color: blue;
 }
```

- Now import this css file to hello.js(in p tag introduce class=myname)

```
 import React from 'react';
import './Hello.css';
function Hello ({firstName}){
return <p class="myname"> this is para in Hello Tag name:{firstName}
}
export default Hello;
```

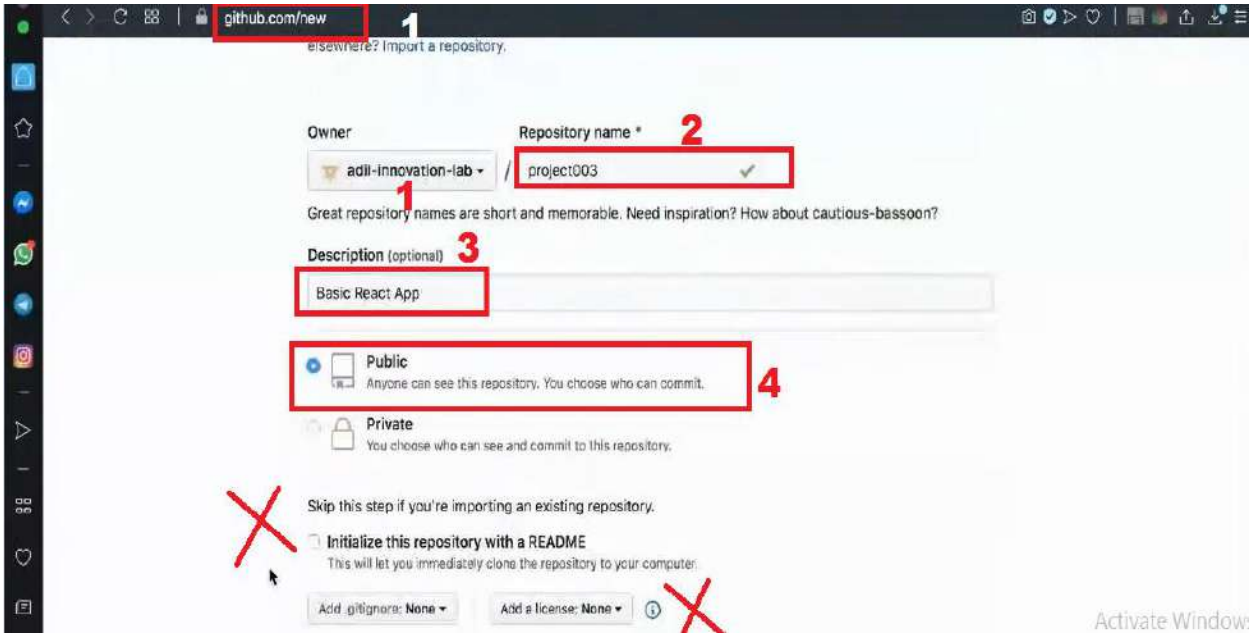
```
src > JS Hello.js > ...
1  import React from 'react';
2  import './Hello.css'
3
4  function Hello({firstName}) {
5    return <p className="myname" >This is para in Hello Tag name : {firstN
6  }
7
8  export default Hello;
9
```

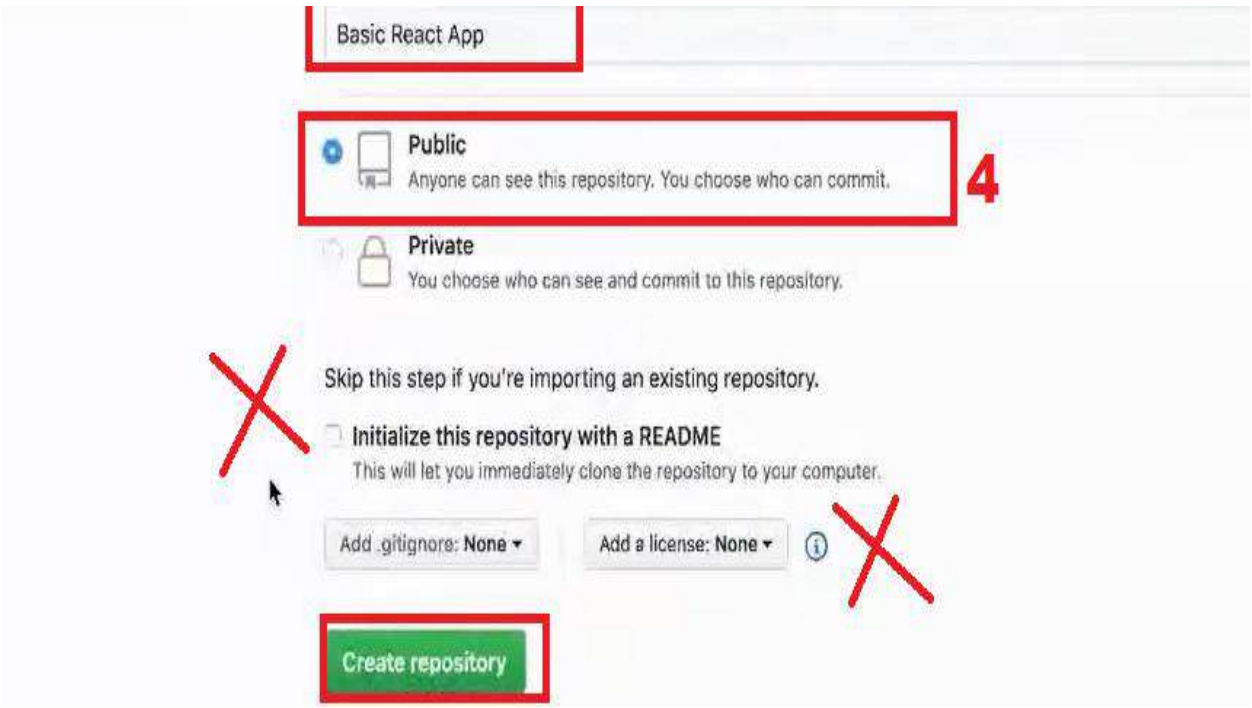
❖ After saving changes refresh your browser page ; css styl in blue color of paragraph has been implemented



5. Repository Creation for Code And Its implementation

- Now go to GitHub.com to open in a browser
- Click to create a new repository
- Enter the new repository name
- Write description
- Click to the public to make it view globally
- No need of authentication as we have opened it in our local host already
- Now click to create a repository


github.com/new
1
elsewhere? Import a repository.
Owner
Repository name *
2
adli-innovation-lab
/
project003
1
Great repository names are short and memorable. Need inspiration? How about cautious-bassoon?
Description (optional)
3
Basic React App
Public
Anyone can see this repository. You choose who can commit.
4
Private
You choose who can see and commit to this repository.
Skip this step if you're importing an existing repository.
Initialize this repository with a README
This will let you immediately clone the repository to your computer.
Add .gitignore: None
Add a license: None
Activate Window


Basic React App
Public
Anyone can see this repository. You choose who can commit.
4
Private
You choose who can see and commit to this repository.
Skip this step if you're importing an existing repository.
Initialize this repository with a README
This will let you immediately clone the repository to your computer.
Add .gitignore: None
Add a license: None
Create repository

- Now a new repository is opened; add its code to our GitHub repository terminal

- Add git by the following command; git init

```
TERMINAL    ...    1: bash

Adil:project003 adil$ git init
Reinitialized existing Git repository in /Users/adil/Documents/GitHub/project0
03/.git/
```

- Add git directory; get it add .

```
Adil:project003 adil$ git add .
```

- Add git commit; git commit -m “initial Commit”

```
git commit –m  "update app.js"
```

- Add remote repository where we want to push code, locate the origin by writing the URL where this repository is in our device

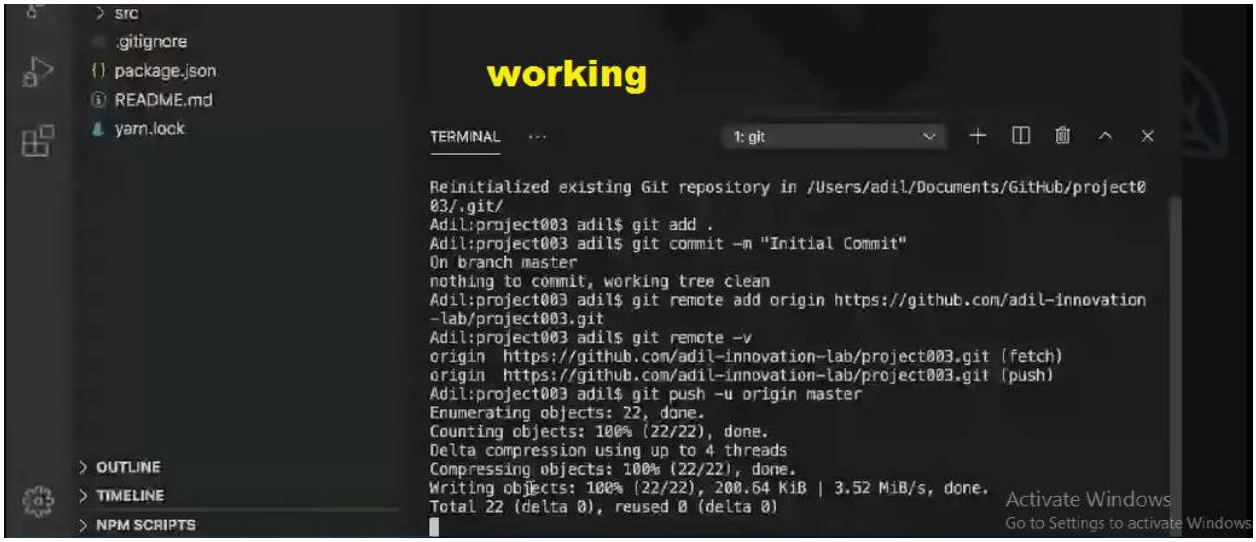
```
Adil:project003 adil$ git remote add origin https://github.com/adil-innovation
-lab/project003.git
```

- verify as to whether this is a valid URL; git remote –v
- do the git push into the origin from the master branch.

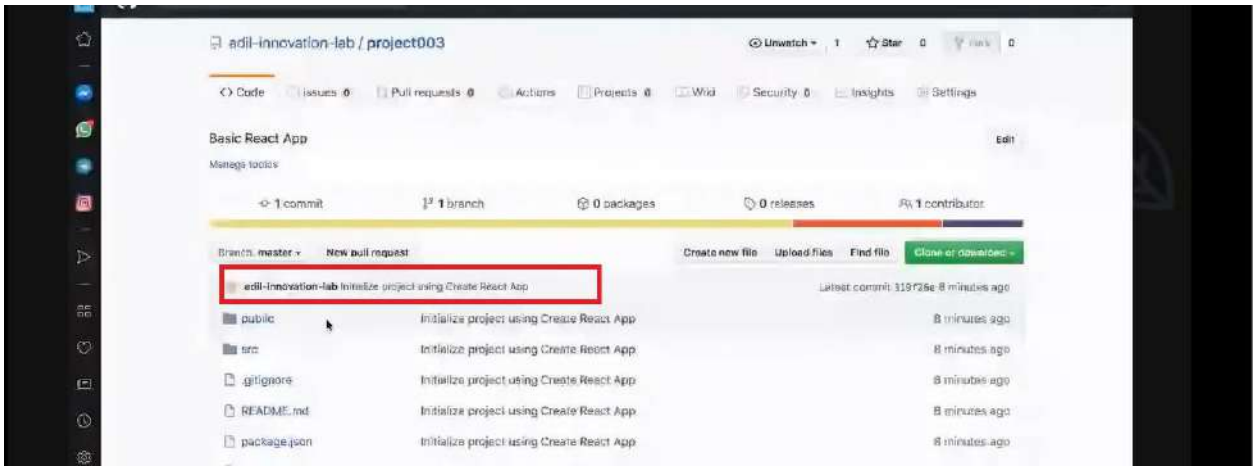
```
Adil:project003 adil$ git remote -v
origin  https://github.com/adil-innovation-lab/project003.git (fetch)
origin  https://github.com/adil-innovation-lab/project003.git (push)
```

```
Adil:project003 adil$ git push -u origin master
```

- It is working



- Go to Github repository which you have created, refresh the page; code has been successfully pushed into the repository



- Click to action to create a workflow
- Create the following workflow:

Name of workflow

```
name: React App Deploy To Surge
```

Event that causes the workflow to execute

```
On:[push]
```

Jobs: that call ubuntu-latest to run that is deploying on surge;

```
Jobs:
    build:
     runs-on:ubntu latest
    name:deploying to
surge
```

Steps To Deploy The Application To Surge

- Check out the code on GitHub repository using an action/GitHub checkout action

```
-uses:action/checkout@v2
```

- Installing node using the setup node action; install node-version:12

```
-name:install Node.js

-uses:actions/setups-

with:

        node version:12
```

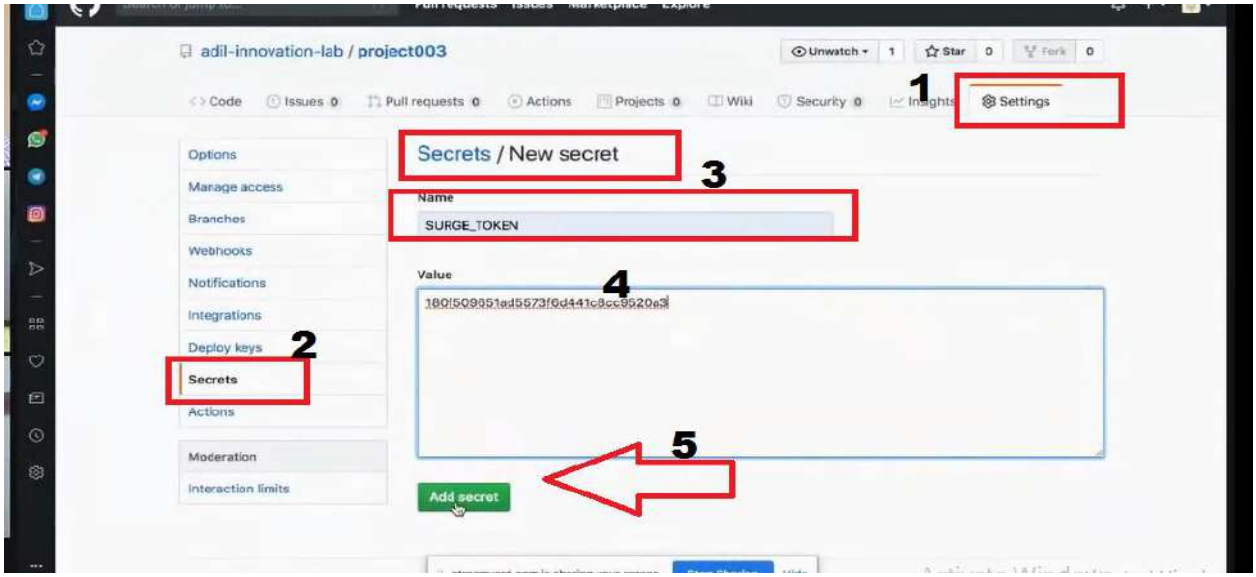
- Install Yarn to build the react application, to build the dependencies, to get the application ready for the run time environment.

```
-name:prepare for yarn installation
```

- Now enter these name, event and actions to install yarn, build react app and yarn build as shown

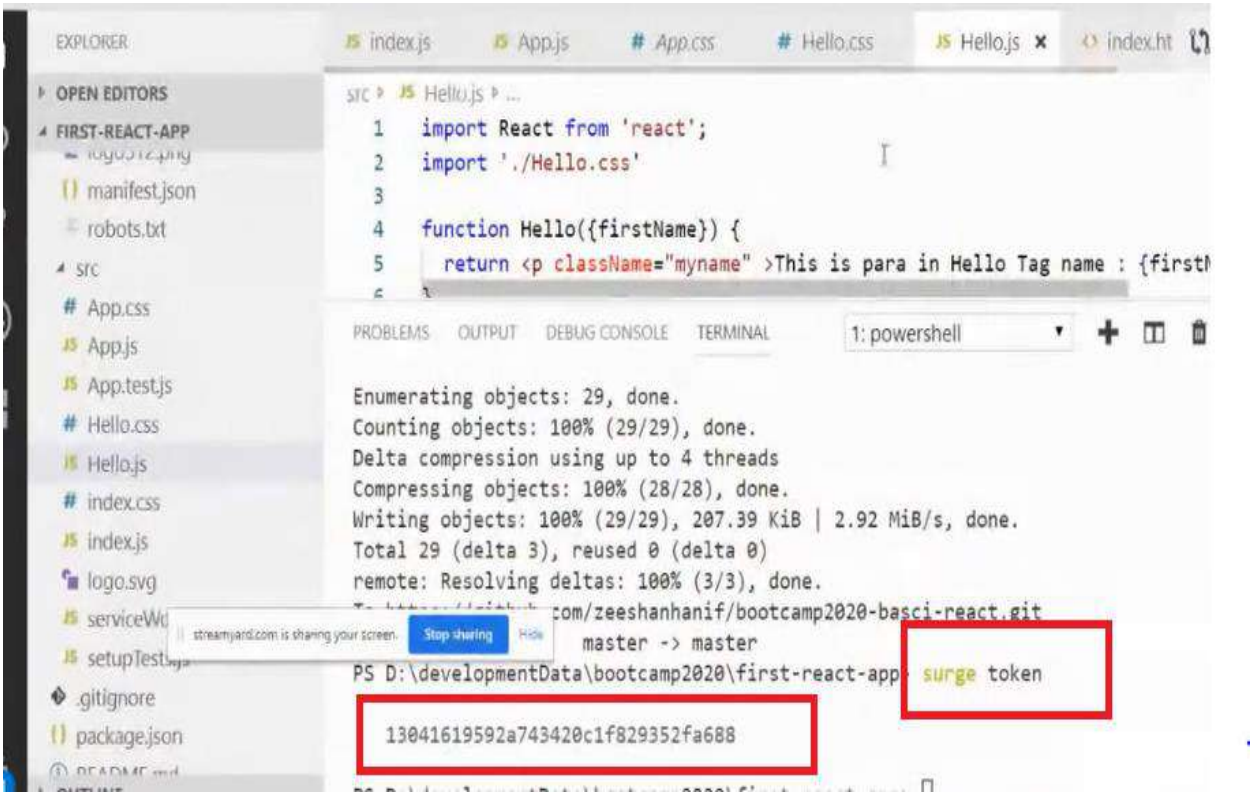
```
-uses:borales/actions-yarn@2.0.0

  with:

     cmd:install # will run 'yarn install' command

-name: Build React app

 run:yarn build
```

- Deploy surge, run to call the folder where the co is ./build and copy URL where is our directory
- For copying, URL follow following steps
  - Now go to setting
  - Click to secrets
  - Enter secrets name
  - Enter value i.e (take URL from the terminal and copy-paste in value section)
  - Now hit the button press to add secrets.



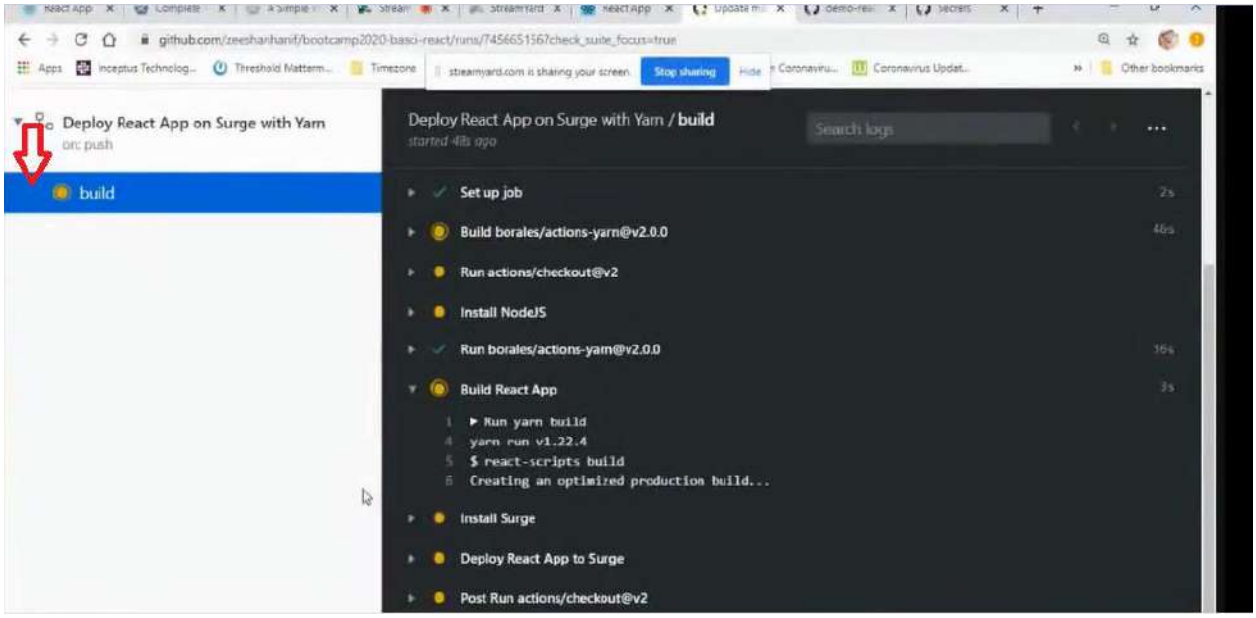
➢ run deployment on the surge using the same token as we have done before in 1st tutorial.

```
-name: Deploy to surge

run:surge ./build project003.surge.sh --token ${{ secrets. SURGE-TOKEN}}
```

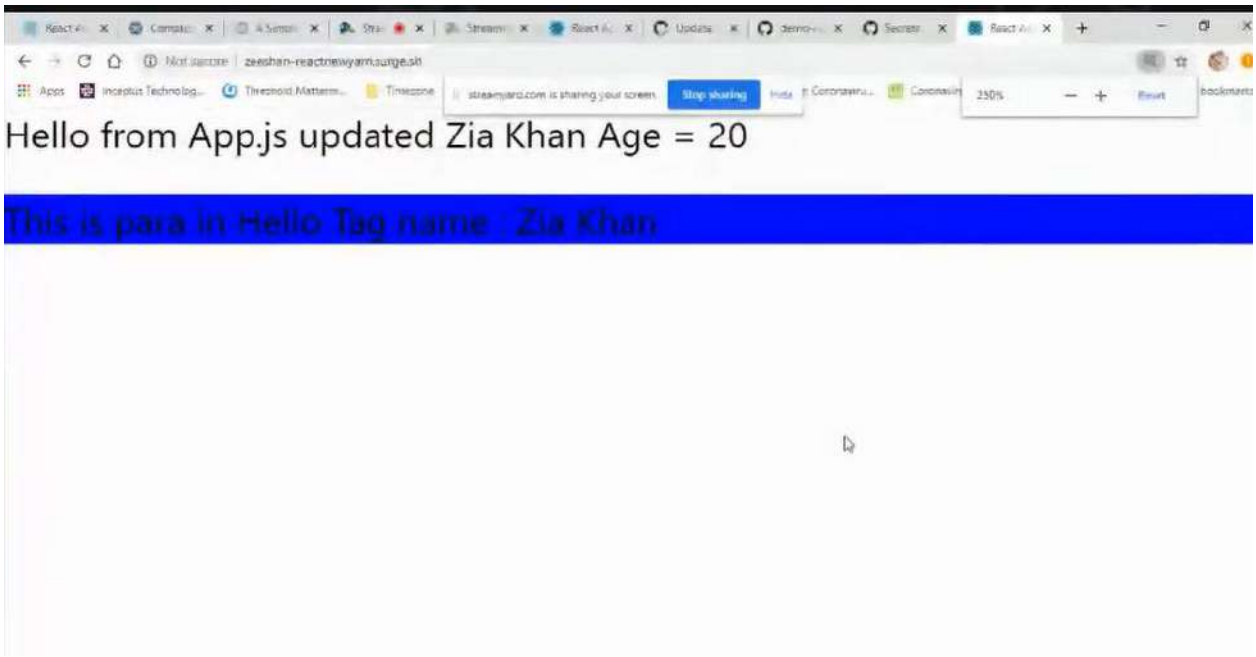


```
name:React App Deploy To Surge
on:[push]
jobs:
  build:
    runs-on:ubntu latest
    name:deploying to surge
    steps:
      -uses:action/checkout@v2
      -name:install Node.js
      -uses:actions/setups-node@v2-beta
       with:
          node version:12
      -uses:borales/actions-yarn@2.0.0
        with:
            cmd:install # will run 'yarn install' command
      -name: Build React app
       run:yarn build
      -name: install surge
       run:npm install- global surge
      -name: Deploy to surge
        run:surge ./build project003.surge.sh--token ${{ secrets. SURGE-TOKEN}}
```

- Now go to action click build it will start running all installation and code executions thereby



- **After its completion go to browser ,refresh your page and look all deployements you made has been displayed on the screen**



Session 4: React_Tutorial

1. What is State?

The state is an instance of React Component Class can be defined as an object of a set of **observable** properties that control the behavior of the component. In other words, the State of a component is an object that holds some information that may change over the lifetime of the component.

1.2. Example Of State; Let's Look Into Deep

Conveniently React has a feature called **state** which allows components to keep track of values that can change – a perfect place to store the state of our light.

To add state to our Room component, we can either turn it into a **class component** or add state directly to the function with **hooks**. We'll call React's useState hook to create a piece of state to hold the lightswitch value:

```
function Room() {
  const [isLit, setLit] =
React.useState(true);

  return (
    <div className="room">the
room is lit</div>
  );
}
```

We've added the call to React.useState at the top, passing in the initial state of **true**. It returns an array with two entries - the first is the state value itself (so, isLit will be true) and the second is a function to change the state.

We're using the *array destructuring* syntax [isLit, setLit] to break out those two entries and give them names. This syntax is ES6 JavaScript, by the way – not React-specific. It's exactly the same as if we'd written it out like this:

```
const state = React.useState(true);
const isLit = state[0];
const setLit = state[1];
```

2. How To Import useState

Quick aside! You'll often see useState imported directly from React like this:

```
import React, { useState } from 'react';
```

After doing that, you can call useState directly (no need to qualify it with React.useState):

```
const [isLit, setLit] = useState(true);
```

Either way is fine, but it's more common to import it. I did it the other way to show it works both ways, and to avoid interrupting the example by having you change the existing import, and now I've gone and added a whole section interrupting you about it. Anyway. Onward!

3. Render Based on State

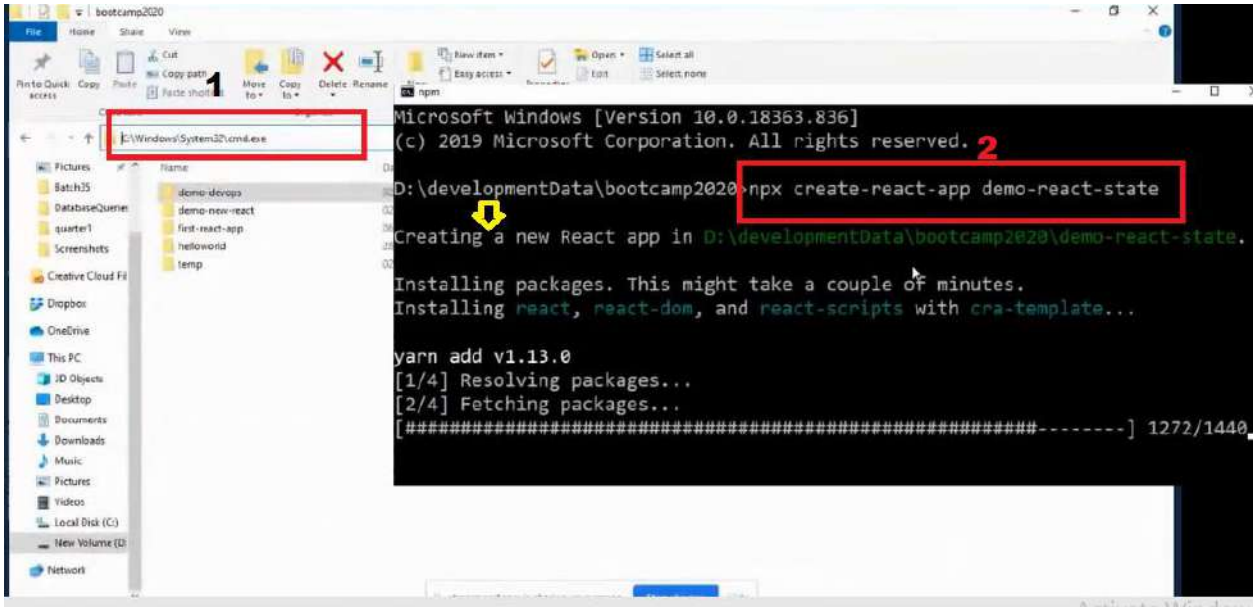
Right now, the component works the same as before, because we haven't changed anything in the actual JSX that's being rendered. Let's have it render differently based on the state of the light. Change the to this:

```
<div className="room">
  the room is {isLit ? 'lit' : 'dark'}
</div>
```

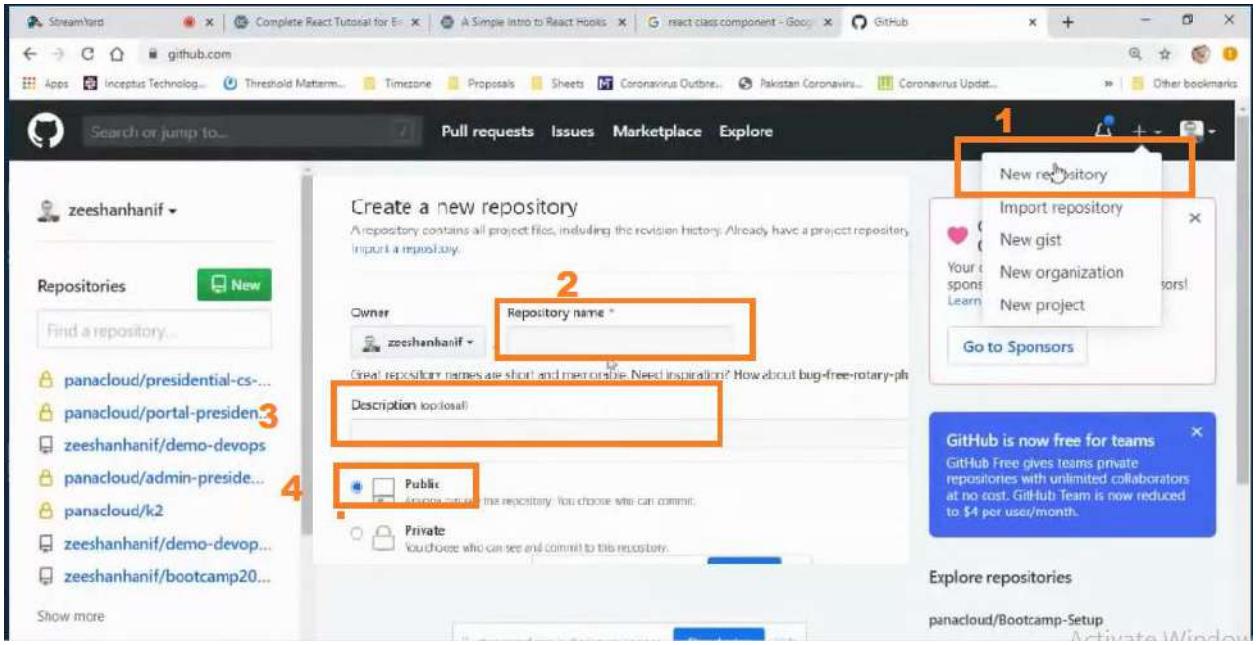
As you can see, the light is still on. Now change React.useState(true) to React.useState(false). The app will re-render saying "the room is dark."

4. Creation Of A Component

- go to the folder where you have installed the GitHub repository
- go to terminal enter cmd
- now give command of npx create-react-app demo-react-state for creating new React app



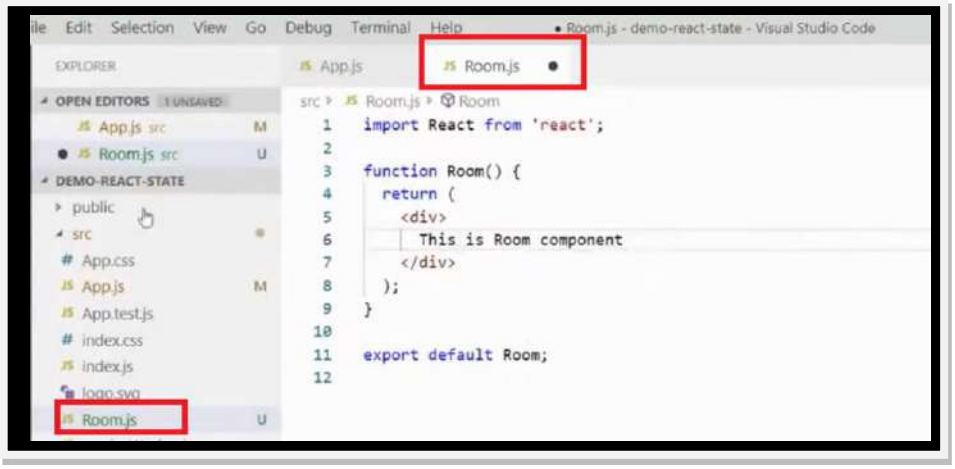
- side by goto Github.com click to + icon and create a new repository
- enter name and description
- click to public
- no need of authentication to be checked as we are already logged in
- click to create a new repository



- create a component Room.js besides App.js

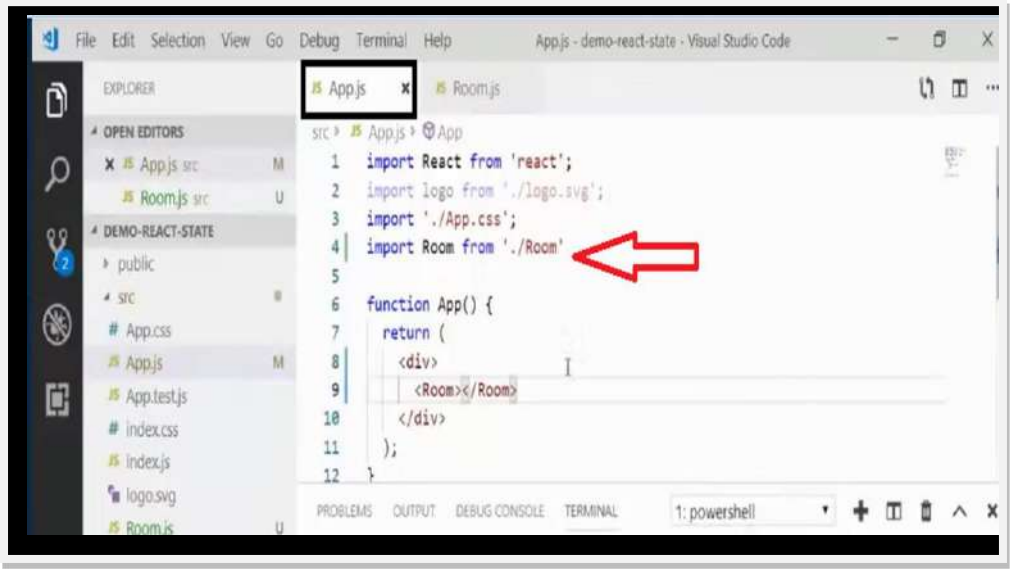
- enter following code:

```
Import React from 'react';

import './App.css';

function Room() {

return (

  <div>

        This is Room component

  </div>

  );

 }

export default Room;
```

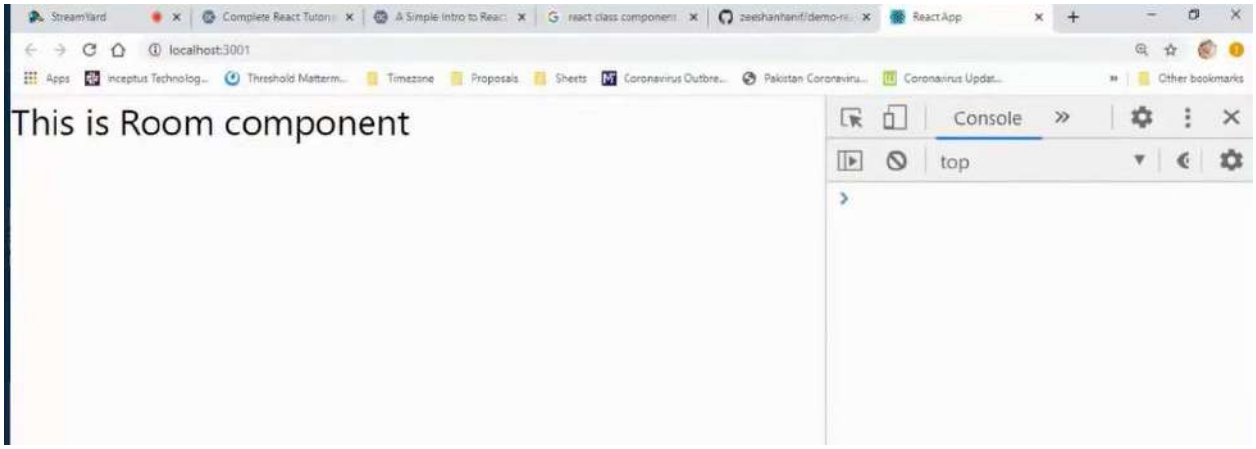


- go to App.js and import Room from './Room; as:

```
Import React from 'react';

import './App.css';

import Room from './Room';

function App() {

return (

  <div>

   <Room></Room>

  </div>

  );

 }

export default App;
```



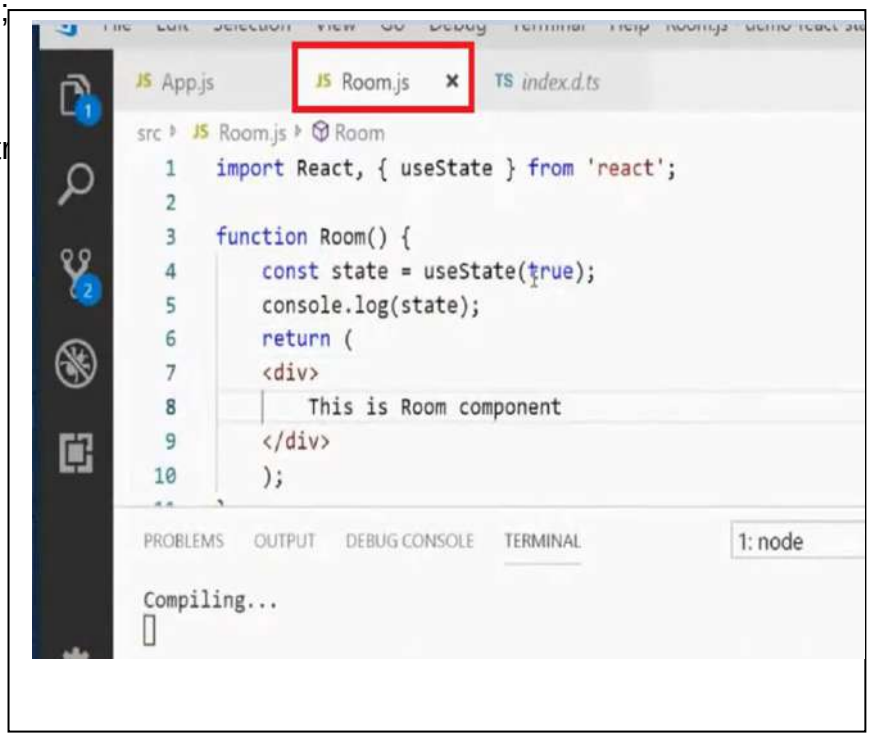
- go to terminal and enter command; yarn start
- this will start developing the server and to run it on other port as shown

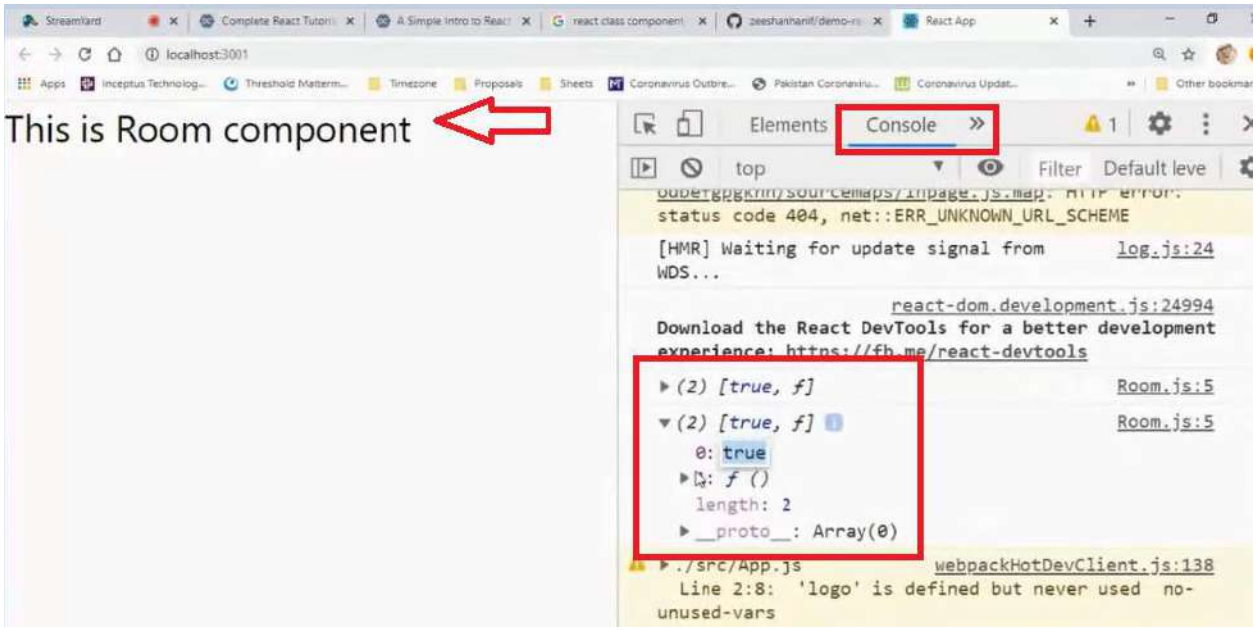


Import useState Inside React

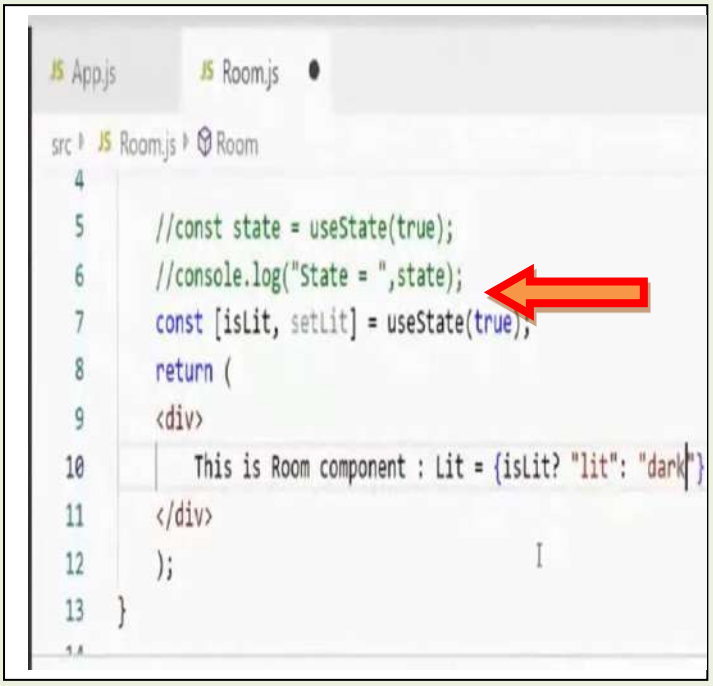
Go to Room.js and follow up the following code

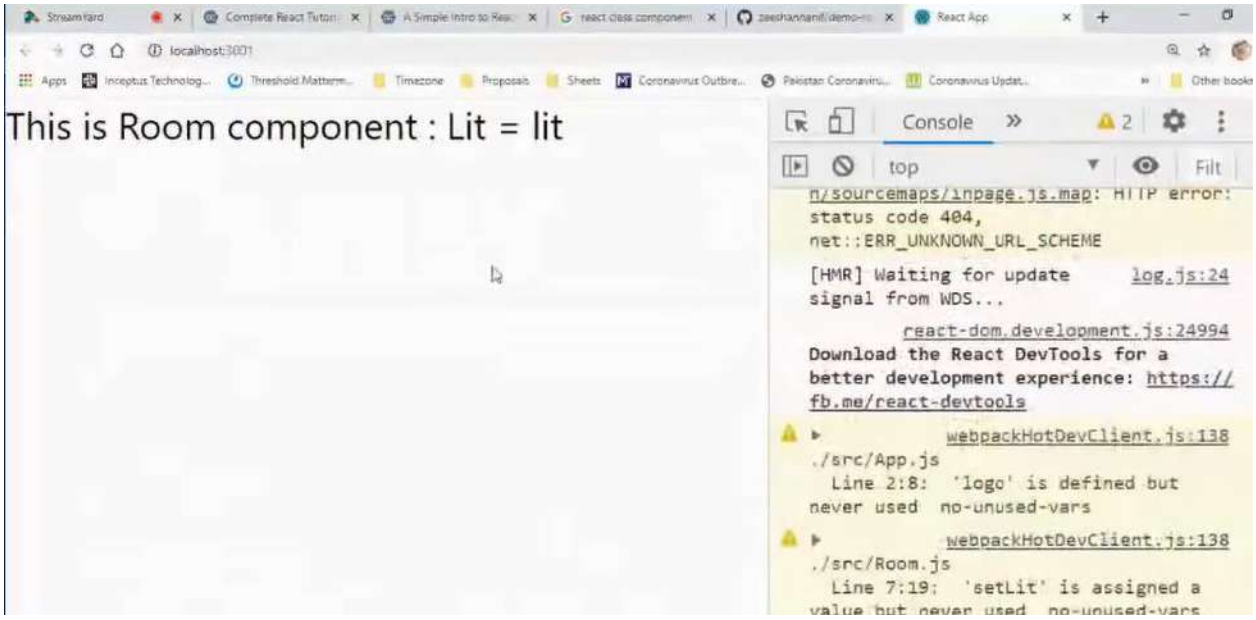
```
import React, { useState } from'react';

function Room() {

        const [islit, setLit] = useState(tr

        return

        <div>

        this is Room Component

        </div>

        );

}

export default Room
```





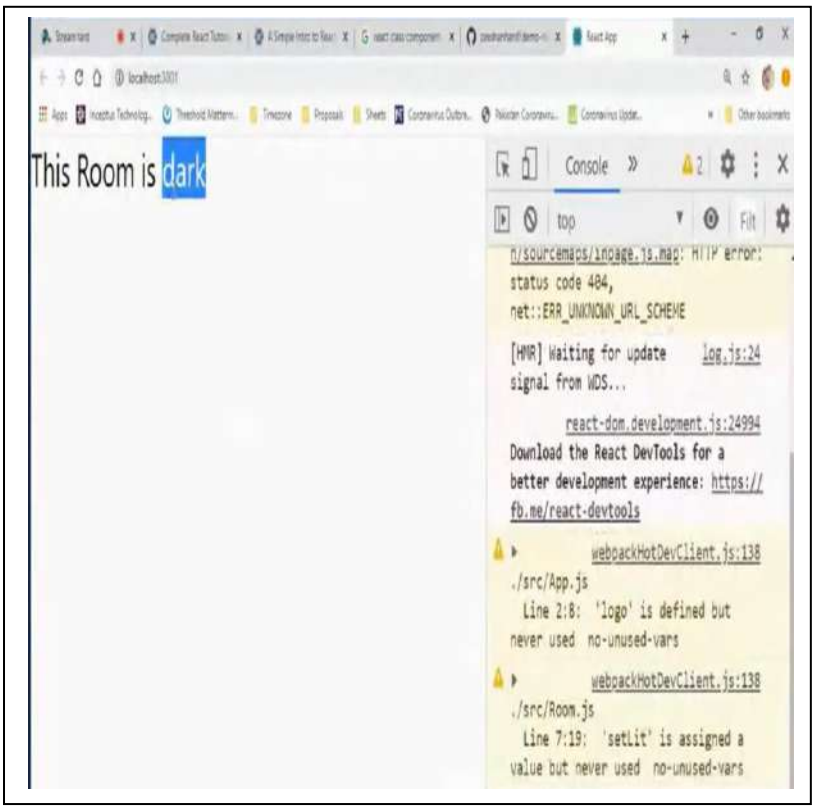

```
import React, {useState} from'react';

function Room(){

const [islit,setLit] = useState(true);

return

<div>

this is Room Comp :Lit ={isLit? “lit”: “dark”}

/div>

);

}

export default Room
```





❖ When you make the initial statement as false it turns out “room is dark”

```
import React, {useState} from'react';

function Room(){

const [islit,setLit] = useState(false);

return

<div>

this room is:Lit ={isLit? “lit”: “dark”}

/div>

);

}

export default Room
```



6. Change The State When You Click The Button

Let's add a button and kick this thing into high gear. Change the div to look like this:

```
<div className="room">
  the room is {isLit ? "lit" : "dark"}
  <br />
  <button onClick={() => setLit(!isLit)}>
    flip
  </button>
</div>
```

So we've got a plain old line break with the <br/>, and then a button that says "flip". When you click the button, it will call its onClick prop.

Remember how we talked about arrow functions earlier? This is one of those. It's embedded right there, anonymously, and passed straight into onClick. That anonymous function calls setLit with a new value.

FYI!! Be careful when you pass functions to props that you don't inadvertently call the function at render time:

```
  // Wrapping with an arrow function
  // delays execution until the click
  // ✅
  <button onClick={() => setLit(!isLit)}>
    flip
  </button>

  // Unwrapped call to setLit will happen
  // before the button is even rendered!
  // ❌
  <button onClick={setLit(!isLit)}>
    flip
  </button>
```

Click the "flip" button now. Does it work? Hooray! We'll fix the stark white background in a sec, but let's talk about this setLit function.

How setLit works

In the onClick function, we're toggling the isLit state true/false depending on what it's currently set to. You might wonder why we don't just write isLit = !isLit. It's because the setLit function actually has 2 jobs:

- first, it changes the state
- then it re-renders the component

If you just change the variable directly, React has no way of knowing that it changed, and it won't re-render. Remember that isLit is a regular old variable – not a special React thing! It will go out of scope at the end of the function and any changes to it would be lost.

So that's why it's important to call the setter so that React can update the value of that hook's state behind the scenes.

6.2. Practical: Change The State When You Click The Button

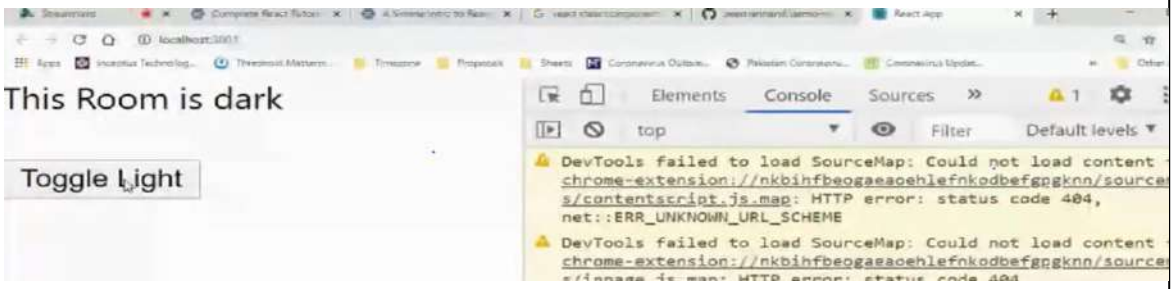
```
import React, {useState} from'react';

function Room(){

        const [islit,setLit] = useState(true);

        function updateLit(){

        console.log("Button clicked");

        setLit(!isLit);

        return (

        <div>

        this room is:Lit ={isLit? "lit": "dark"}

        <br/>

        button on Click={updateLit} >Toggle light
        </button>

        /div>

        );

}
```

src > Room.js > Room > updateLit

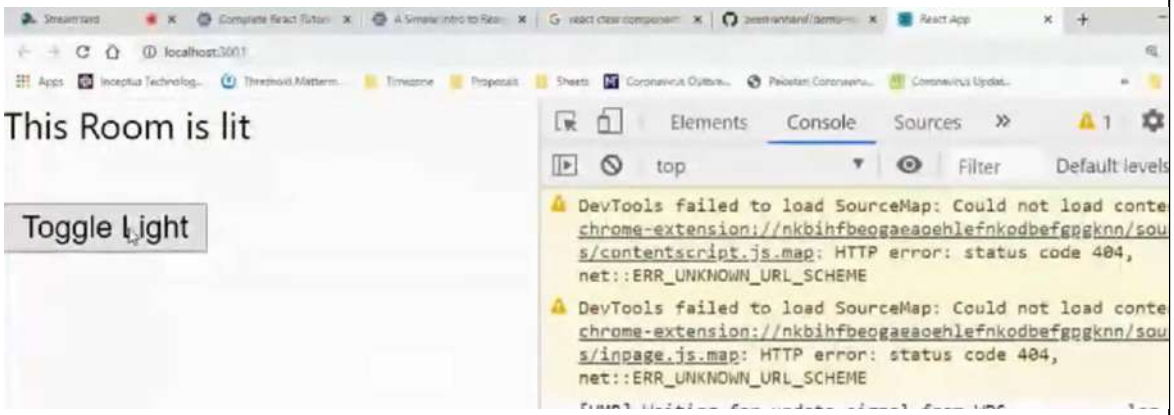
```
6       //console.log("State = ",state);
7       let [isLit, setLit] = useState(false);
8       function updateLit() {
9           console.log("Button clicked");
10          setLit(!isLit);
11          //isLit = !isLit;
12
13      }
14
15      return (
16      <div>
17          This Room is {isLit? "lit": "dark"}
```

```
        <br/>
        Age : {age}
        <br/>
        <button onClick={updateLit} >Toggle Light</button>
    </div>
    );
}
```

- **At first click to Toggle light**



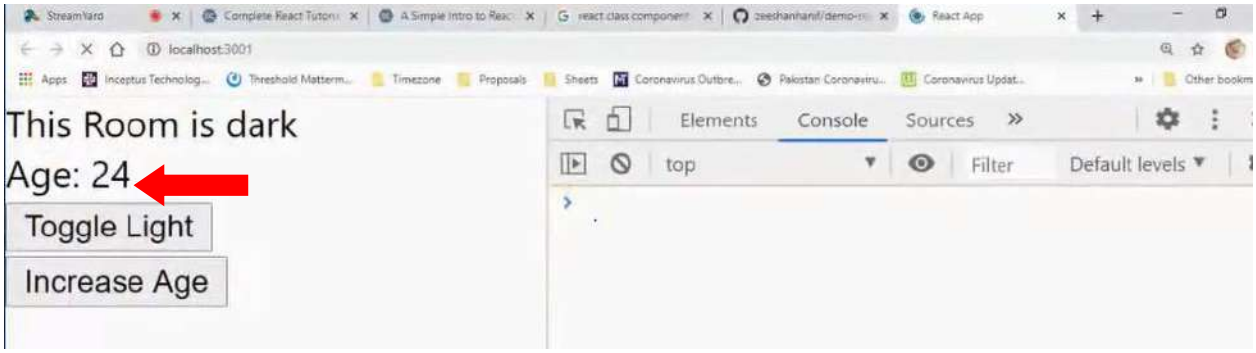
- **At 2nd click to toggle light**

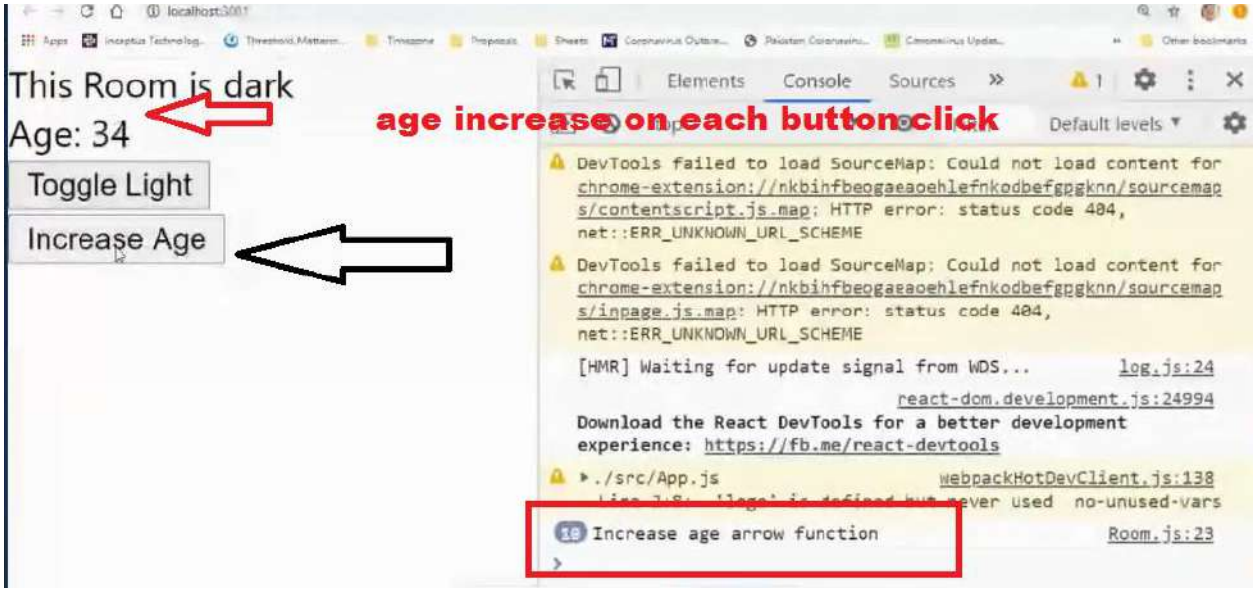


7. Provide Function As An Arrow In The Event

❖ **To use 2 Button at a time follow up the following code in Room.js**

```
import React, { useState } from'react';

function Room() {

        const [islit,setLit] = useState(true);

        let [age , setAge] = useState(24);

        function updateLit(){

        console.log("Button clicked");

        setLit(!isLit);

        function IncreaseAge() {

        console.log(" increaseAge Bottun clicked")

        setAge(++age);

        return (

        <div>

        this room is:Lit ={isLit? “lit”: “dark”}

        <br/>

        Age :{age}

        <br/>

        <button onClick={updateLit} >Toggle light </button>

        <br/>

        console.log(" increaseAge Bottun clicked")

        setAge(++age);

        } }

        Increase Age</Button>

        /div>

        );

}
```



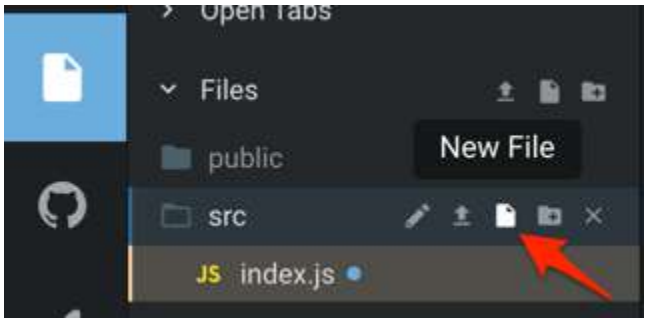


8. Change The Background Color

To do that we're going to use some CSS.

Not CSS-in-JS, not some special React CSS (not a thing), just plain old CSS.

In the CodeSandbox editor, hover over the "src" folder on the left and click the "New File" button.



Name it index.css, and paste this in. We don't need TOO much style here, but we do need some.

```
html, body, #root, .room {
  height: 100%;
  margin: 0;
}
.lit {
  background-color: white;
```

```
  color: black;
}
.dark {
  background-color: black;
  color: white;
}
```

At the top, we are setting the height of absolutely-freaking-everything to 100%, so that the whole page goes dark instead of just the top 20 pixels. Then we have the class selectors .lit and .dark, which will set the background and text color when those CSS classes are applied.Now click over to index.js, and at the top of the file, import the CSS file like so:

```
import "./index.css";
```

(We can do this because of Webpack: it will see this import and take it to mean “index.js depends on index.css” and will add the CSS to the page in the correct spot)

Now all that’s left is to apply the “lit” and “dark” classes to the Room component. Remember how the in Room has the prop className="room"? We need to change that dynamically to either room lit or room dark, depending on the state.At the top of our component, after state is created, we’ll create a variable for the lightness:

```
const brightness = isLit ? "lit" : "dark";
```

Then use that variable, plus ES6’s template strings, to change the className. Like this:

```
<div className={`room ${brightness}`}>
```

The backticks signify a template string in ES6.

Template strings allow you to insert variables within them, which is what the ${brightness} is doing.And finally, the whole string is surrounded by single braces because it’s an expression and because React says so. I’m not sure why the React allows you to just pass someProp="string" without the braces but requires the braces when it’s someProp={`template string`}, but it does, so don’t forget them.Click the button now, and you’ll see that the background color changes along with the text. Woohoo! □If your code isn’t working, you can compare it to the final working example here.

And one more thing: we could’ve written the className like this, all inline, without the extra variable:

```
<div className={`room ${isLit ? "lit" : "dark"}`}>
```

That would work the same (try it out!). I created the variable just to make it easier to read

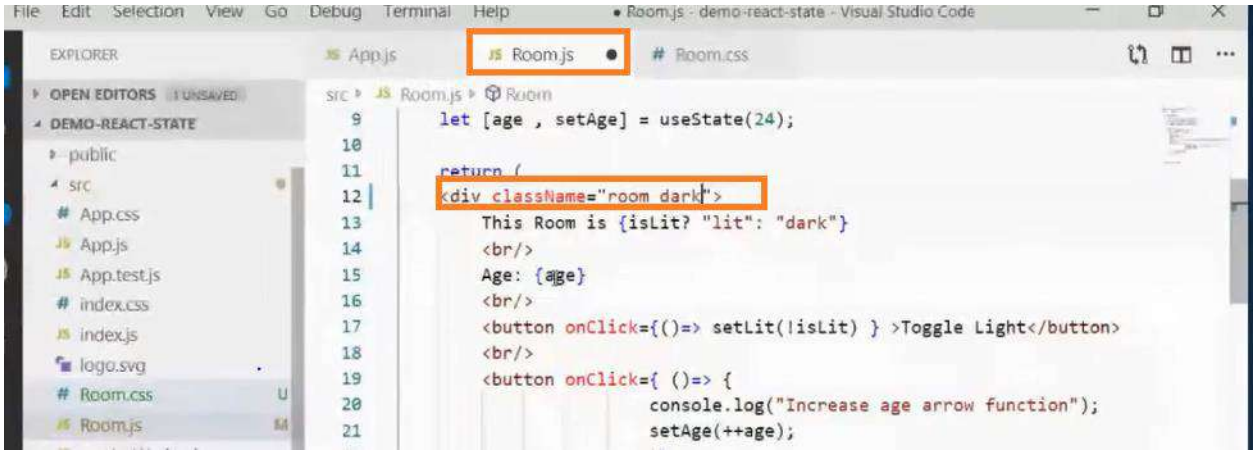
8.2 CSS Style Implementation

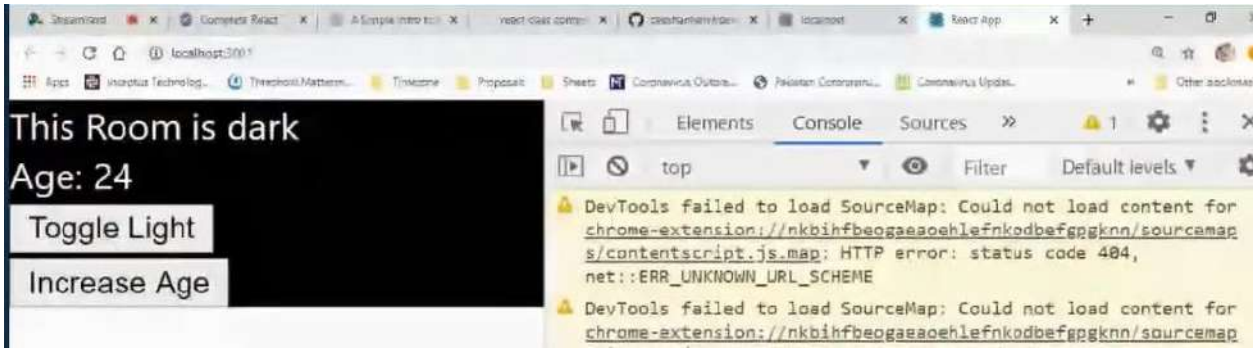
Create another component Room.css and follow the following code in order to provide color:

```
.room{
height:100%;
}
.lit {
    background-color:white:
    color: black;
}
.dark {
background-color:black;
```

- **Now in order to implement CSS style or background colours just go to the Room.js file and provide div with classes as follows for three of these classes: let all other codes remain as same**

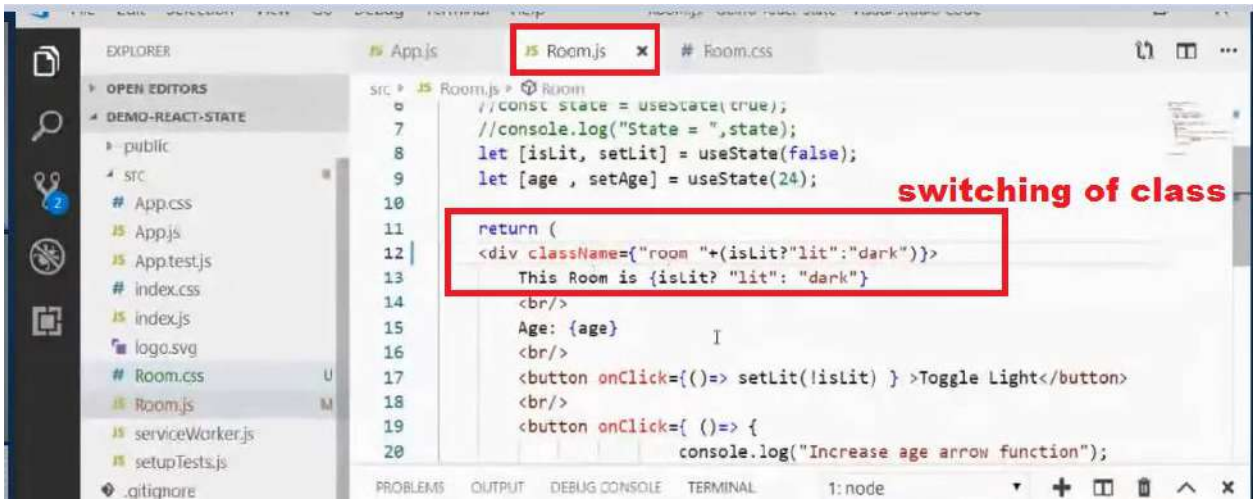
<div className=”room dark”>



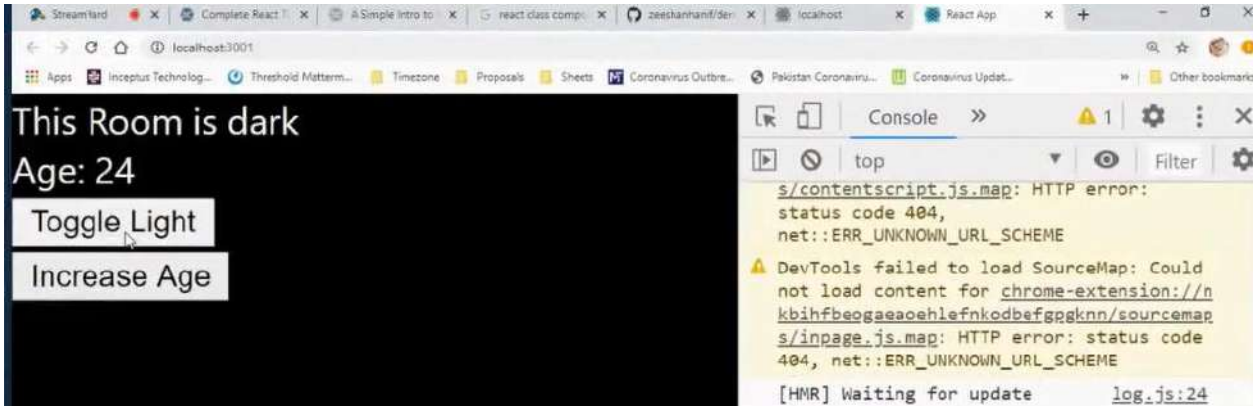


- Now if you want to switch these class as light background on the lit property as true and black background when the lit property is false; then in the div section follow up these coding for Room.js file .let the remaining code be as same

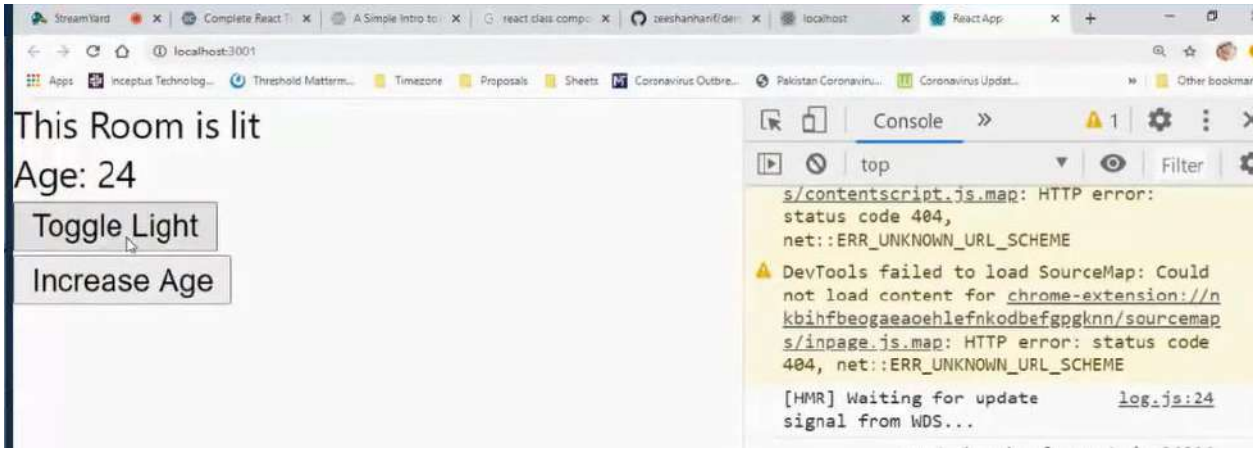
<div className={“room “+(isLit? “lit”: “dark”)}



- When the lit property is true; this appears as onclick to Toggle Light



- When the lit property is false: it appears as onlick toggle Light



THE END

Session 5: React's Context API and UseReducer React Hook

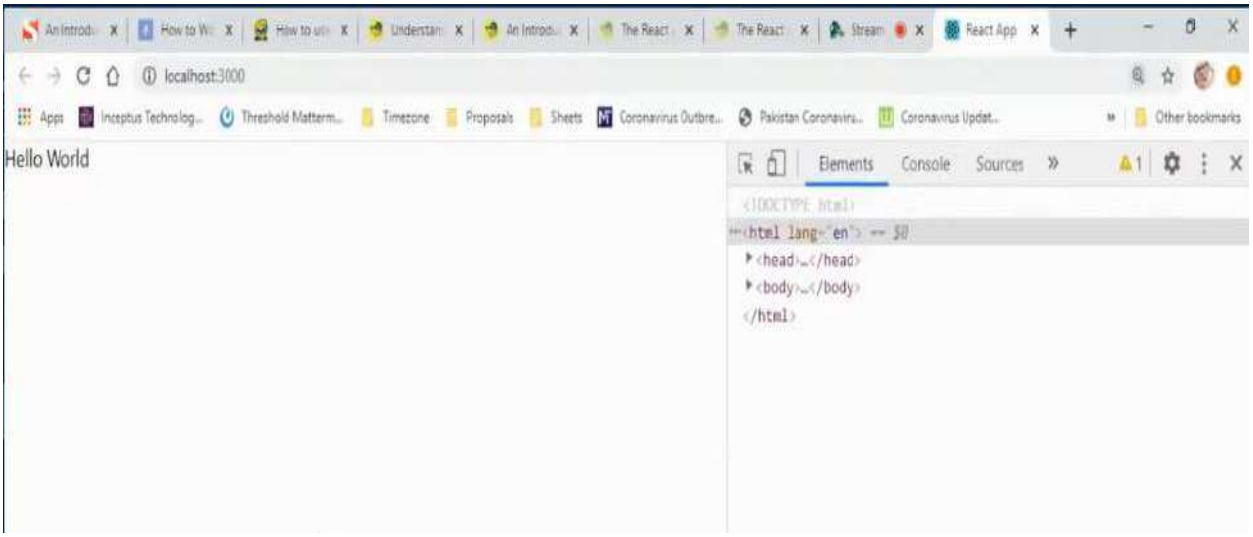
1. Transfer of Data in a Tree Having Single property

Let's look at how to share data from one component to another component

- For that first create a simple hello world in App.js and open up in the browser in its display form as

```
import React from 'react';
import './App.css';
function App () {
return(
<div>
   Hello World
  </div>
);
export default App;
```

```
App.js
src > App.js > App
1   import React from 'react';
2   import logo from './logo.svg';
3   import './App.css';
4
5   function App() {
6     return (
7       <div >
8         Hello World
9       </div>
10    );
11  }
12
13  export default App;
14
```



- now let's create 2 component
  - parent.jss
  - child .jss

- **for Parents Class**

```
import ./App.css';

function Parent(props) {

return (

<div>

   Parent

  </div>

);

export default Parent;
```

```
App.js   Parent.js   Child.js   index.js
src > Parent.js > Parent
1   import React from 'react';
2
3   function Parent(props) {
4     return (
5       <div >
6         Parent
7       </div>
8     );
9   }
10
11  export default Parent;
```

- **For child class**

```
import ./App.css':

function child() {

return(

<div>

   child

  </div>

);

export default child;
```

```
App.js   Parent.js   Child.js   inde
src > Child.js > ...
3   function Child(props) {
4     return (
5       <div >
6         Child
7       </div>
8     );
9   }
10
11  export default Child;
12
```

- Go to App.js file and provide properties and value as **<u>let [number, setNumber] = useState(45);</u>**
- Pass this property to parent component as **<u><parent num={number}></parent></u>** and parent component has child component that will display the value over screen as shown

App.js

```
import React from 'react';
import ./App.css';
function App () {
return(
        <div>
           Hello World
        <Parent
        num={number}></Parent>

          </div>
);
export default App:
```

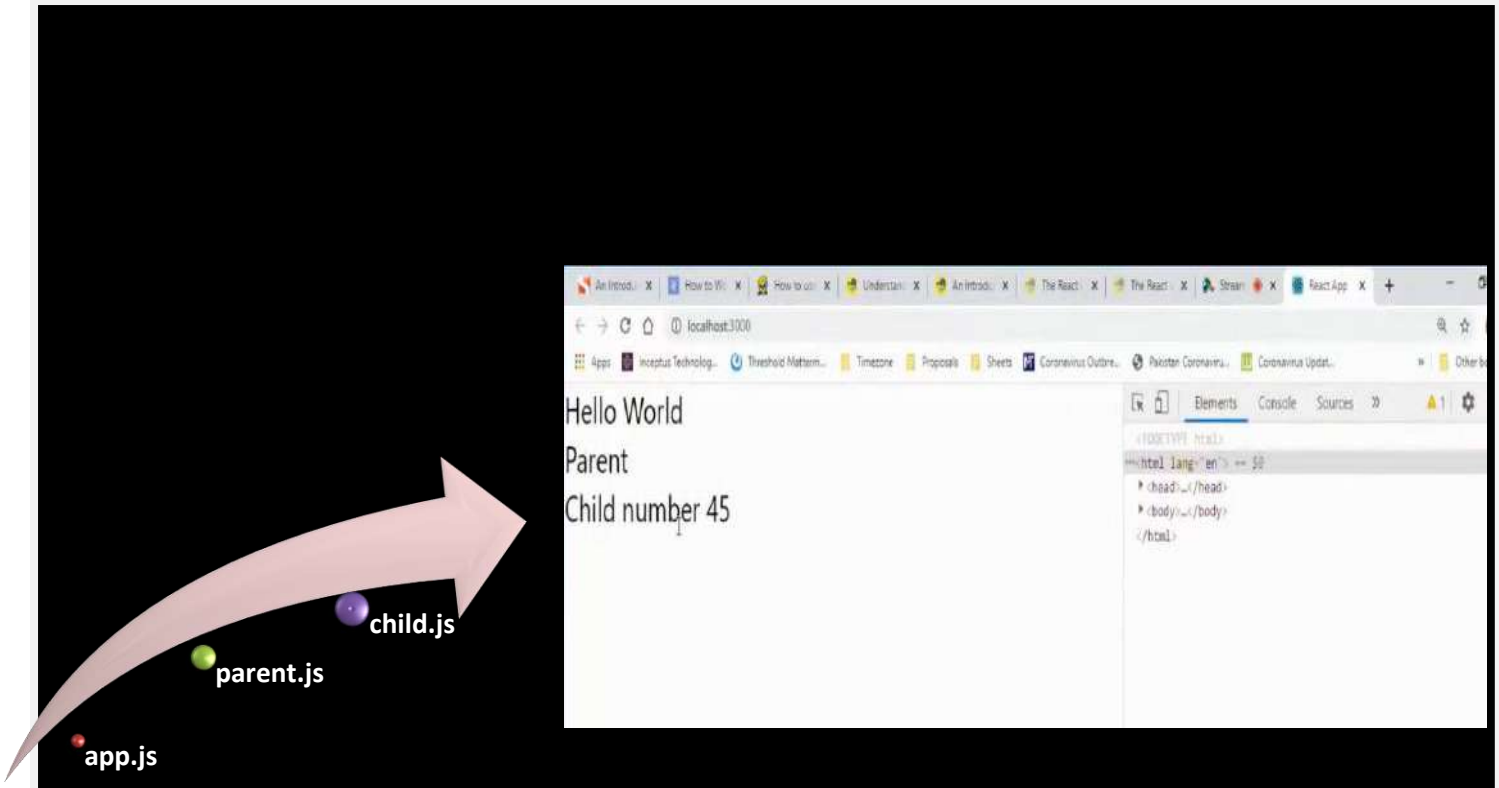
```
src > App.js
5
6   function App() {
7     let [number, setNumber] = useState(45);
8     return (
9       <div >
10        Hello World
11        <Parent num={number}></Parent>
12      </div>
```

Parent.js

```
import React from 'react':
import  child from './child
function Parent (props){
  return(
          <div>
              Parent
          <child
          num={props.num}></child>
            </div>
);
}
export default App:
```

child.js

```
import React from 'react':
function child (props){
 return(
   <div>
       child number {props.num}
  </div>
  );

}

Export default child;
```



app components contain parent component, parent components contain child components and child component display the value of props.num

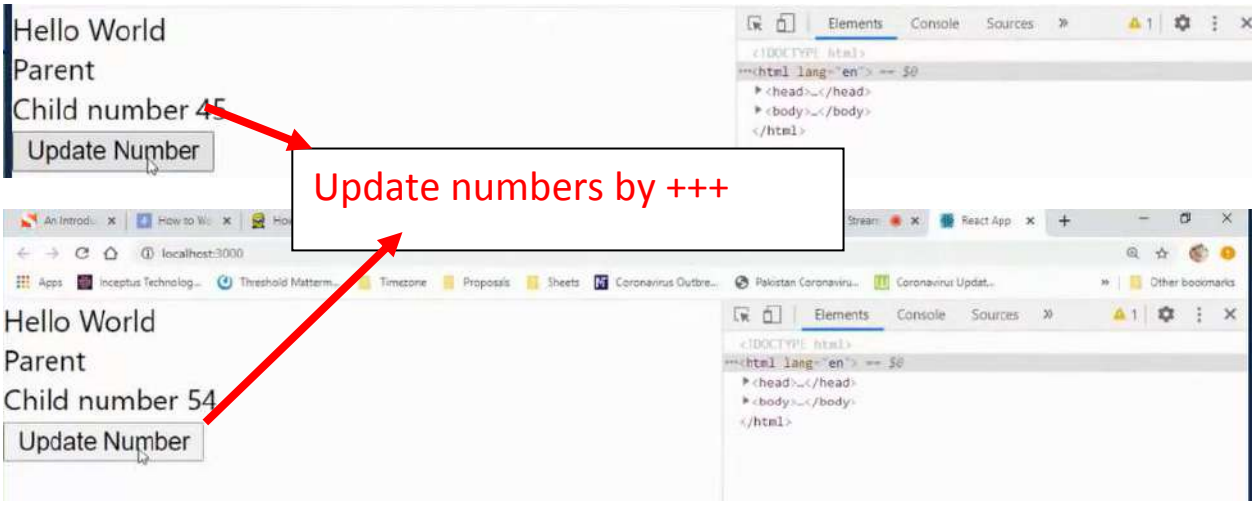
OnClick Button properties

- Now for onclick property go to app.js and insert the button tags with following properties and value **as;**

<button onClick={()=>{ setNumber(++number) }}>updateNumber</button>

```
import React from 'react';
import './App.css';
function App (){
return(
<div>
   Hello World
<Parent num={number}></Parent>
<button onClick={()=>{
setNumber(++number)
}}>updateNumber</button>


  </div>
);
export default App;
```

- **Now every time you click on update number it will its value by +++**



  - Since in a tree a change results to deploy changes in all of the components in a sequence to avoid this problem we context

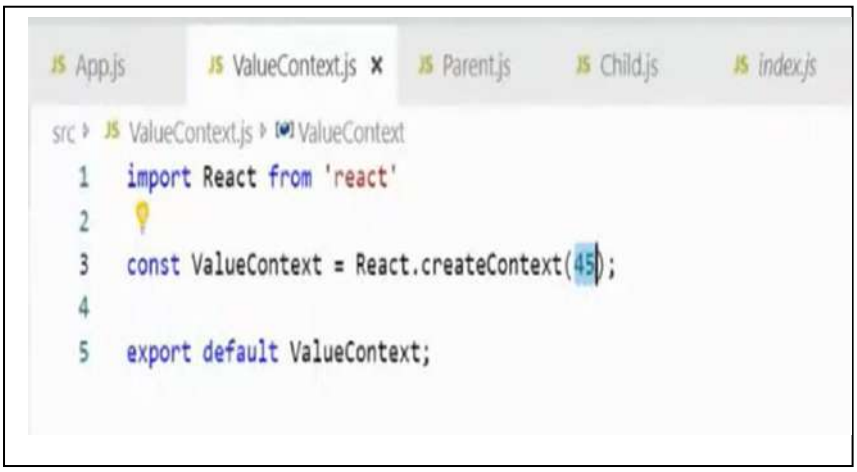
2. **<u>Context.js component</u>**

<u>Context provides facility to switch the properties, a component owe the context; that will have value, all the component within this context have values, whenever you need to call the value you can directly call it no need of deployment in each single components;</u>

- so create a new component named as valuecontext.js with the following code value as:

valueContext.js

```
import React from'react';

const ValueContext =React.createCont

export default ValueContext;
```



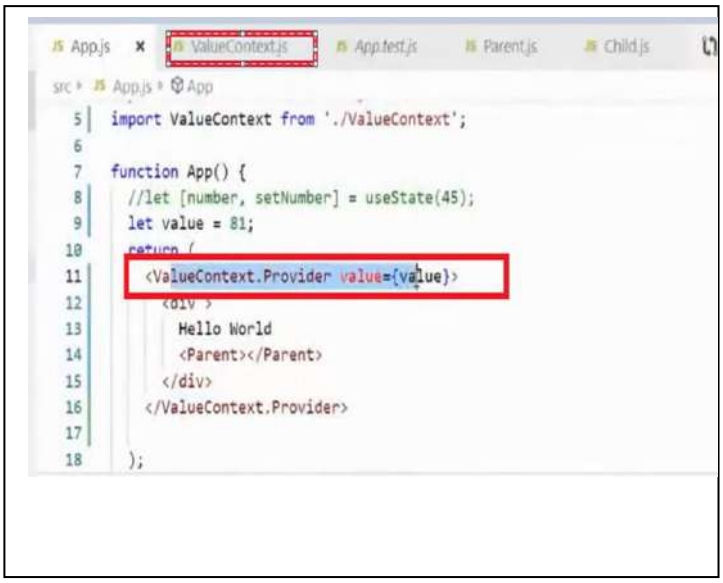
context has 2 things

- **provider** (provides value to the component tree
- **consumer**(consumes the value)

- in app.js create the value of provider as:

<valueContext.provider value={value}>

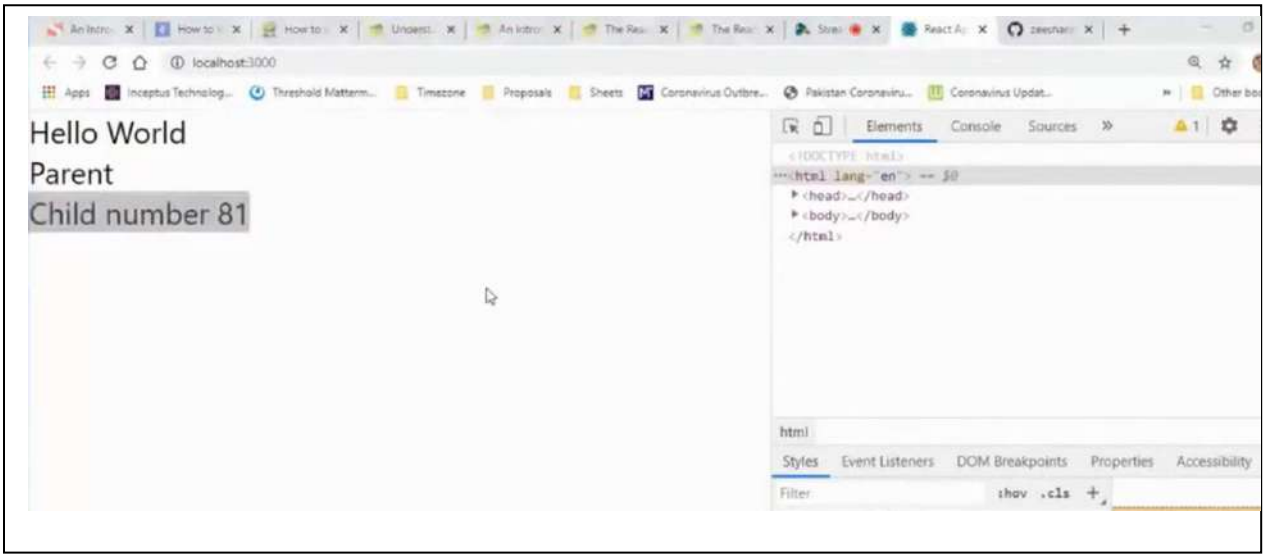
```
import React from 'react';
import valueContext from './VlaueContext';
function App (){
   let value =81;
  return(
   <valueContext.provider value={value}>

            <div>
                Hello World
            <Parent ></Parent>

              </div>
         </valueContext.provider>
);
export default App;
```



- In child.js we have to call the value **<u>let value =useContext(valueContext);</u>**

```
import React  {useContext}  from 'react';
import valueContext from ‘./VlaueContext’;
function child(props){
    let value =useContext(valueContext);
 return(
   <div>
       Child number{value}
  </div>
  );
}
Export default child;
```

```
1   import React, { useContext } from 'react';
2   import ValueContext from './ValueContext';
3
4   function Child(props) {
5     let value = useContext(ValueContext);
6     return (
7       <div >
8         Child number {val
9       </div>
10    );
11  }
12
13  export default Child;
```

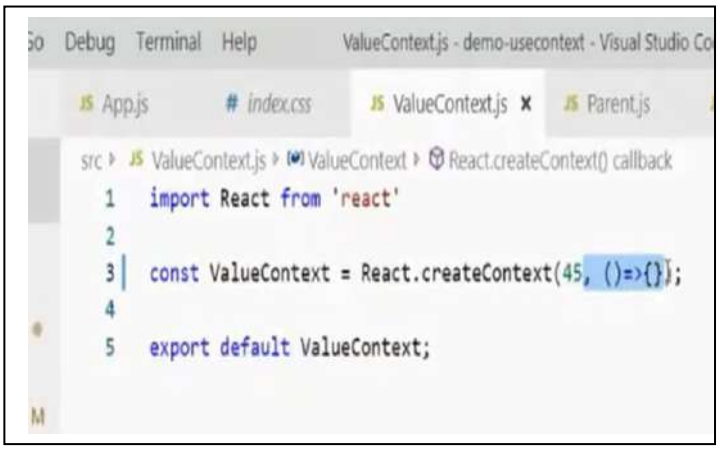


Update the value

valueContext.js

- in CreateContext we have created 2nd element the function that is needed to be updated value this is a default value we are not going to use
- so follow up following code;

```
import React from'react'

const
valueContext=React.createContext(45),()
=>{});

export default ValueContext;
```



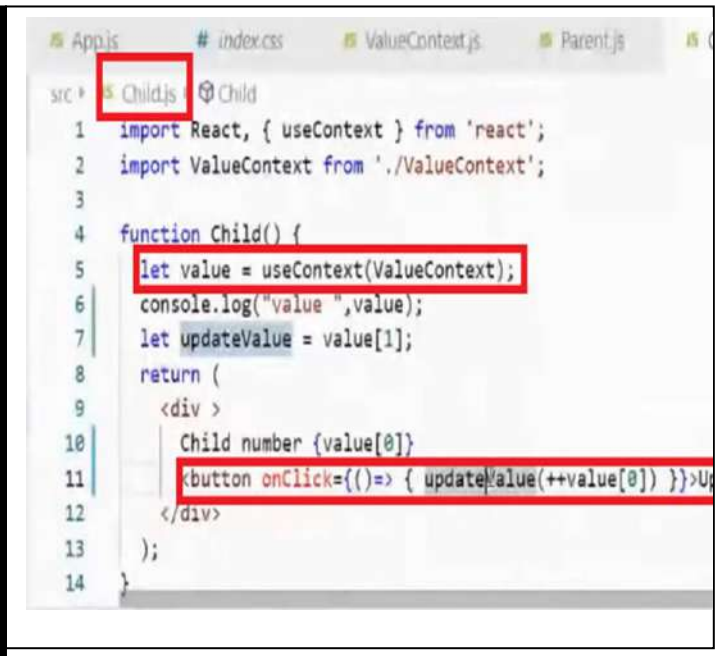
App.js in provider usestates are created **let value =useState(48)** it returns array;whatever the array that is being used by the context we will pass it to the context so that all the child or subcontents of tree can have access to this state

```
import React from 'react';
import valueContext from ‘./VlaueContext’;
function App (){
   let value =useState(48)
  return(
   <valueContext.provider value={value}>

          <div>
              Hello World
          <Parent ></Parent>

            </div>
        </valueContext.provider>
);
export default App;
```

For child.js

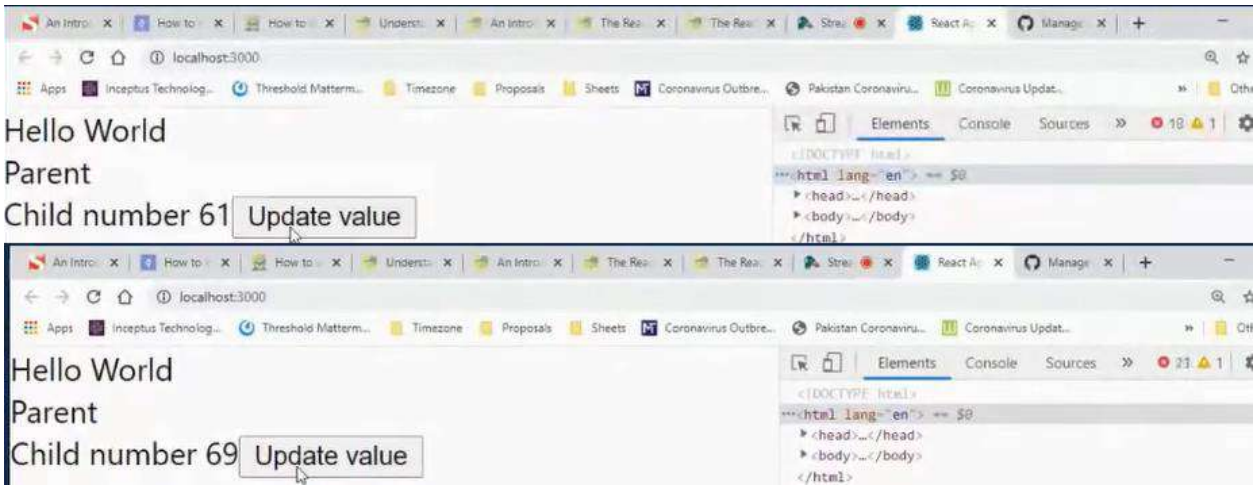
- we use the useContext and call out the valueContext.this is the function of react .it will return an array. this array will have two values
  - for index 1 to update the value
  - 2nd index will include the current value that will be incremented.
- This is how we can Access the value anywhere in the child trees.
- So on every onClick, the value will be updated by calling update value

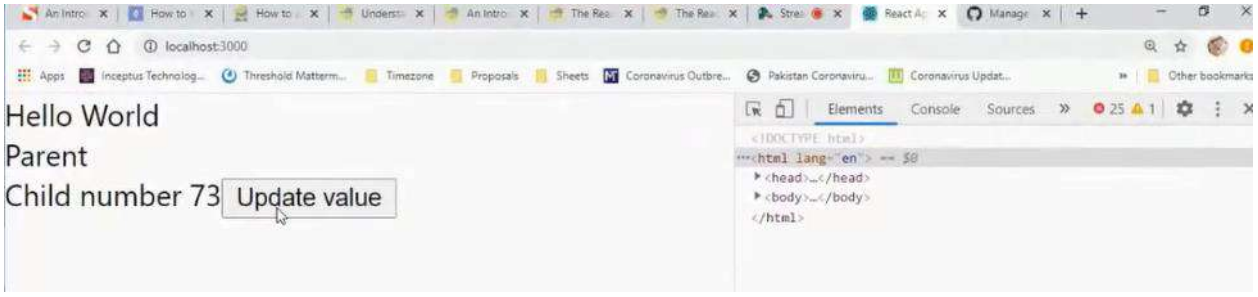
```
import React  {useContext}  from 'react';
import valueContext from ‘./VlaueContext’;
function child(){
    let value =useContext(valueContext);
  console. Log(“value “, value)
   let updateValue =value [1];
 return(
   <div>
       Child number{value[0]}
<button
onClick={()=>{updateValue(++value(0)}}>updatevalu
e</button>
  </div>
  );

}

Export default child;
```



NOTE : Parent.js would remain the same

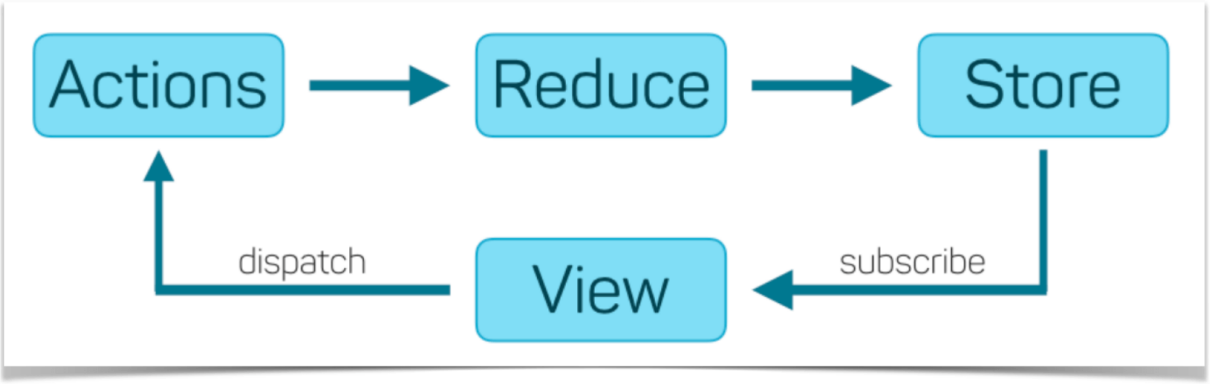
❖ **So the value will be updated on every onClick**





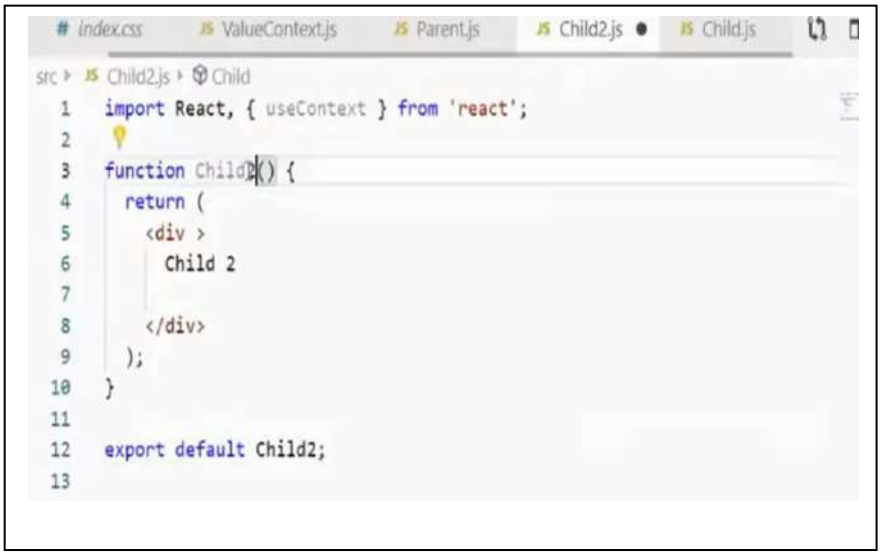
3. **<u>Reducer</u>**

The **reducer** is a pure function that takes the previous state and an action and returns the next state. (previous state, action) => nextState. It's called a **reducer** because it's the type of function you would pass to Array.



❖ **create another child2.js as:**

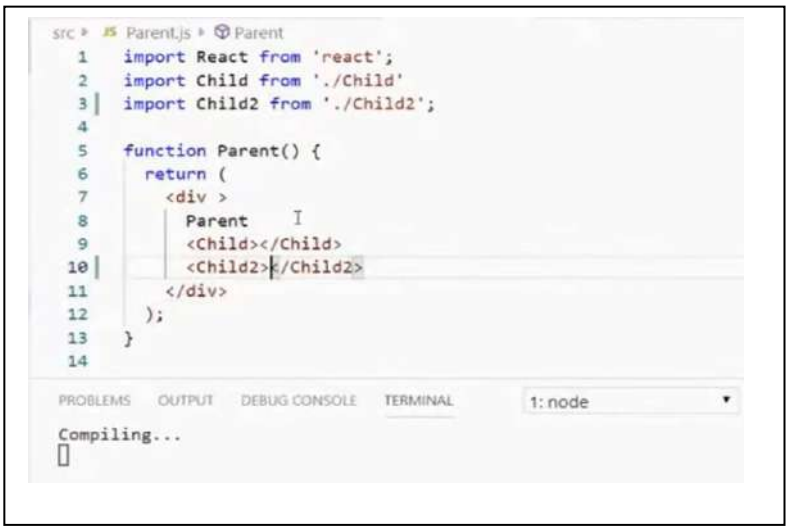
```
import React, {useContext} from 'rea

function child 2(){

return(

        <div>

          child 2

        </div>

   );

}

Export default child2;
```

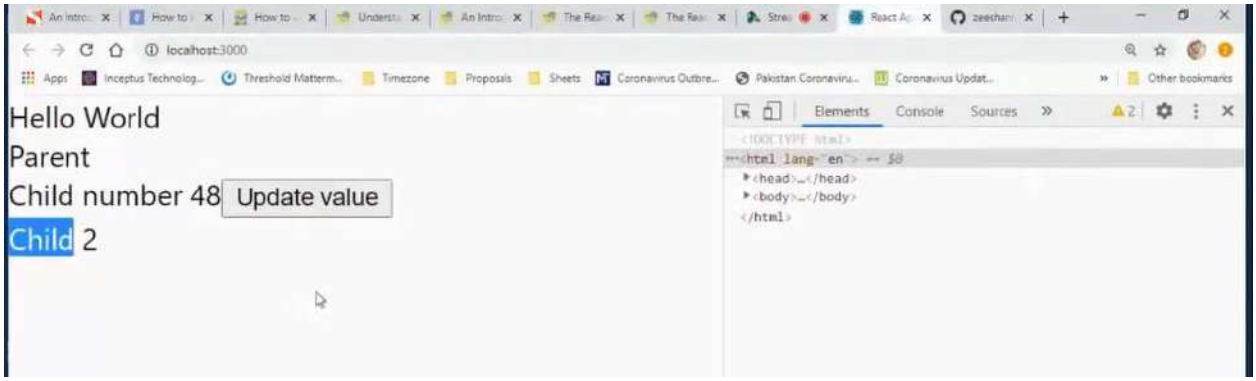


➢ Declare child2 in the parent class as;

for parent.js

```
import React from 'react':
import  child from './child';
function Parent (){
  return(
        <div>
            Parent
        <child ></child>
        <child2></child2>

          </div>
  );
}
export default App:
```



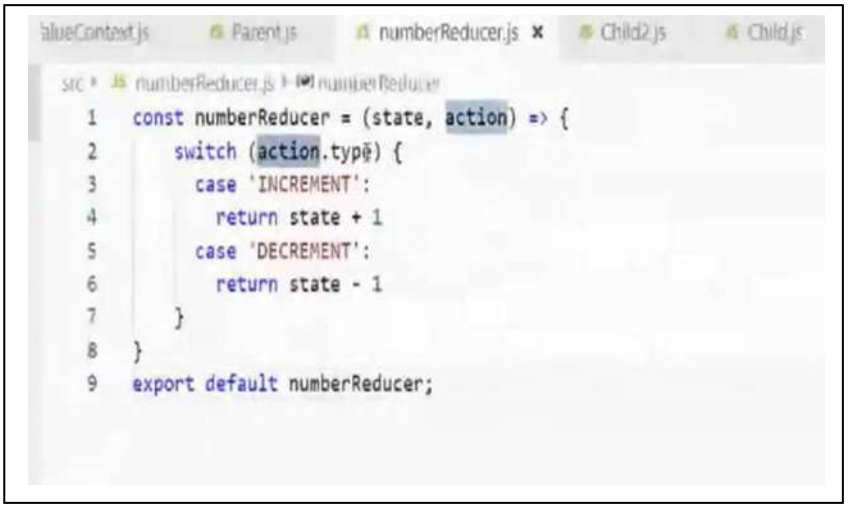


- We use reducers to have multiple applications instead of just updating values.
- Management state of the whole application. it is a pure function it doesn't update the previous value but it creates a new every time you call it

- so create a new component

numberReducer.js

```
const numberReducer = (state,
action) => {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1

  }
}
Export default numberReducer;
```



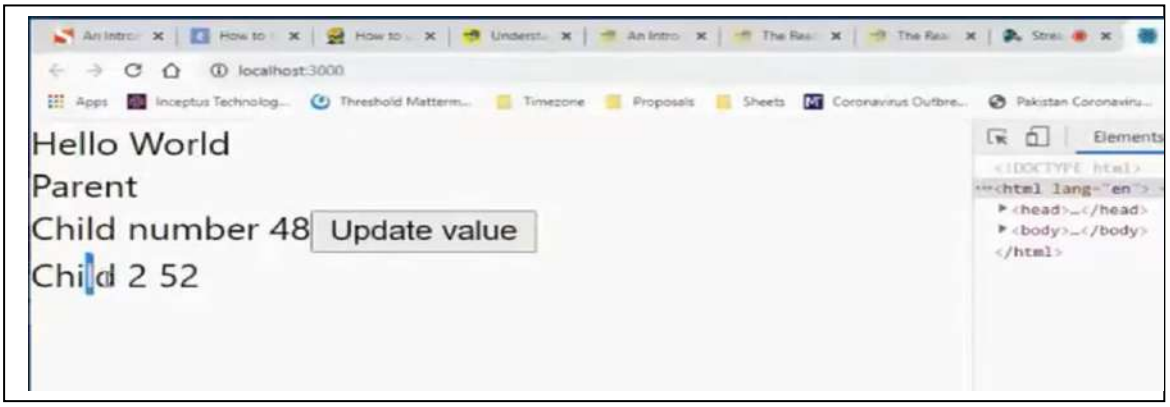
- In child2 provide state and dispatch (any number) and assign any value as **let [state, dispatch]=useReducer,numberReducer52**

- import numberReducer from ‘./numberReducer’;
- import React,{ useContext,useReducer} from 'react':
- open up the browser , refresh page, and child2 with current value has been displayed over the screen

`

Child2.js

```
import React,{ useContext,useReducer} from
'react':
import  numberReducer from
‘./numberReducer’;
function child 2 (){
  let [state, dispatch] =
useReducer,numberReducer52
  );
Return(
   <div>
     Child 2{number}
</div>
);
}
export default child2;
```



Let's update the value by using onClick action

- Choose 2 button one for incrementing and other decrementing the value in child2.js file

  - <button onClick={()=>{dispatch ({type:INCREMENT"}); }}>increment</button>
  - <button onClick={()=>{dispatch ({type:DECREMENT"}); }}>decrement</button>

```
import React,{ useContext,useReducer} from 'react';
import  numberReducer from './numberReducer';
function child 2 (){
  let [state, dispatch]=useReducer,numberReducer52
  );
Return(
   <div>
     Child 2{state}

  <button onClick={()=>{dispatch
({type:INCREMENT"});  }}>increment</button>

  <button onClick={()=>{dispatch
({type:DECREMENT"});  }}>decrement</button>


</div>
);
}
export default child2;
```

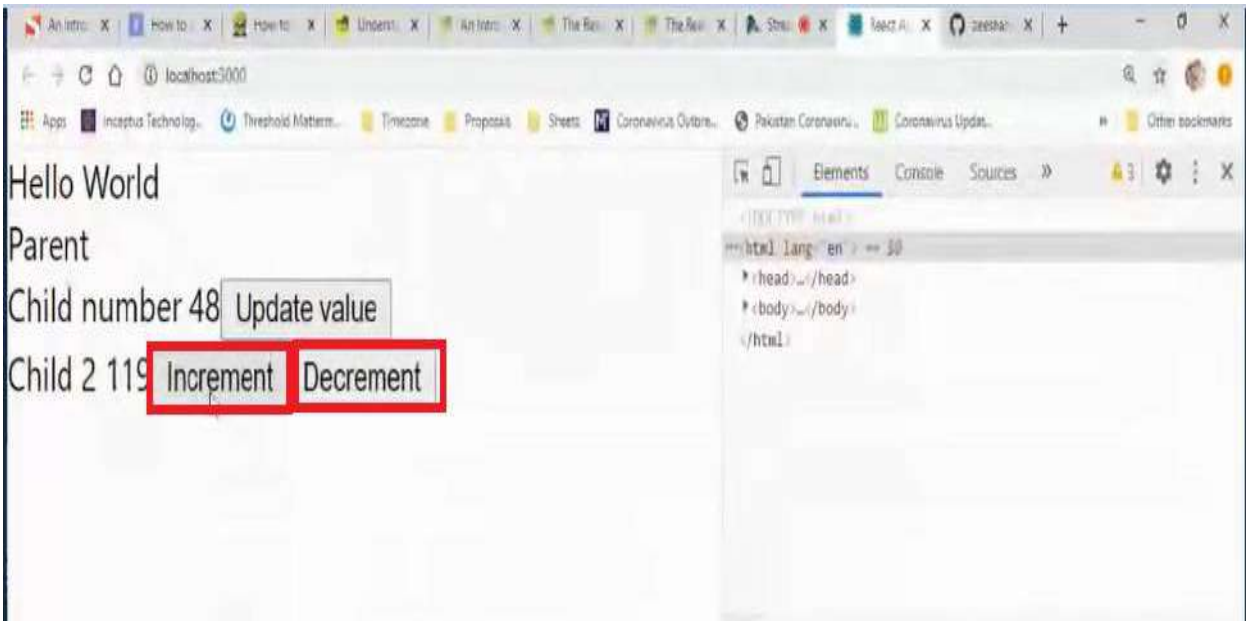
```
import React, { useContext, useReducer } from 'react';
import numberReducer from './numberReducer';

function Child2() {
  let [state, dispatch ] = useReducer(numberReducer,52);
  return (
    <div >
      Child 2 {state}
      <button onClick={()=> { dispatch({type:"INCREMENT"});  }}>Increment
      <button onClick={()=> { dispatch({type:"DECREMENT"});  }}>Decrement
    </div>
  );
}
```

NumberReducer.js

```
const numberReducer = (state, action) =>
{
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1


  }
}
Export default numberReducer;
```

```
const numberReducer = (state, action) => {
    switch (action.type) {
      case 'INCREMENT':
        return state + 1
      case 'DECREMENT':
        return state - 1
    }
}
export default numberReducer;
```

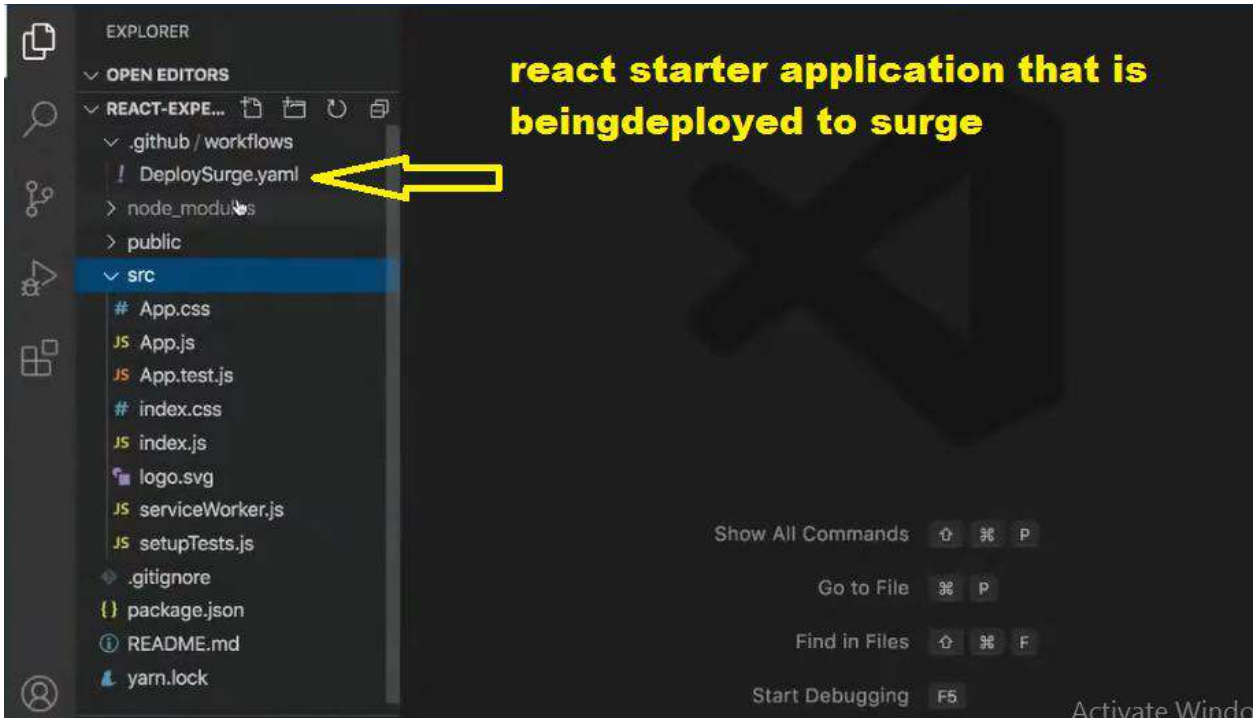
- ✓ In this way, you can either increase or decrease the value just by click to increment or decrement button

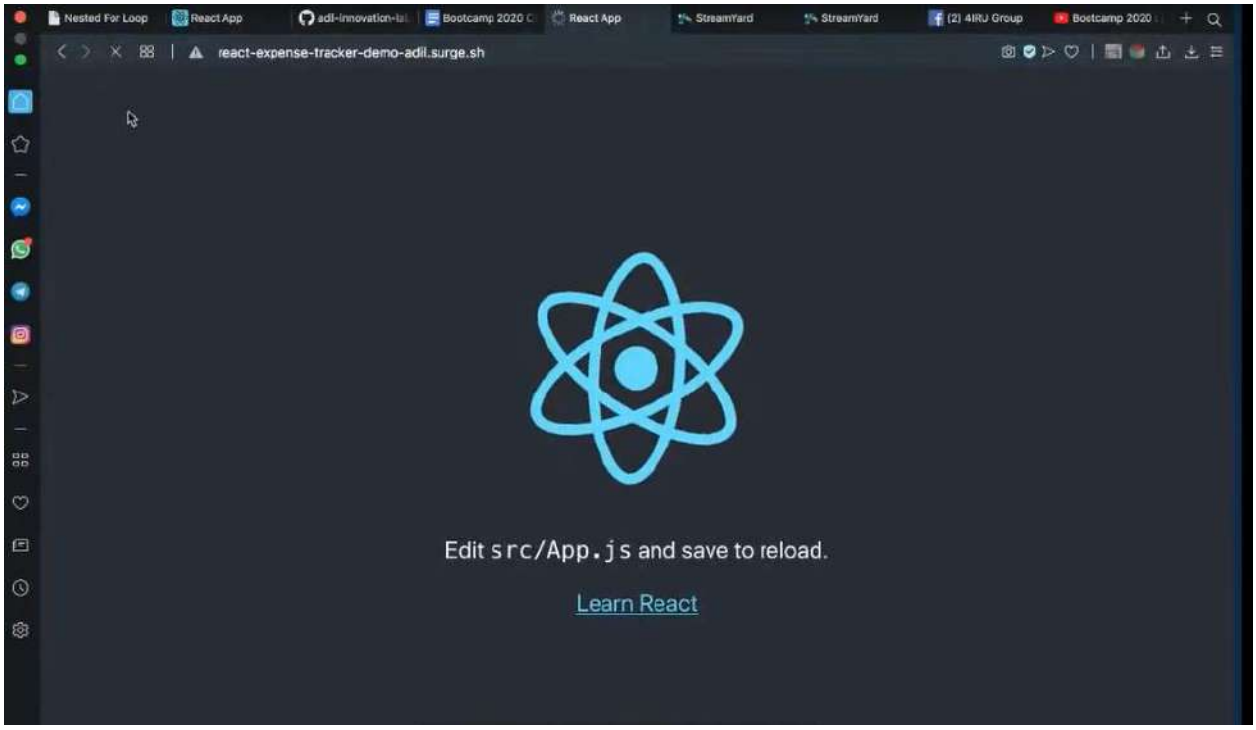


Session 6: Expense_Tracker_App

1. **<u>React Expense App; how To Build That Application</u>**

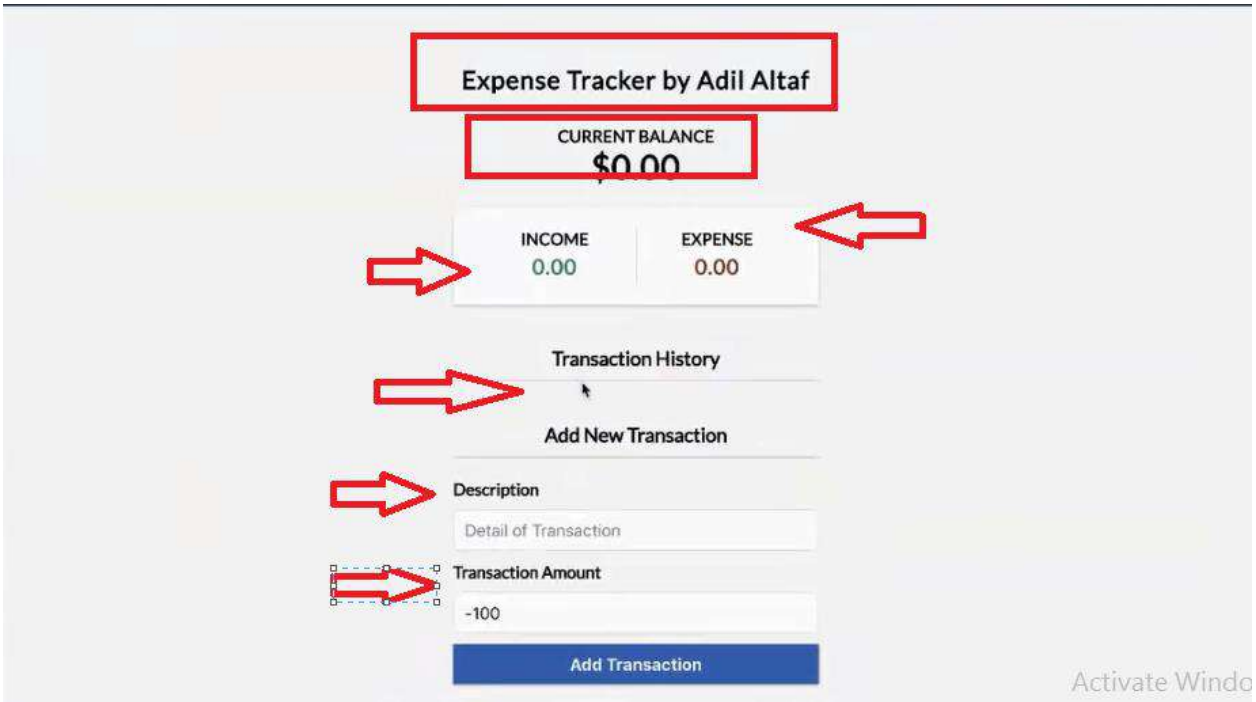
- create React base application as in class 1
- deploy it to surge
- follow up all steps of class 1 to make a basic starter react app



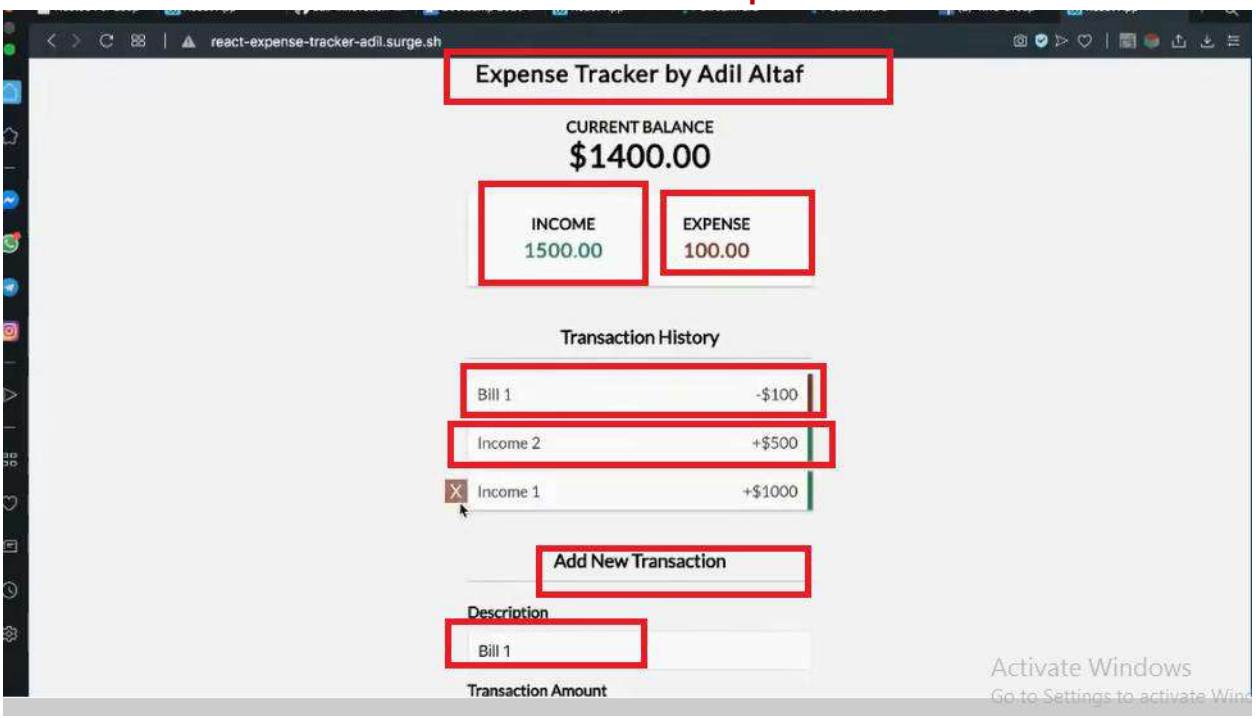


- **To build this expense tracker app we need the following components**
- Header component
- Current balance component
- Income component
- Expense component
- Transaction history
- Add new transaction
- Description

- Transaction amount

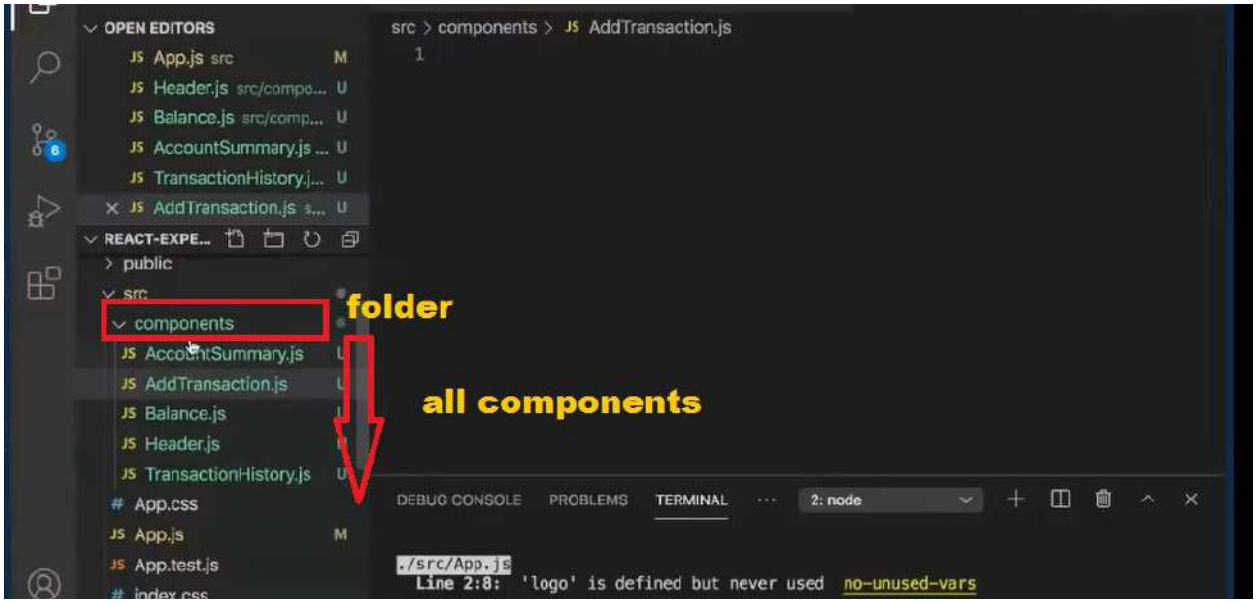


our final outcome should be in this shape that we wanna built so let's built:



Let's create a folder with name" component "and reside all components in it as;

- Header. js
- Balance.js
- Accountsummary.js
- TransactionHistory.js
- AddTransaction.js

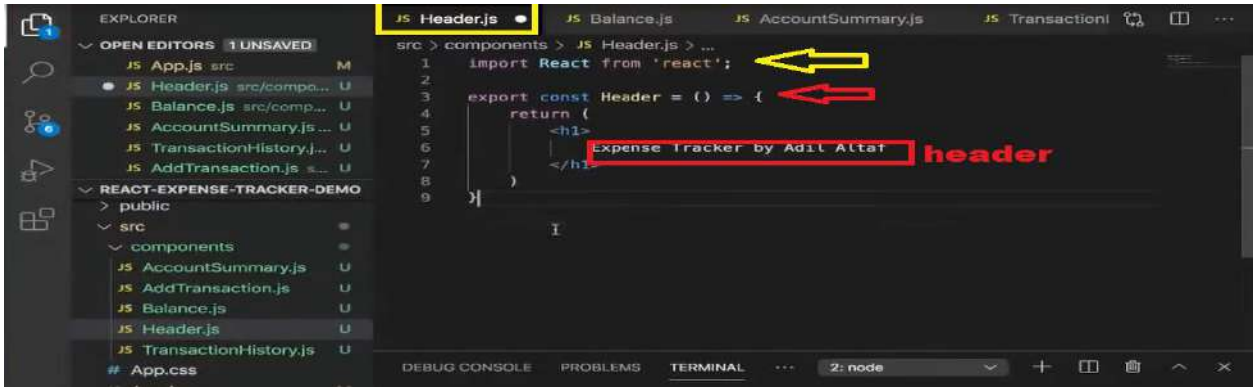


a. Header.js

Let's begin with our 1st component Header.js

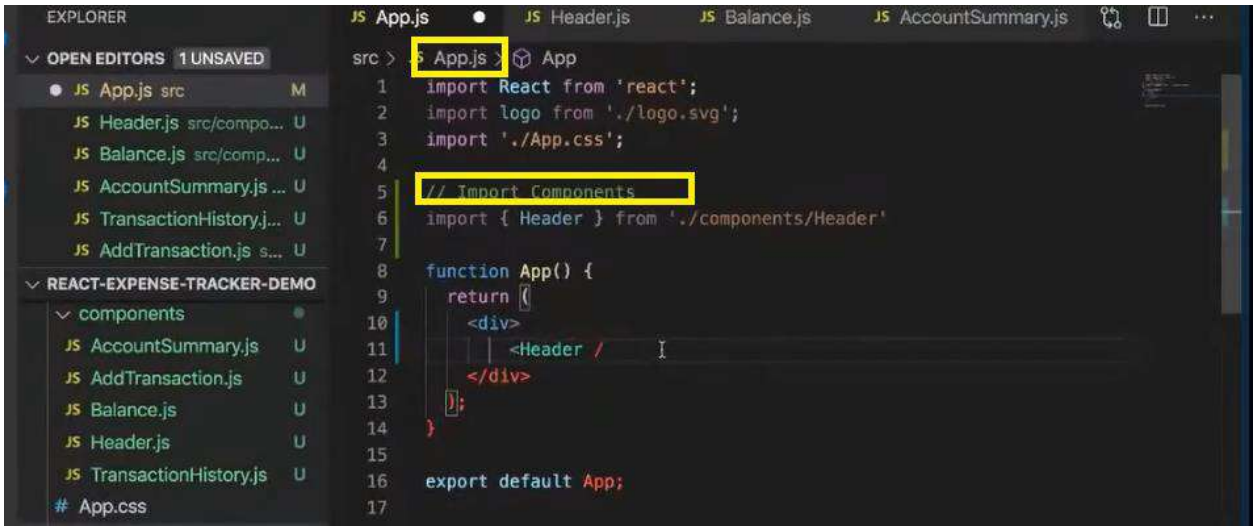
- For that purpose go to the header component file and follow this coding in Header.js
- Meanwhile, connect app.js with browser t show up all activities

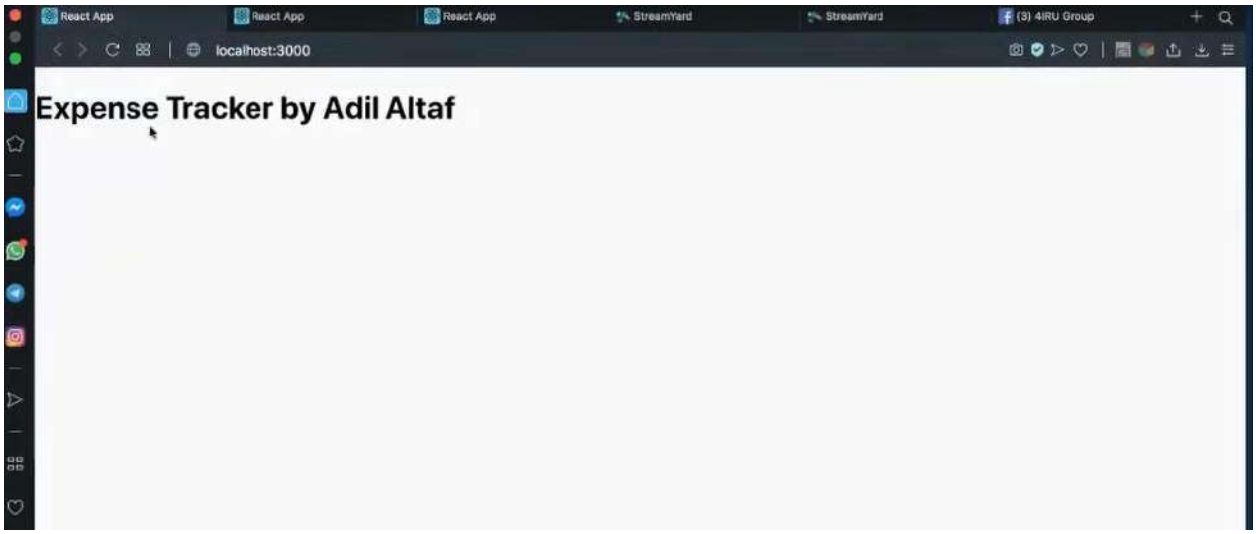
```
1. Import React from 'react';
2. Export const Header =()=>{
3. Return(
4. <h1>
5. Expert tracker by Adil Altaf
6. </h1>
7. );
8. }
```



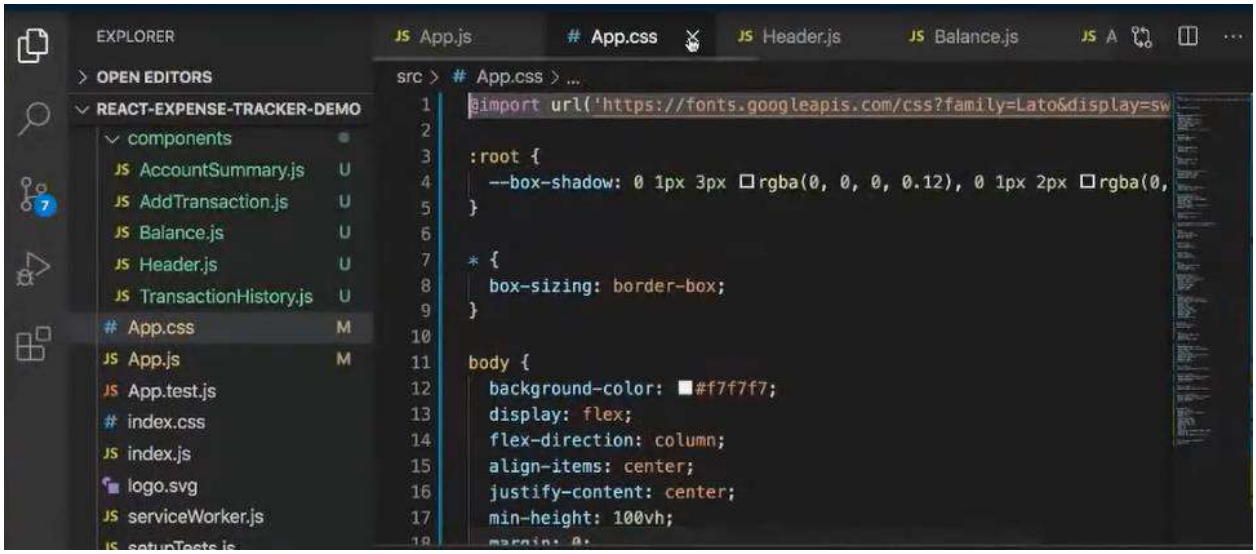
- Now import the header component from the component folder into the App.js file as and then display the output:

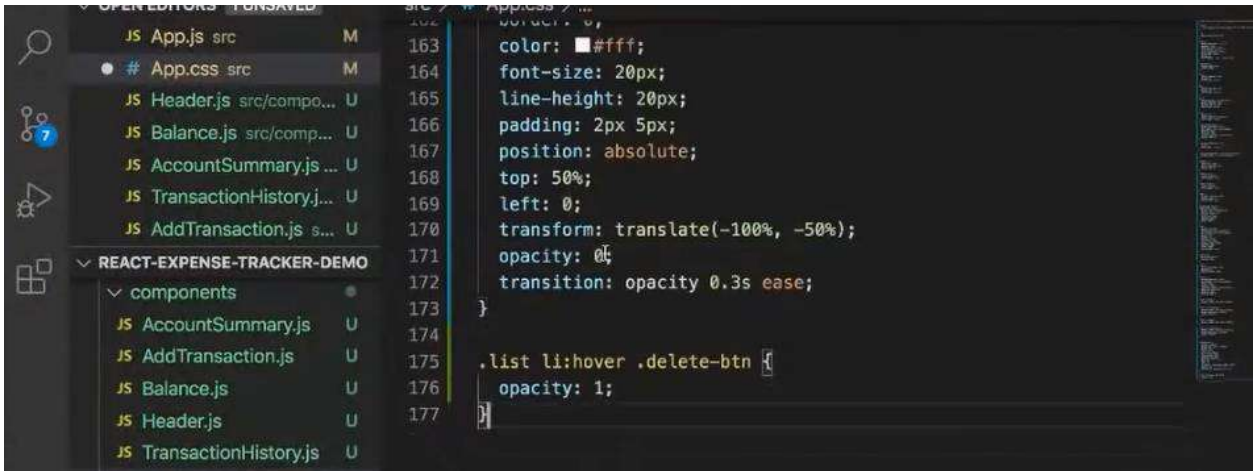
```
import React from ‘react’;
import logo from ‘./logo.svg’;
import ‘./App.css’;
Import {Header} from ‘./components/Header’
Function App(){
Return(
<div>


        <Header>

        </div>

        );

}

export default App;
```





Apply CSS from the following coding

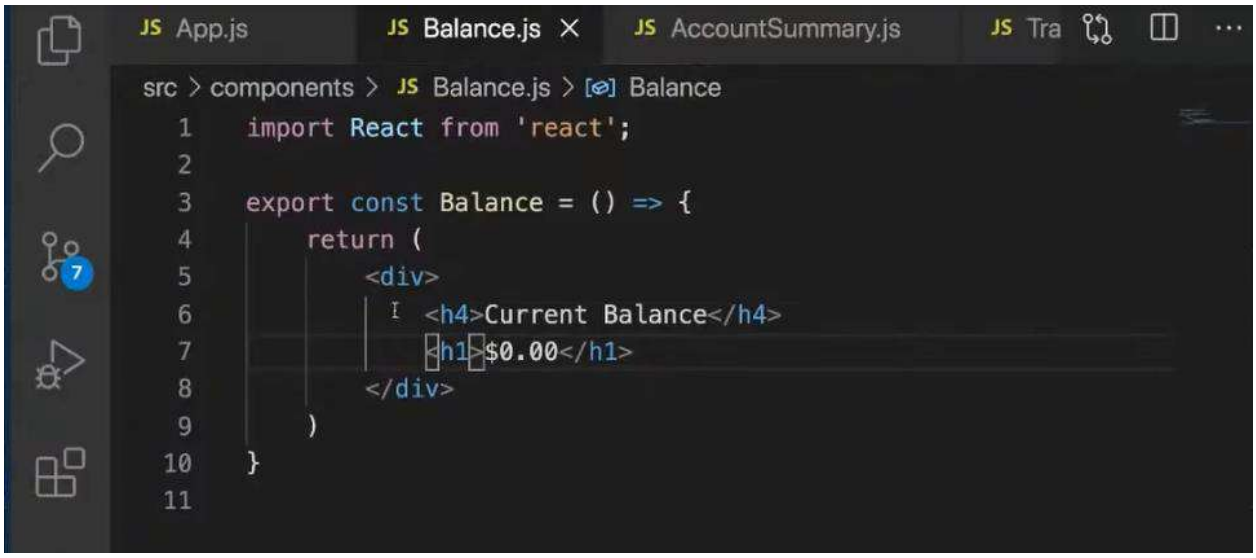




b. Balance.js

- Now create another component for current balance in the component folder as;

```
import React from ‘react’;
Export const Balance = () => {
    Return (
<div>
        <h4>current Balance</h4>
        <h1>$0.00</h1>
        </div>
  )
}
```



- Import balance.js component in the App.js as follows

App.js

```
Import react from ‘react’;
Import {Header}from ‘./components/Header’
import {balance}from './/components/Balance'
Function App(){
        Return(
        <div className="container">
        <balance>
        </div>
        );
        }
```

a. AccountSummary.js

- **Let's create another component "account summary javascript"**

```
import React from 'react'
export const AccountSummary = () = > {
    return(
      <div ClassName="inc-exp-container">
        <div>
         <h4> income</h4>
        </div>
         <div>
         <h4>Expense<h4>
          </div>
       </div>
      )

  }
```

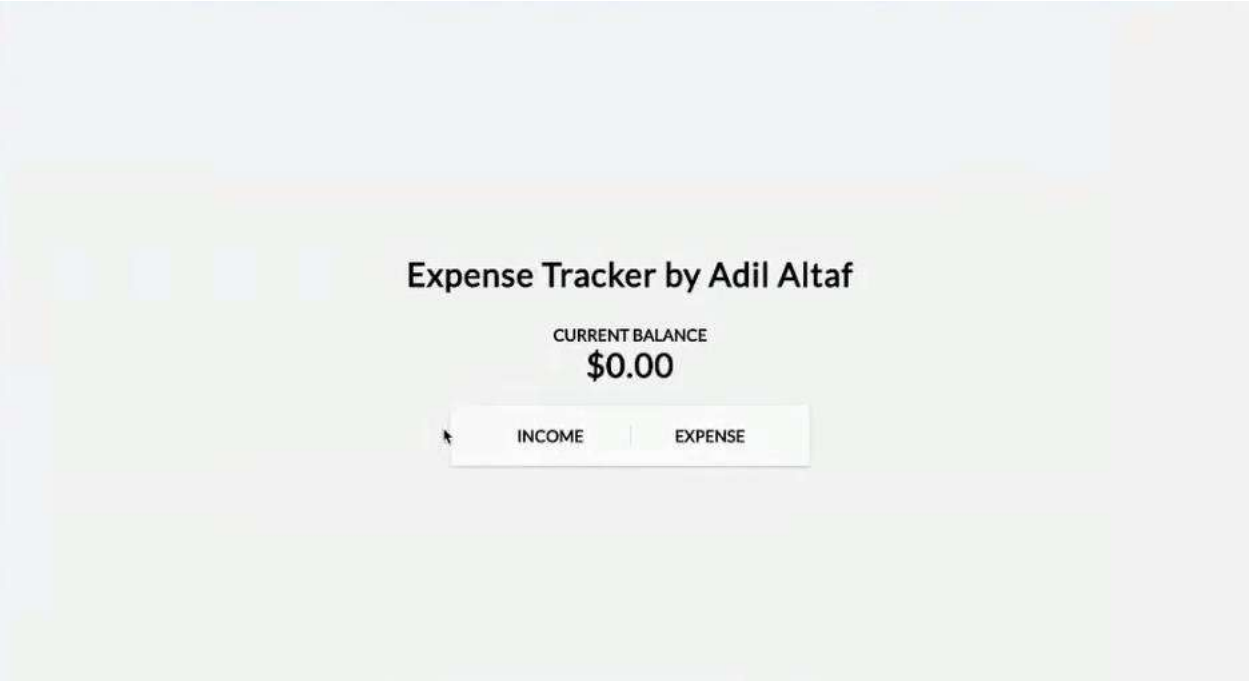
```
JS AccountSummary.js ×   JS TransactionHistory.js   JS AddTransactic
src > components > JS AccountSummary.js > AccountSummary
1   import React from 'react'
2
3   export const AccountSummary = () => {
4       return (
5           <div className="inc-exp-container">
6               <div>
7                   <h4>Income</h4>
8               </div>
9               <div>
10                  <h4>Expense</h4>
11              </div>
12          </div>
13      )
14  }
15
```

- Now import this AcoountSummary.js component into the App.js as;

```
import {AccountSummary} from'.//components/AccountSummary'
function App(){
return(
<div>
<Header/>
<div ClassName="container">
  <Balance/>
  <AccountSummary>
   </div>
 </div>
);
}
```

```
// Import Components
import { Header } from './components/Header'
import { Balance } from './components/Balance'
import { AccountSummary } from './components/AccountSummary'

function App() {
  return (
    <div>
      <Header />
      <div className="container">
        <Balance />
        <AccountSummary
      </div>
    </div>
  );
}
```

- **Look at the display it contains a container of current balance account with expenses as well as income**

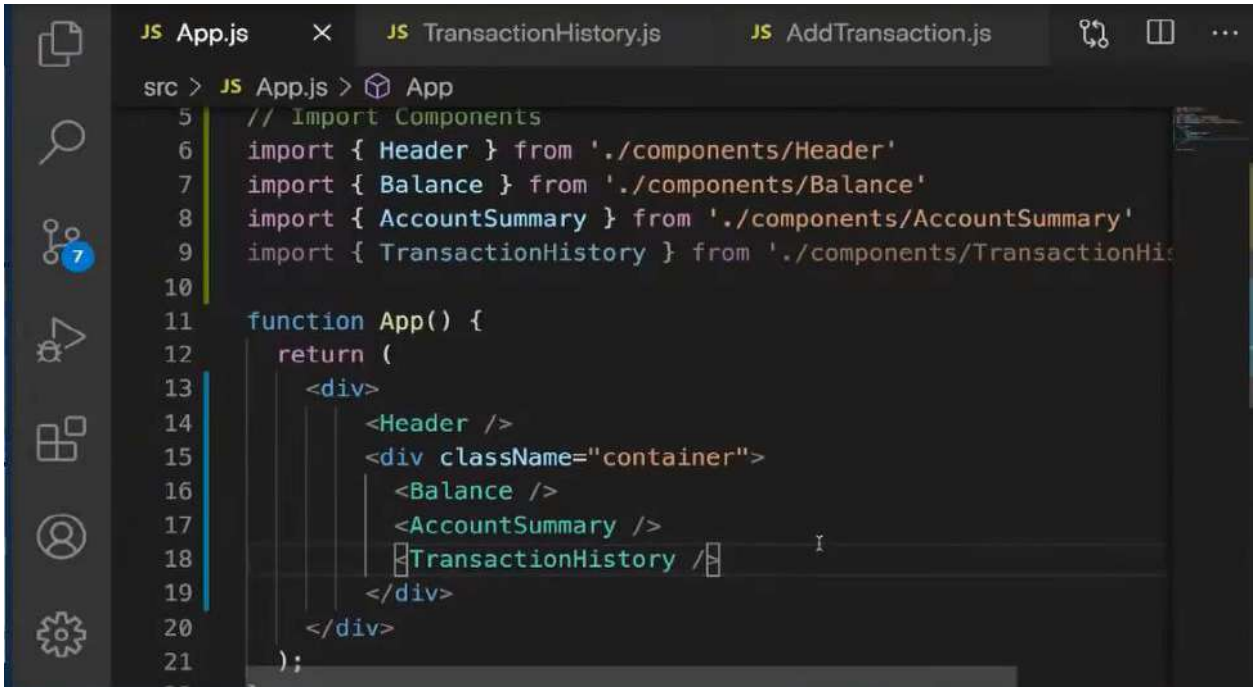


```
src > components > JS AccountSummary.js > [@] AccountSummary
3   export const AccountSummary = () => {
4       return (
5           <div className="inc-exp-container">
6               <div>
7                   <h4>Income</h4>
8                   <p>
9                       $0.00
10                  </p>
11              </div>
12              <div>
13                  <h4>Expense</h4>
14                  <p>
15                      $0.00
16                  </p>
17              </div>
18          </div>
19      )
```

d. TransactionHistory

- Create another component TransactionHistory.js and import it **in App.js file** as;

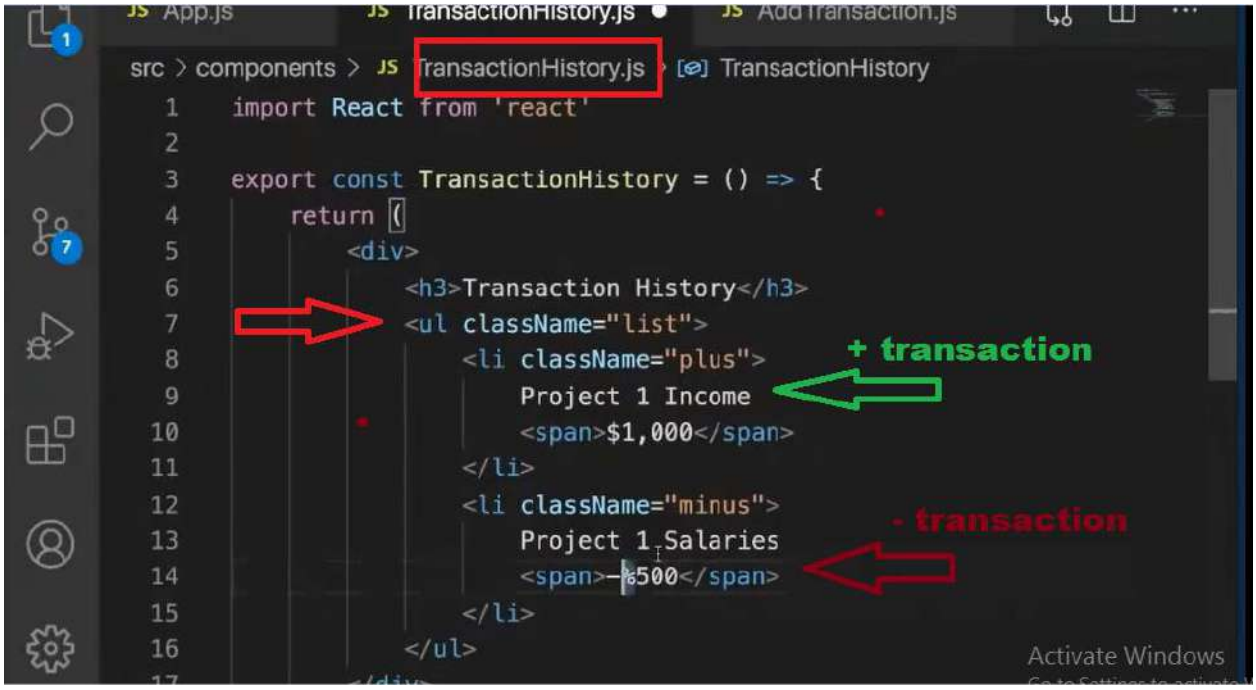
```
Import {Header} from'./components/Header'
Import {Balance}  from'./component/Balance'
Import  {AccountSummary} from'.//components/AccountSummary'
import {transactionHistory} from './components/TransactionHistory.js
function App(){
return(
<div>
<Header/>
<div ClassName="container">
  <Balance/>
  <AccountSummary>
</transactionHistory>
   </div>
 </div>
);
}
```



In TransactionHistory.js follow up the following coding:

Add transaction class, order list + for increment and – for decrement add Description amount as $1000>. Project 1 income. Positive transaction will be shown by the green color and transaction will be shown by the brown color. This is how it would look like in your browser.

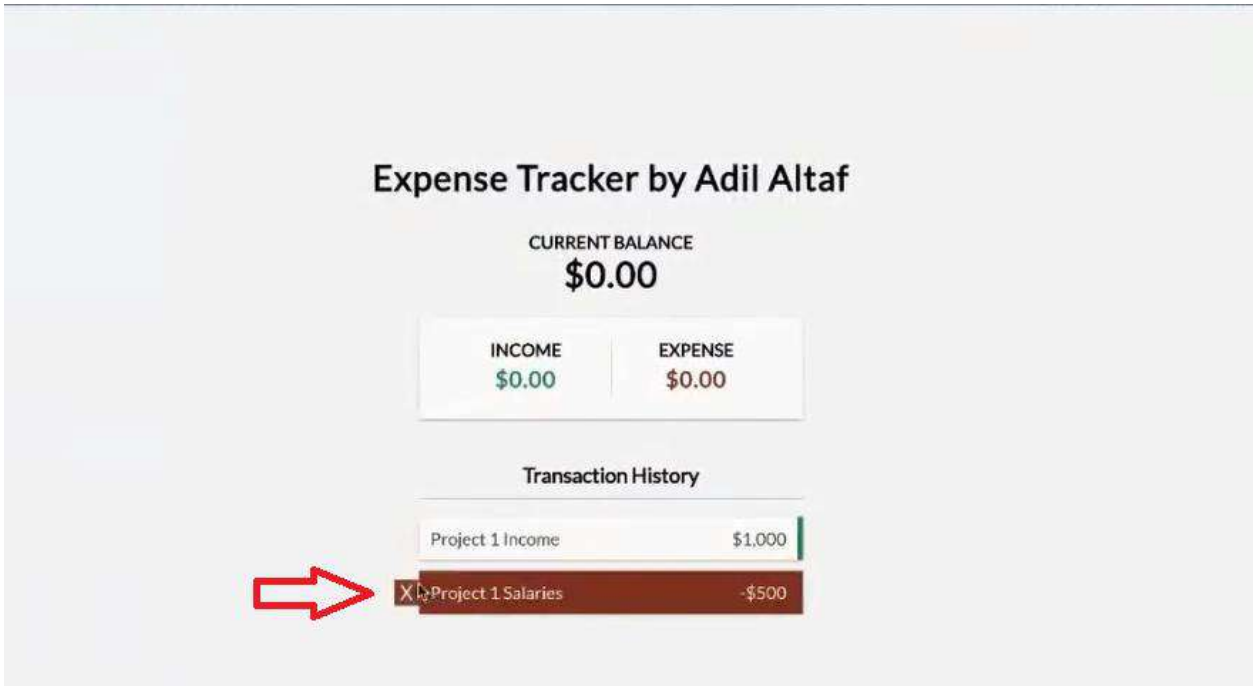
```
import React from 'react';
export const TransactionHistory=()=>{
return(
     <div>
       <h3>Transaction History</h3>
        <ul className="List">
         <li className="plus">
         project 1 Incom
         <span>$1,000</span>
         </li>
         <li className="minus">
            project 1 Salaries
         <span>-500</span>
         </li>
       <ul>
    <div>
   );
 }
```



- In order to remove any of transaction value you will have to create a "delete button "

<button className="delete-btn">X</button>

```
1. import React from 'react';
2. export const TransactionHistory=()=>{
3. return(
4. <div>
5. <h3>Transaction History</h3>
6. <ul className="List">
7. <li className="plus">
8. project 1 Incom
9. <span>$1,000</span>
10. <button className="delete-btn">X</button>

11. </li>
12. <li className="minus">
13. project 1 Salaries
14. <span>-500</span>
15. <button className="delete-btn">X</button>

16. </li>
17. <ul>
18. <div>
19. );
```



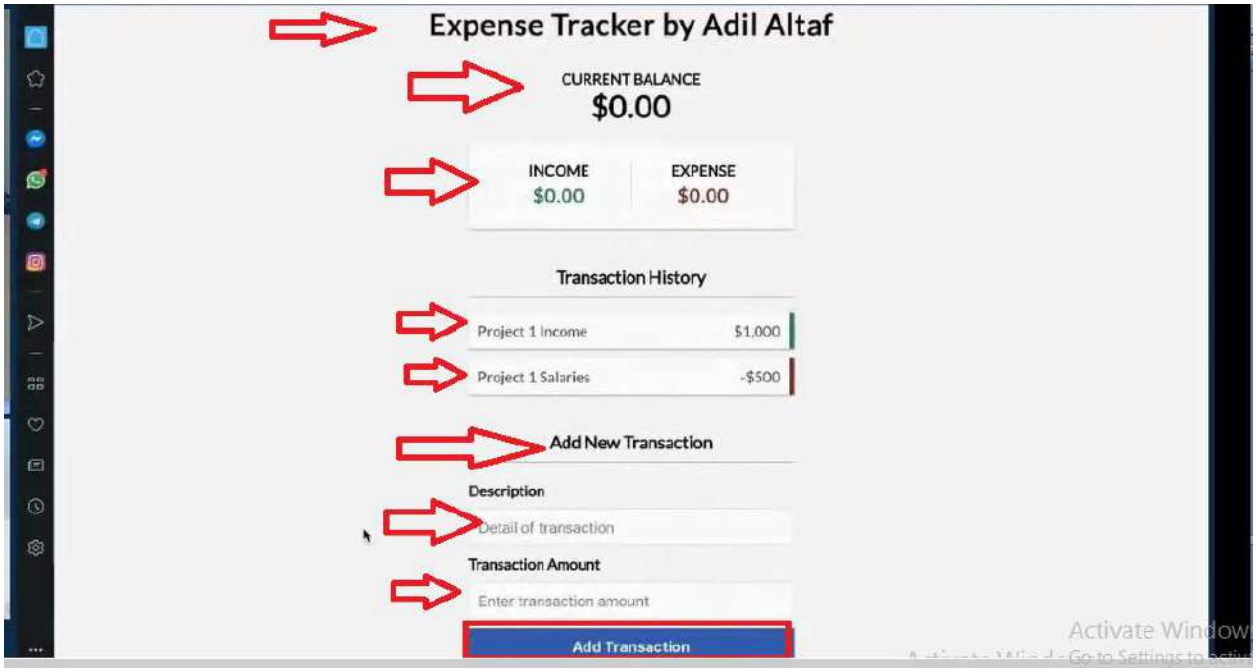
e. AddTransaction.js

- for this we need descripion intext form
- value to be entered in number

  - so continue with this coding in Addtransaction.js
  - put heading of “Add New Transaction”
  - make a form and label its description and amount of transaction as shown;
  - in order to add more “Add transaction” create a button as :

```
import React from 'react';
export const AddTransaction=()=>{
  return(
   <div>
      <h3>Add New Transaction</h3>
      <form>
        <div ClassName="form-control">
          <label htmlFor='description'>
             Description
           </label>
           <input  type="text"
            id="description"
            placeholder="Detailof transaction"
            />
      </div>
<div className="form-control">
<label htmlFor="transactionamount">
    Transaction Amount
</label>
<input type="number
    id="transactionamount"
    placeholder="enter transaction amount

   />
</div>
<button className=>"btn">
   Add Transaction
</button?
</form>
</div>
)
```

Our final outlook of this app is as shown :



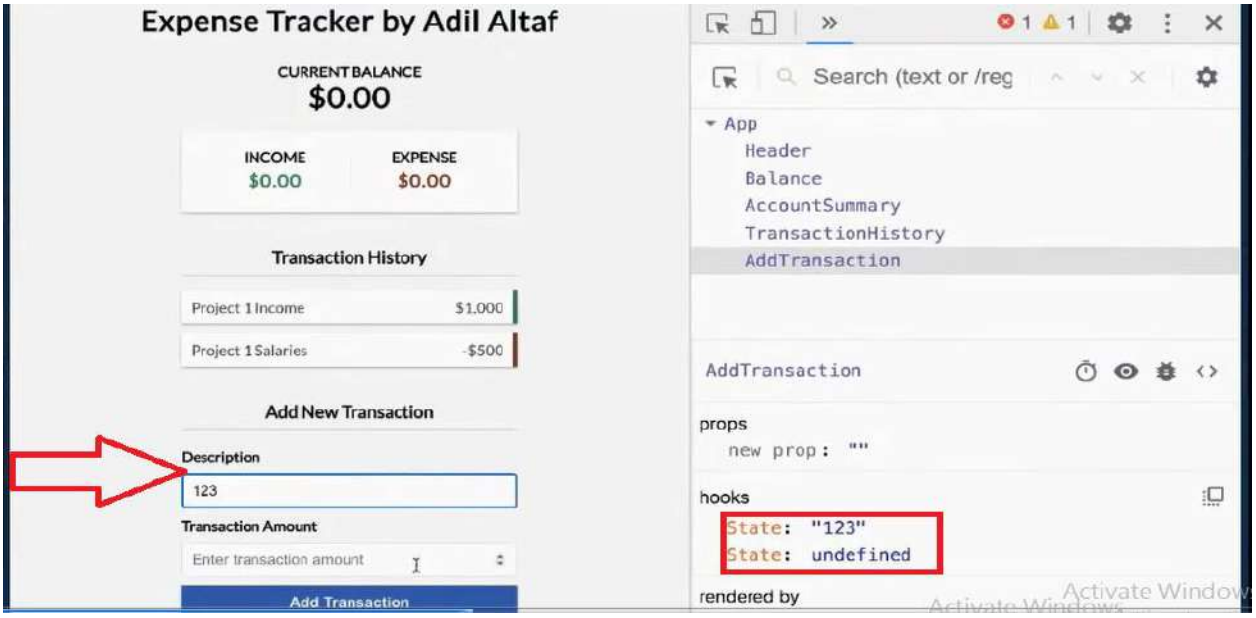
- **To store data in the description and transaction we are going to use "useState"**
- we will take 2 const, one for description and the other for transaction
- then we will link these 2 useState with the Actual forms as

  - imoprt const Addtransaction=()=>{
  - const [description,setDescription] =useState();
  - const [transactionAmount,,setTransaction] =useState();

- to link usestate add this in input form as
  - value={description}
  - value={description}
- now for triggering any kind of update in form input add the following funstion to update state
  - onChange={(e)=>setDescription(e.target.value)}
  - onChange={(e)=>setTransaction(e.target.value)}

```
1.  import React from 'react';
2.  imoprt const Addtransaction=()=>{
3.  const [description,setDescription] =useState();
4.  const [transactionAmount,,setTransaction] =useState();
5.  export const AddTransaction=()=>{
6.    return(
7.     <div>
8.        <h3>Add New Transaction</h3>
9.        <form>
10.        <div ClassName="form-control">
11.          <label htmlFor='description'>
12.             Description
13.          </label>
14.          <input  type="text"
15.           id="description"
16.           placeholder="Detailof transaction"
17.            value={description}
18.            onChange={(e)=>setDescription(e.target.value)}
19.           />
20.       </div>
21.<div className="form-control">
22.<label htmlFor="transactionamount">
23.    Transaction Amount
24.</label>
25.<input type="number
26.    id="transactionamount"
27.    placeholder="enter transaction amount
28.                  value={Transaction}
29.onChange={(e)=>setTransaction(e.target.value)}
30.   />
31.</div>
32.<button className=>"btn">
33.   Add Transaction
34.</button?
35.</form>
36.</div>
37.)
```

By executing the above code you will get all your transaction on be updated whenever you trigger any kind of change as shown



f. context

- in order to pass values manually and link it.
- create a component **GlobalState.js inside** context folder

```
1. import react,{createContext, useReducer} from 'react';
2. Import AppReducer from './AppReducer.js';
3. const initialState={
4. transation:[
5. {id:1,description:"income 1", transactionAmount:1000},
6. {id:2,description:"expense 1", transactionAmount:-100},
7. {id:3,description:"income 2", transactionAmount:2000},
8. {id:4,description:"expense 2", transactionAmount:-500},

//so in oder to use all above trasaction to be used all over the application we will have to creat GlobalContext const as
9. export const GlobalContext =createContext(initialState):
//next create a provider for global context as:
10. export const GlobalProvider=>{
// create a reducer that will perform as a reducer
11. const[state,dispatch] =useReducer(AppReducer,initialState)
12. return (
13. <GlobalContext.Provider value={
14. transactions:state.transaction
15. }
16. }>
// allow access to  children like header.js,balnace account,summary, history
17. {children}
18. </GlobalContext.provider>
19. );
20. }
```

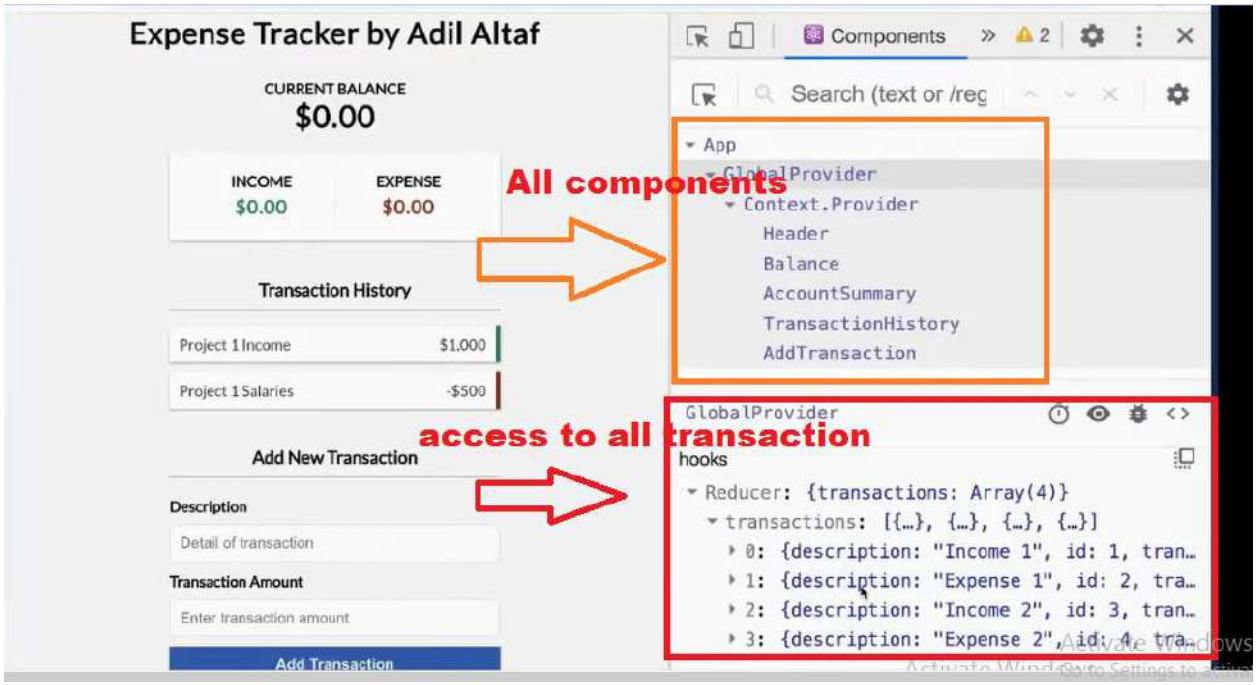
- Now create **Reducer.js** that will return a state whenever any action will be performed in it and we will have to import this new component into **App.js**

AppReducer.js

```
// which state and action are to be performed:

export default(state, action)=>

    switch(action.type){

   default:

   return state;
```

app.js

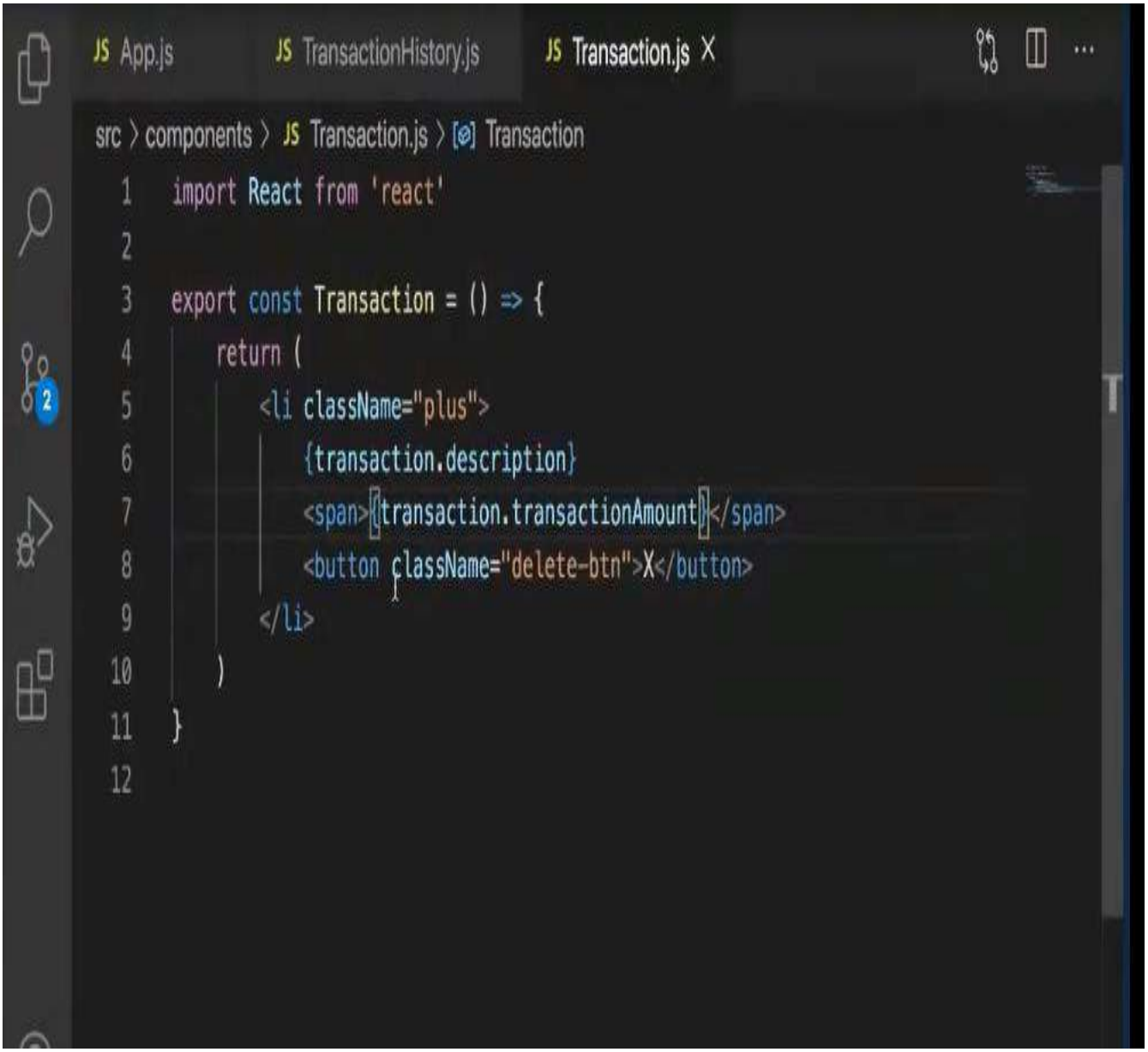
```
import {GlobalProvider } from ./context/GlobalState
      Import{Header} from'./components/Header'
      Import{Balance} from'./component/Balance'
      import {AccountSummary} from'.//components/AccountSummary'
      import {transactionHistory} from './components/TransactionHistory.js
      import {transactionHistory} from './components/TransactionHistory.js

      function App(){
      return(
      <div>
      <Header/>
      <div ClassName="container">
        <Balance/>
        <AccountSummary>
      </transactionHistory>
         </div>
       </div>
      );
```



- Now to display all transaction make another component transaction.js as

transaction.js

```
import React from 'react';

export const Transaction=(transactiob-n)=>{

  return(

        <li ClassName="plus">

      {transaction.description}

     <span>{transaction.transactionAmount}</span>

     <button ClassName="delete-btn">X</button>

      </li>

      )
```

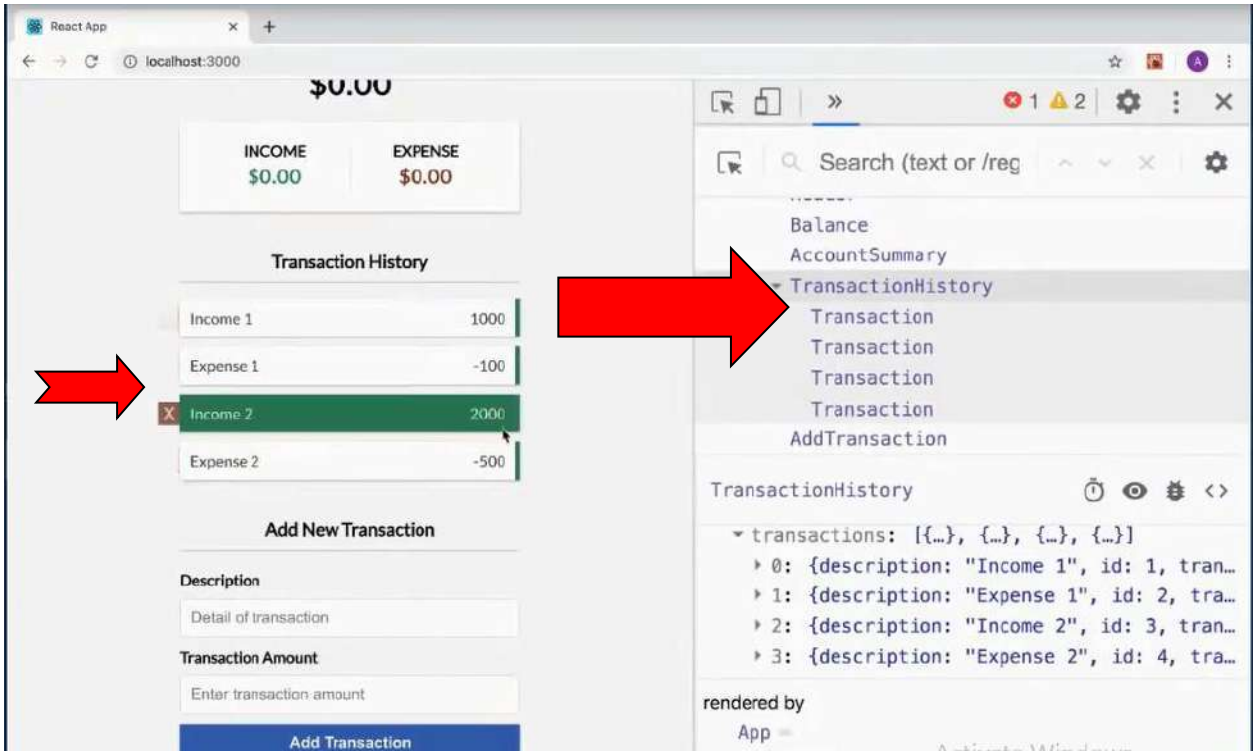


- Now export this transaction to the transactionHistory.js and use map to display all transaction as **transaction.map(transaction=>)**

```
import react,{useContext}from'react';
import {globalContext} from './context/GlobalState';
import{Transaction} from './Transaction
export const TransactinHistory=()=>{
  const {transaction}=useContext(GlobalContext);
return(
   <div>
   <h3> Transaction History</h3>
<ul  ClassName='list>
  {transaction.map(transaction=>)
<transaction transaction={transaction}/>
</ul>
</div>
)
}
```

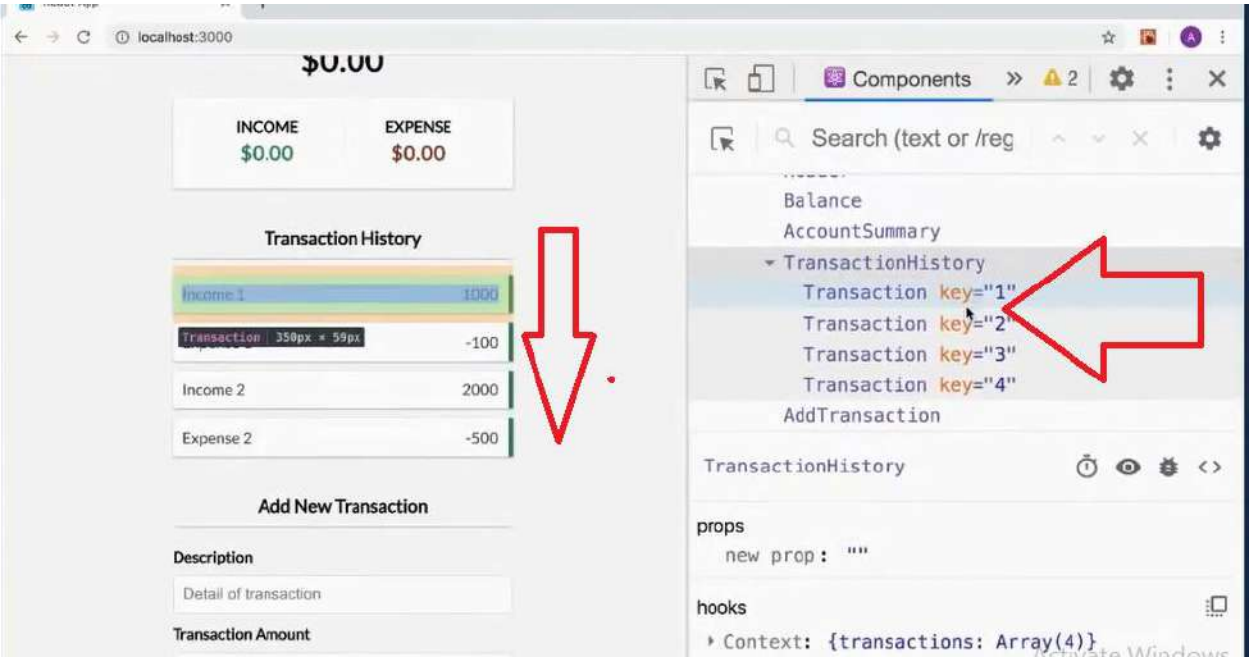
```
JS App.js    JS TransactionHistory.js ●    JS Transaction.js
src > components > JS TransactionHistory.js > [@] TransactionHistory
3   // import the Global Context
4   import { GlobalContext } from '../context/GlobalState';
5
6   // Import Transactions
7   import { Transaction } from './Transaction'
8
9   export const TransactionHistory = () => {
10
11      const { transactions } = useContext(GlobalContext);
12
13      return (
14          <div>
15              <h3>Transaction History</h3>
16              <ul className="list">
17                  {transactions.map(transaction => )
18                  <Transaction transaction={transaction} />
19              </ul>
20          </div>
21      )
22  }
23
```

Now it will display all transaction;



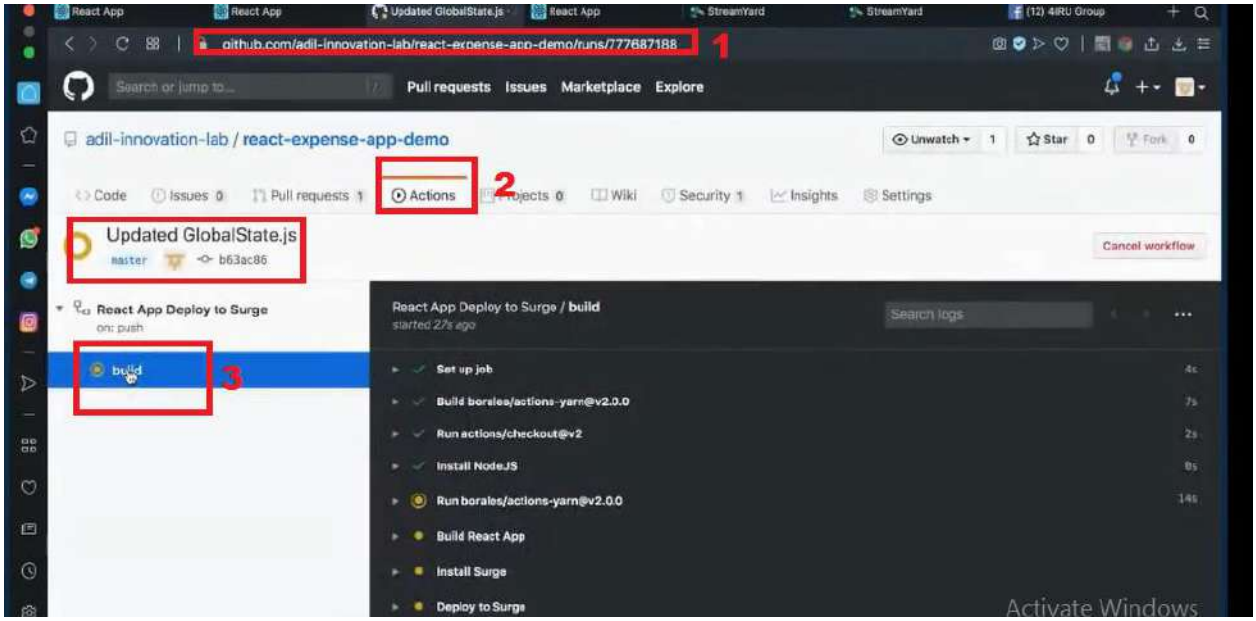
You can also use the key to access all of the transaction by providing id of that transaction

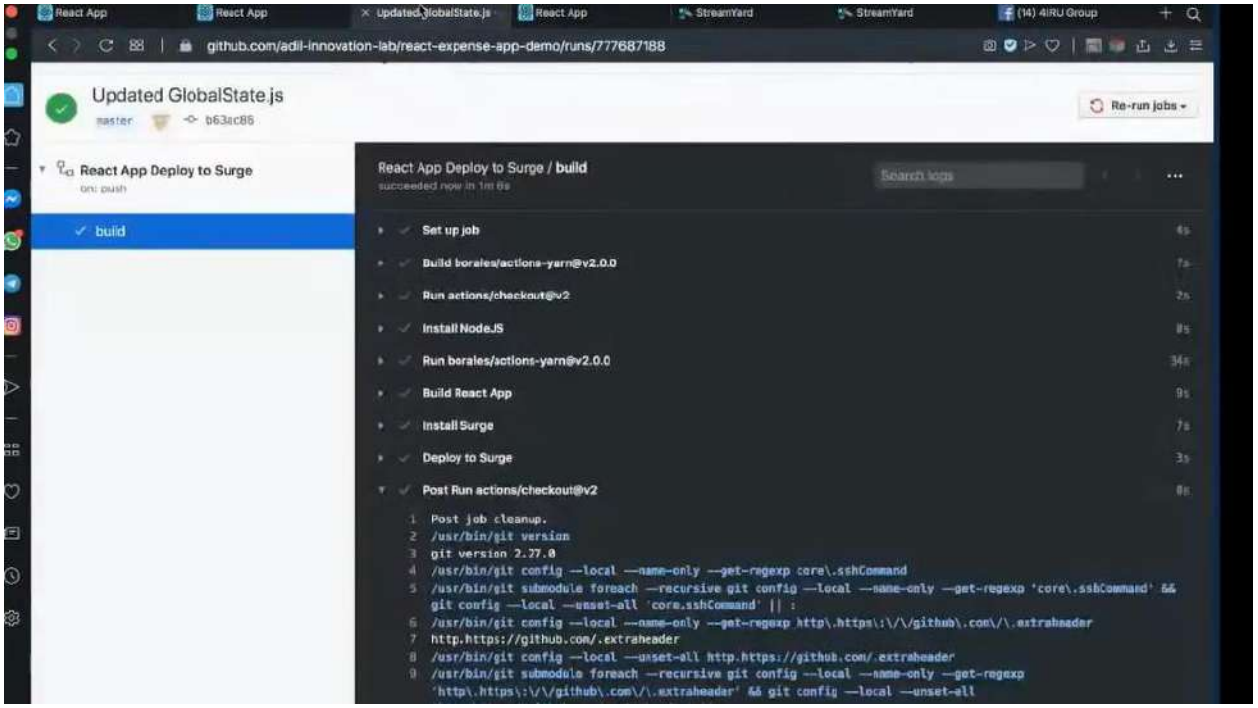
<Transaction key={transaction.id} transaction={transaction}

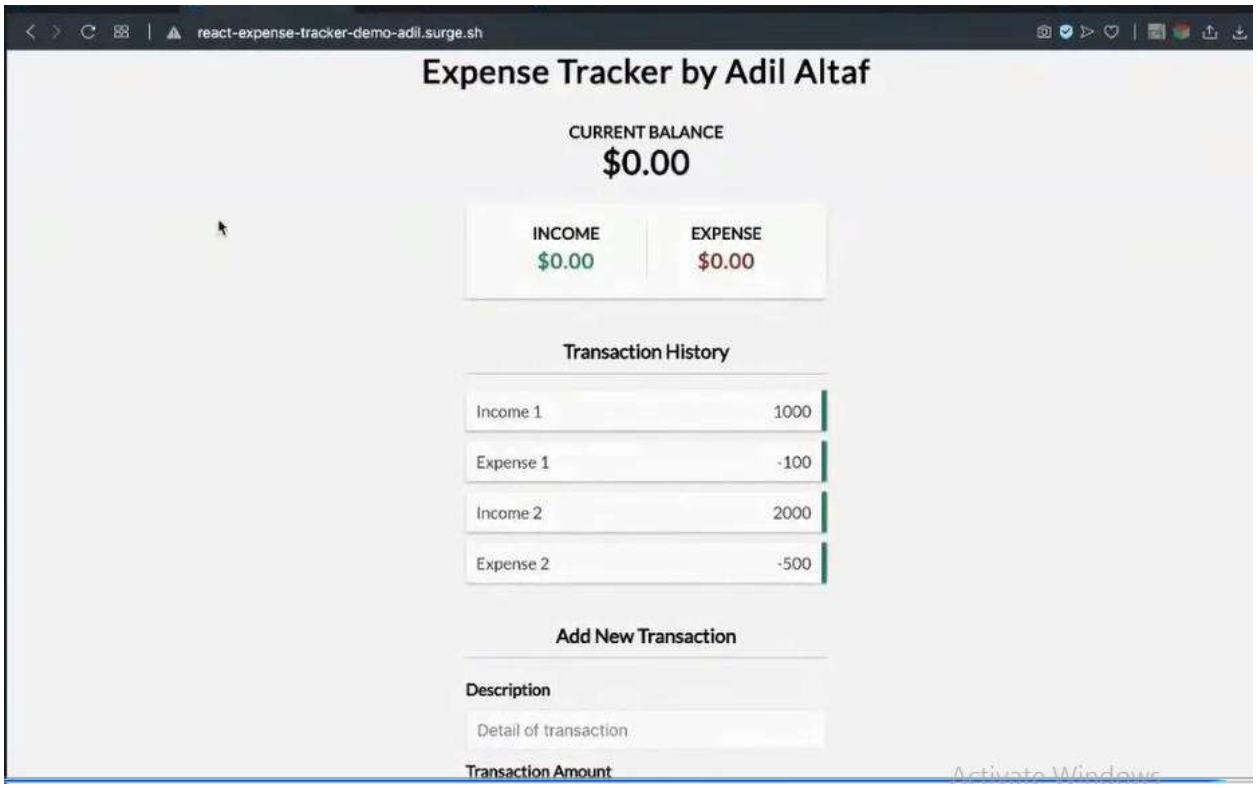


So now push this code into the origin and to the GitHub repository as we did in the previous section

Select file code, click to action, go to workflow and click to build than all of the installations gets completed this application would be easy on your Github repository as;




React App
React App
Updated GlobalState.js
React App
StreamYard
StreamYard
(14) 4IRU Group
github.com/adil-innovation-lab/react-expense-app-demo/runs/777687188
Updated GlobalState.js
master
b63ac86
Re-run jobs
React App Deploy to Surge
on: push
build
React App Deploy to Surge / build
succeeded now in 1m 6s
Search logs
Set up job
Build borales/actions-yarn@v2.0.0
Run actions/checkout@v2
Install NodeJS
Run borales/actions-yarn@v2.0.0
Build React App
Install Surge
Deploy to Surge
Post Run actions/checkout@v2
Post job cleanup.
/usr/bin/git version
git version 2.27.0
/usr/bin/git config --local --name-only --get-regexp core\.sshCommand
/usr/bin/git submodule foreach --recursive git config --local --name-only --get-regexp 'core\.sshCommand' && git config --local --unset-all 'core.sshCommand' || :
/usr/bin/git config --local --name-only --get-regexp http\.https\:\/\/github\.com\/\.extraheader
http.https://github.com/.extraheader
/usr/bin/git config --local --unset-all http.https://github.com/.extraheader
/usr/bin/git submodule foreach --recursive git config --local --name-only --get-regexp 'http\.https\:\/\/github\.com\/\.extraheader' && git config --local --unset-all


react-expense-tracker-demo-adil.surge.sh
Expense Tracker by Adil Altaf
CURRENT BALANCE
$0.00
INCOME
$0.00
EXPENSE
$0.00
Transaction History
Income 1 1000
Expense 1 -100
Income 2 2000
Expense 2 -500
Add New Transaction
Description
Detail of transaction
Transaction Amount

Session: 7 React

1. Synchronous And Asynchronous Coding

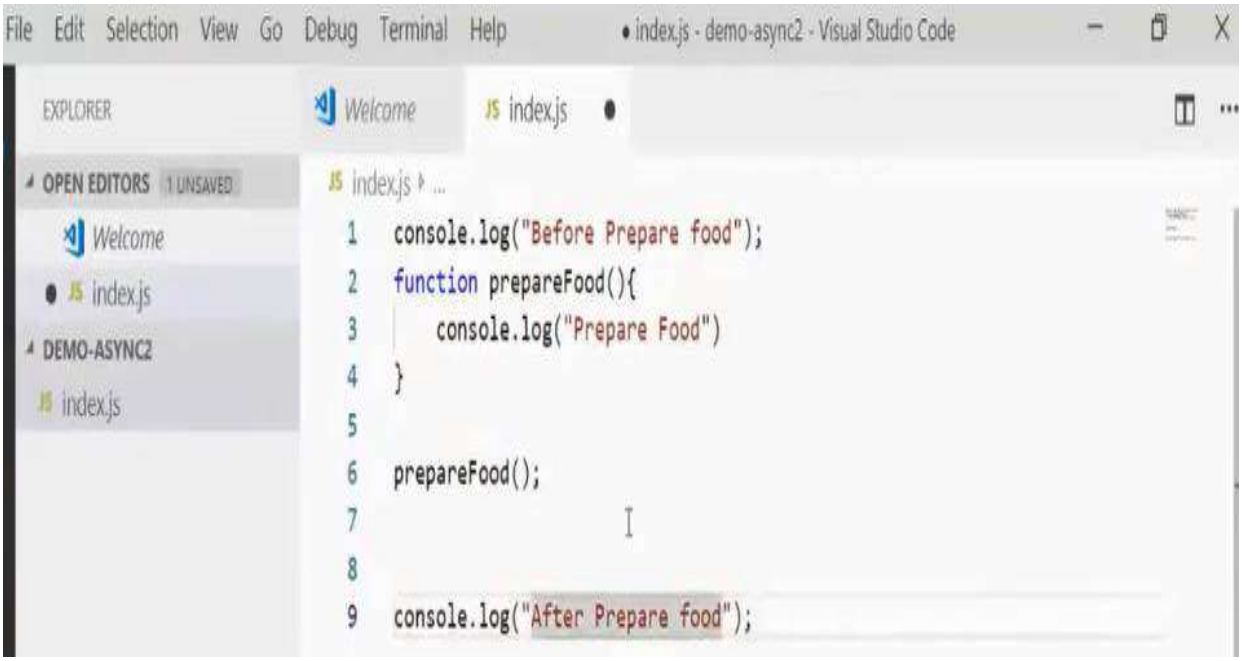
In programming, synchronous operations block instructions until the task is completed, while asynchronous operations can execute without blocking other operations. Asynchronous operations are generally completed by firing an event or by calling a provided callback function.

2. Synchronous Coding Example

- **Let's take n example of a normal synchronous function call, go to the index.js and write the following coding**

index.js

```
1. console.log("Before prepare food");
2. function prepareFood(){
3. console.log("Prepare Food");
4. }
5. prepareFood();

6. console.log ("After Prepare food dinner is ready")
```



- Now link index .js with index.html and write Hello world in it

index.html

```
1. <DOCTYPE html>
2. <html lang="en">
3. <head>
4. <title>Document</title>
5. <script src="index.js"></script>
6. </head>
7. <body>
8. Hello World
9. </body>
10. </html>
```

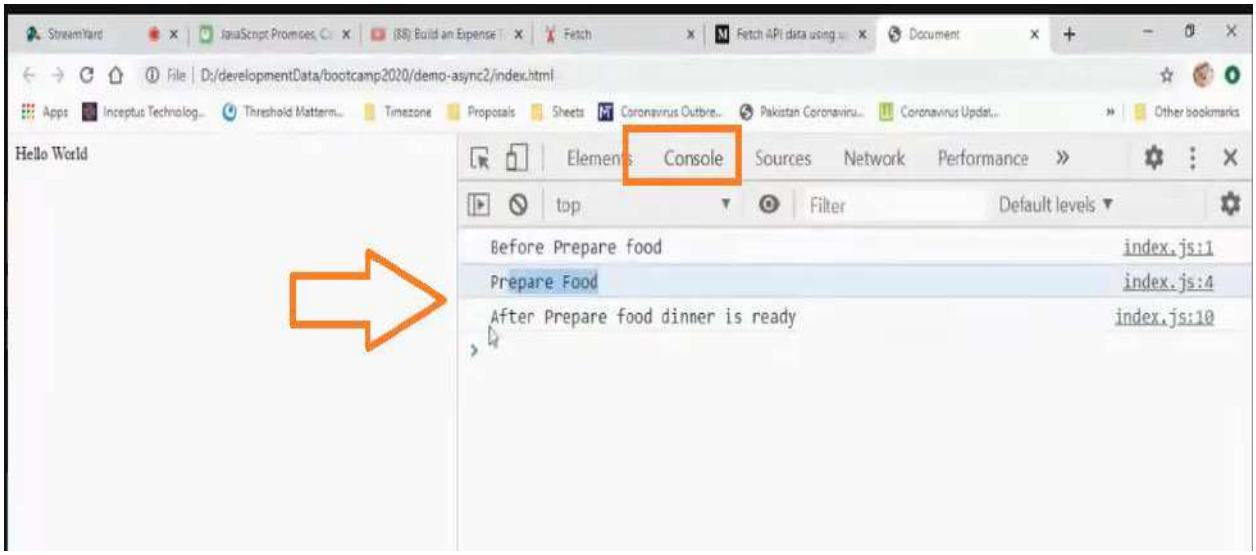
2.1. How synchronous Coding Works:

firstly this line "console.log("Before prepare food");" will be executed then this function"function prepareFood(){" would be declare ;once we call this function execution will transfer to

this function of console log "console.log("Prepare Food")" then this will be executed and after that, it will be returned to "prepareFood():" and once the food is return we will go to "console.log ("After Prepare food dinners ready")"

this is how in a sequence synchronous coding execute and run;

- **go to the browser open up the coding in it**



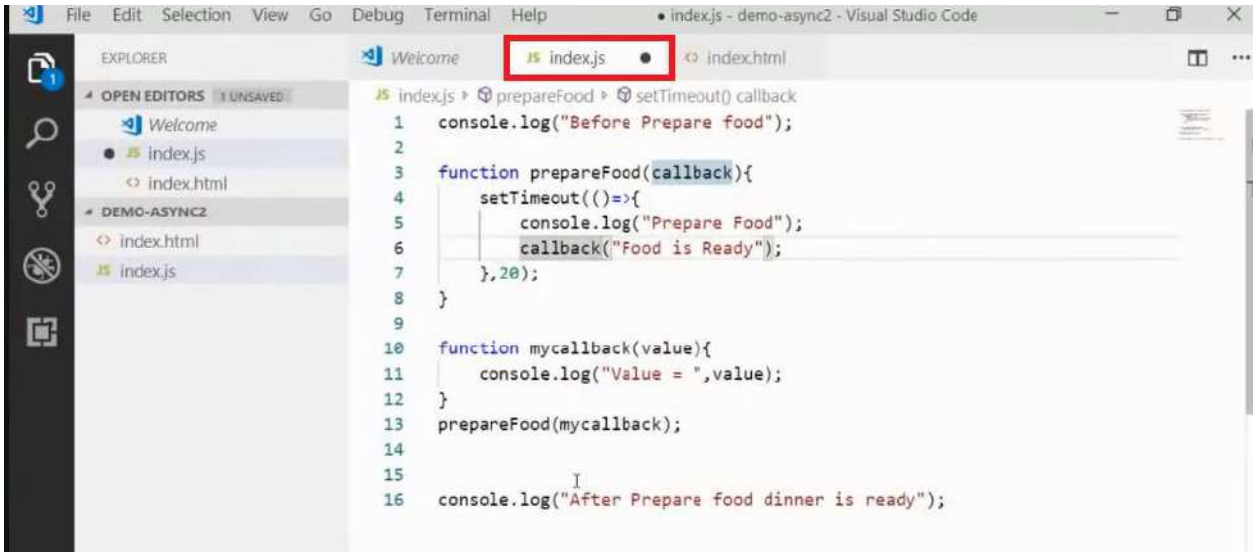
3. Asynchronous coding

- to create a background call, set timeout it requires time, runs timeline behind the scene
- add call back to get to know that our work is being done qt its timelines
- function calls another once it has performed its execution so to proceed with the execution of further function at the proper timeline as added by time call

3.1. Coding Example of Asynchronous

- Go to the index.js and follow up the following coding of food prepared as

```
console.log("Before Prepare Food");
function prepareFood(callback){
setTimeout(()=>{
console.log("prepare Food");
callback("Food is Ready");
},20);
}
function myCallback (value){
console.log("value= ",value);
}
preapareFood(mycallback);
console.log("After Prepare Food Dinner is ready");
```



3.2. How Does Asynchronous coding Work

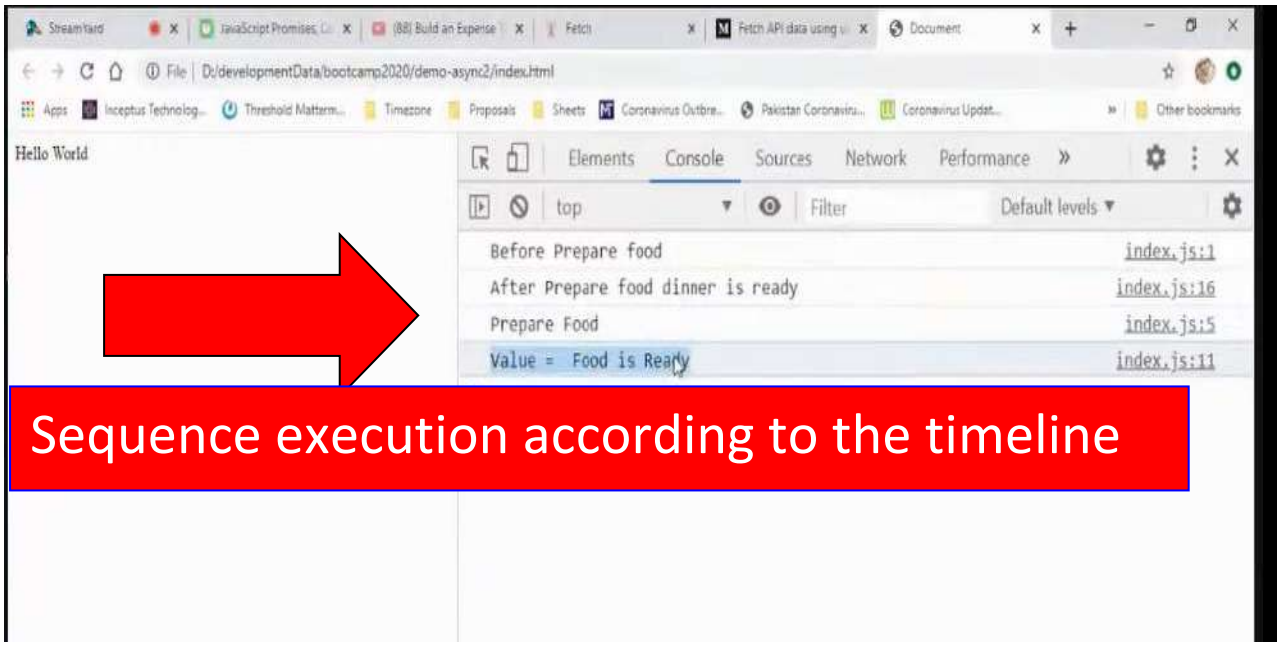
- once this javascript execution starts it would execute the "console.log("Before Prepare Food")"
- then it will declare function;

```
"function prepareFood(callback){
setTimeout(()=>{
    console.log("prepare Food");
    callback("Food is Ready");
  },20);
}
```

- it will declare another function:

```
function myCallback (value){
  console.log("Value= ",value);
}
```

then it will call prepareFood and pass the function(mycallaback) and you will get to know that the food is ready



3.3. Example: consider we want to prepare food and French toast in a different span. First food is being cooked by our chef and then we want him to cook French toast then how this asynchronous coding and execution will proceed at the provided time slot

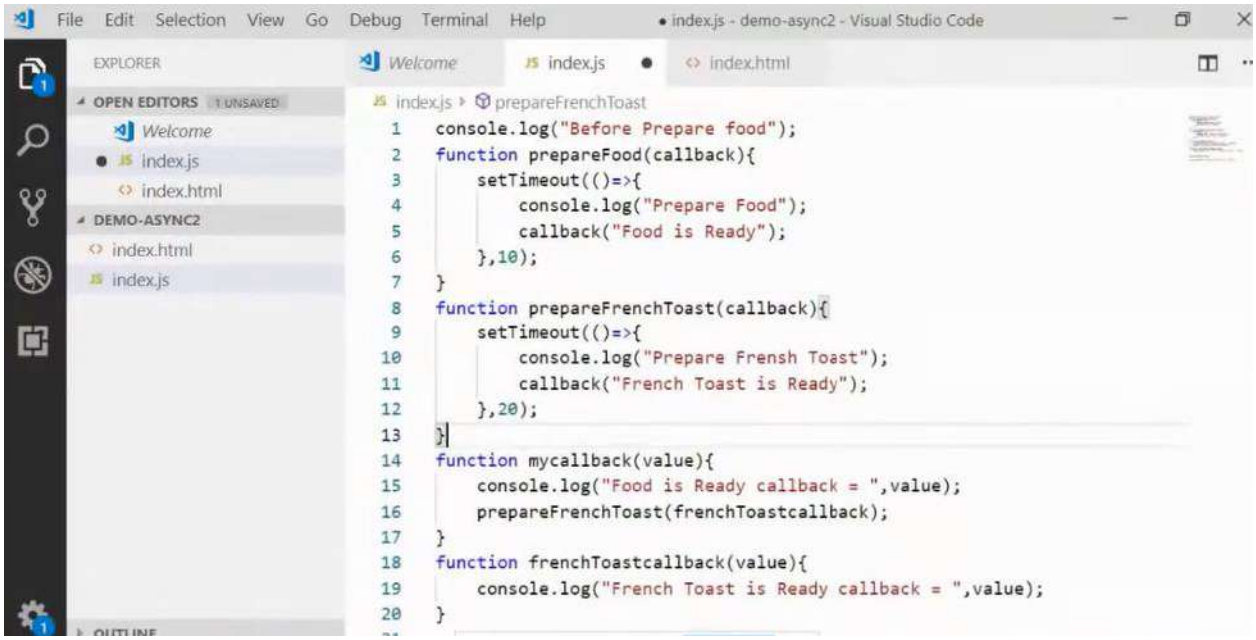
- **Food:10 minute**
- **French toast:20 minute after the food is ready**

Solution To The Problem

So let's get started towards the solution

- Go to the index.js and write this code

```
console.log("Before Prepare Food");
function prepareFood(callback){
setTimeout(()=>{
console.log("prepare Food");
callback("Food is Ready");
},10);
}
function prepareFrenchToast(callback(){
setTimeout(()=>{
console.log("prepare french toast");
callback(" french toast is Ready");
},20);
  }

}
function mycallback(value){
console.log("Food is ready  callback=",value);
prepareFrenchToast(frenchToastcallback);
}
function frenhToastcallback(value){
console.log("french Toast is ready callback=",value);
}
prepareFood(mycallback);

console.log ("After Prepare food dinner is ready");
```



How Does It Work?

- **prepare food function is executed first in a separate timeline of 10**

```
console.log("Before Prepare Food");
function prepareFood(callback){
setTimeout(()=>{
   console.log("prepare Food");
   callback("Food is Ready");
  },10);
```

- **then it will go to function mycallback(value)that will return food is ready and completion of first food preparation execution**

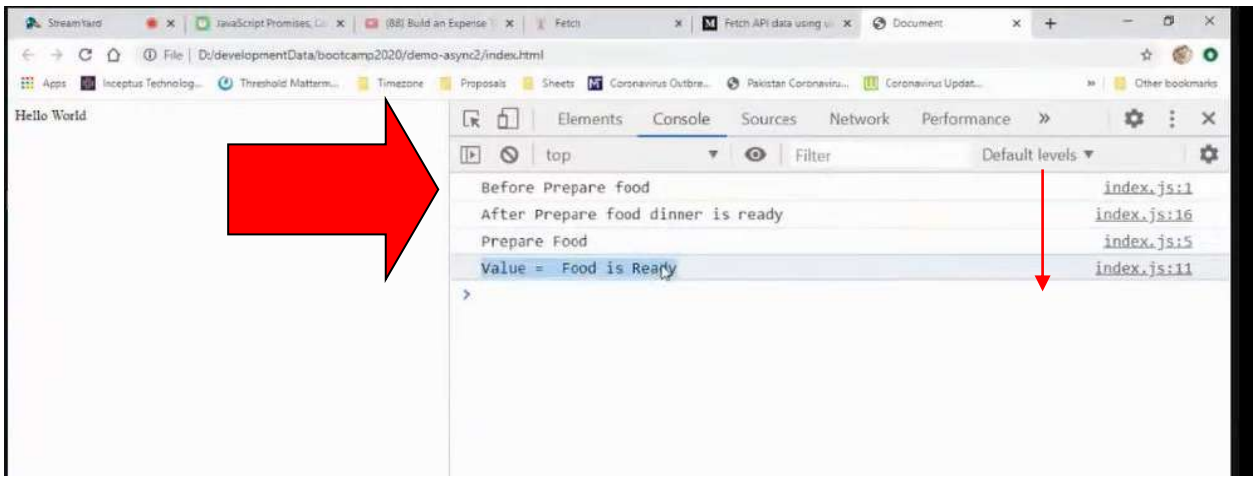
```
function mycallback(value){
  console.log("Food is ready  callback=",value);
  prepareFrenchToast(frenchToastcallback);
```

- **and this will callback next function of french toast as**

```
function prepareFrenchToast(callback){
setTimeout(()=>{
    console.log("prepare french toast");
    callback(" french toast is Ready");
  },20);
```

- **then this function will run function frenchToastcallback in differnt alloted time span**

```
function frenhToastcallback(value){
 console.log("french Toast is ready callback=",value);
}
```



4. Callback Hell

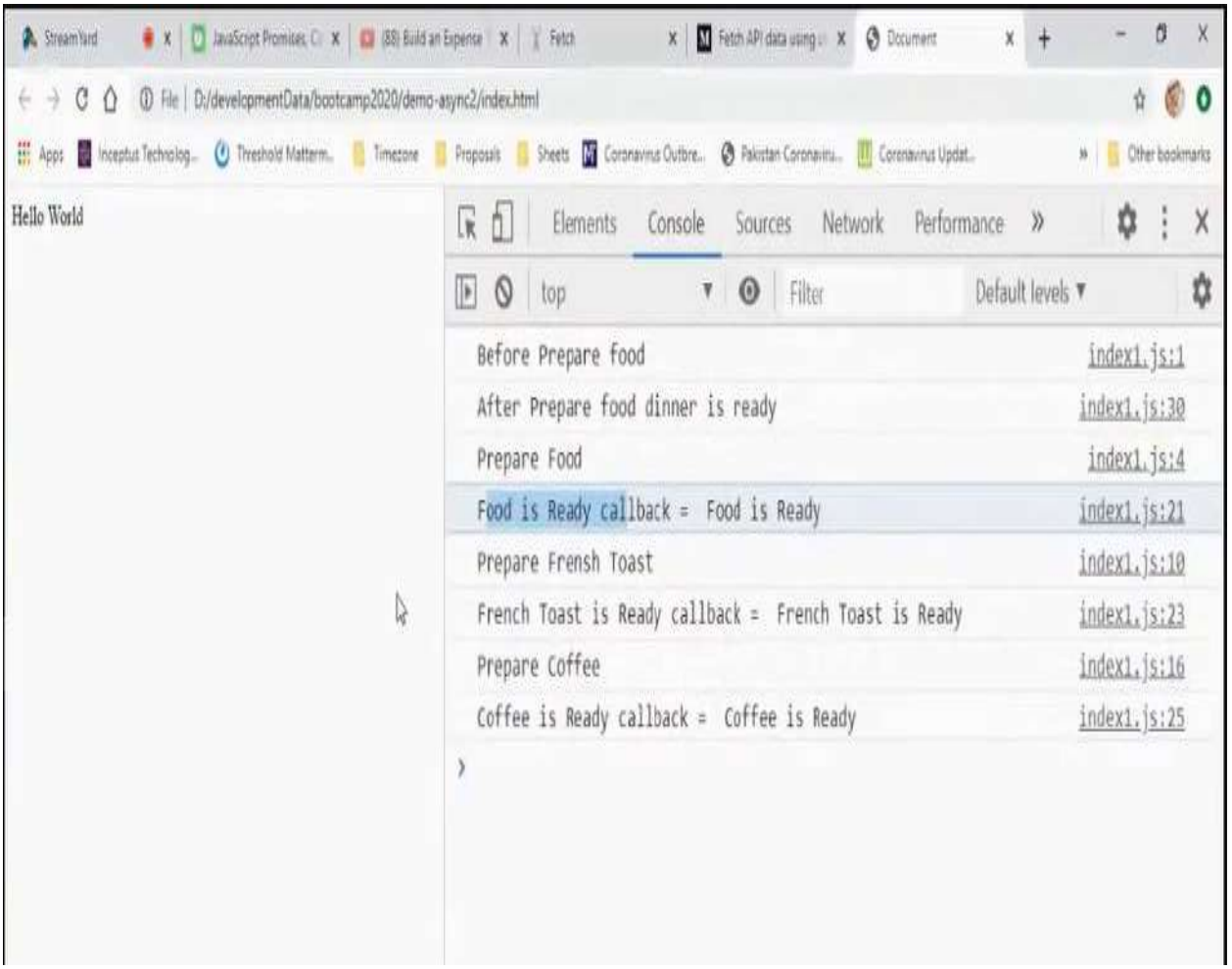
- Asynchronous JavaScript, or JavaScript that uses callbacks, is hard to get right intuitively.
- as a built-in callback can not handle the error and debugging becomes difficult, there is no feature to handle this problem
- Thereby it makes coding difficult for a daily basis long coding project consisting of more than 50 functions or so on.

Now let's consider our chef is going to cook food, French toast as we as coffee So it will result in a number of callbacks that will create lots of problem in coding

```
console.log("Before Prepare Food");
function prepareFood(callback){
setTimeout(()=>{
console.log("prepare Food");
callback("Food is Ready");
},10);
}
function prepareFrenchToast(callback){
setTimeout(()=>{
console.log("prepare french toast");
callback(" french toast is Ready");
},20);
}
function preparecoffee(callback){
setTimeout(()=>{
console.log(prepare coffee")
callback("coffee is ready")
},30);

  // taking call backs as a parameter there
prepareFood(function(value{
        console.log("Food is ready  callback=",value);
   prepareFrenchToast(function(value{

       console.log("french Toast is ready callback=",value);
      prepareCoffee(function(value{
         console.log("coffee is ready callback=",value);
     });

   });

 });

prepareFood(mycallback);

console.log ("After Prepare food dinner is ready");
```

Call back

Call back



Note: **in our daily life codes goes on increasing, hence thereby long results in many functions and several callbacks have to create, just consider a code of 10000 callback functions, how you are going to handle this when there is no feature to debug any error?**

5. Promise

Promises have been introduced in **ES6** (2015) to allow for more readable asynchronous code than is possible with callbacks. The main difference between callbacks and promises is that with callbacks you **tell the executing function** what to do when the asynchronous task completes, whereas with promises the executing function returns a special object to you (the promise) and then you **tell the promise** what to do when the asynchronous task completes.

5.2 Example

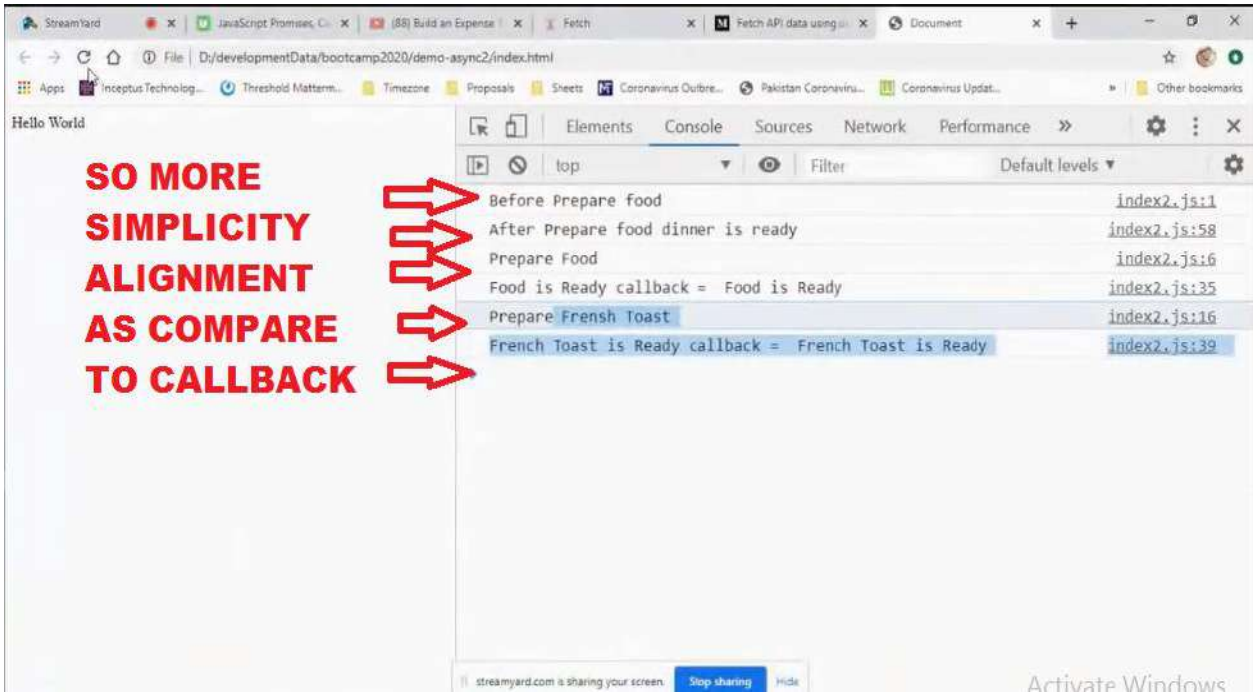
Now let's take an example we want our **chief to cook food, French toast, and coffee at different time slots then how we would resolve the problems of callback and cover it with promise; discuss and execute it in detail.**

Note

- **Since in callback, we face problems of nested if structure as the coding and functions call back increases but in the case of promises this problem is being solved by .then function that calls the function but not in a nested way and makes the coding easier as it has been clearly shown and discussed in code part :**

- Create another indx.js component and follow up these coding comments has been passed where needed

```
//Promise is a object and require a function that needs two values
function prepareFood(){
let promise="new promise(function (resolve, reject){
//all value of function reside in this function and this returns a promise
setTimeout(()=>{
    console.log("prepare Food");
    resolve("Food is Ready");
   },10);
});
  return promise;
//so we have converted this food into a promise
function prepareFrenchToast(){
let promise="new promise(function (resolve, reject){
setTimeout(()=>{
console.log("prepare frenchToast");
resolve(" french toast is Ready");
},20);
});
  return promise;
function preparecoffee(){
let promise="new promise(function (resolve, reject){
setTimeout(()=>{
console.log(prepare coffee")
resolve("coffee is ready")
},30);
});
  return Promise;
}
// we will call the prepare food that will return promise
      let  promise =prepare food();
     // console.log("promise= ",promise)
//promise.then is a function and it requires another function that accepts a value(and that
function will be apss out as a parameter of "resolve')

promise.then(function value){
console.log("Food is ready  callback=",value);
return prepareFrench/toast();
})
.then (function(frenchToast){
 console.log("french Toast is ready callback=",frenchToastValue);
return prepareCoffee();
})
.then(function(coffeValue){
 console.log("coffee is ready callback=",coffeValue);
  });


.catch (function(error){
console.log("Error=error");
```

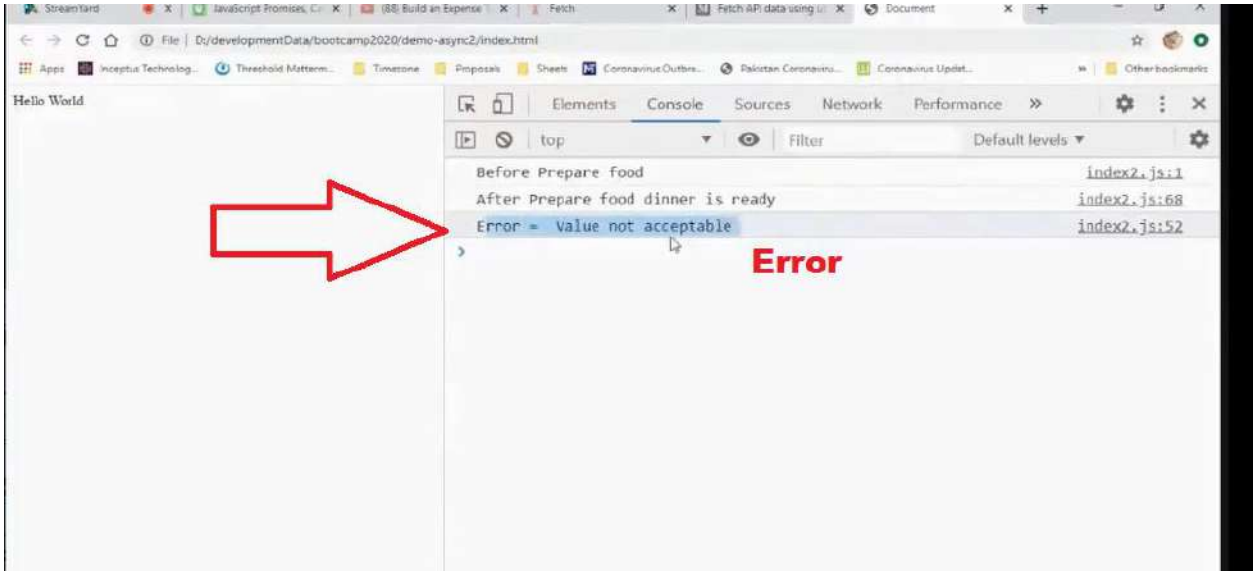


5.2. Example

Now let's take an example we want our chief to cook food, French toast, and coffee at different time slots then how we would resolve the problems of callback and cover it with promise discuss and execute and if the promise does not resolve it then how it will call out reject function and show any error. it in detail.

- **IN SOME CASE when the resolve function is not being executed due to some error then we call the reject function; the condition does not fulfill then it shows an error**

```
console.log('before prepare food");

function prepareFood(data){\
  let promise =new promise(function(resolve,reject){
    (()=>{
       if (data !==2){
         console.log(prepare Food");
            resolve{"Food is ready");
        }
        else{
           reject("Value not acceptable")
        }

      }.10);
    });
return promise;
function prepareFrenchToast(){
let promise="new promise(function (resolve, reject){
setTimeout(()=>{
console.log("prepare frenchToast");
resolve(" french toast is Ready");
},20);
});
  return promise;
function preparecoffee(){
let promise="new promise(function (resolve, reject){
setTimeout(()=>{
console.log(prepare coffee")
resolve("coffee is ready")
},30);
});
  return Promise;
}
      let  promise =prepare food(2);


promise.then(function value){
console.log("Food is ready  callback=",value);
return prepareFrench/toast();
})
.then (function(frenchToast){
 console.log("french Toast is ready callback=",frenchToastValue);
return prepareCoffee();
})
.then (function(coffeValue){
 console.log("coffee is ready callback=",coffeValue);
  });

.catch (function(error){
console.log("Error=" ,error);
```



- When a condition becomes false promise provides built-in functionality how to handle the error; when the wrong value or any resolve value becomes wrong then we call the catch function directly and all chain of execution of promises breakdown and it shows an error.

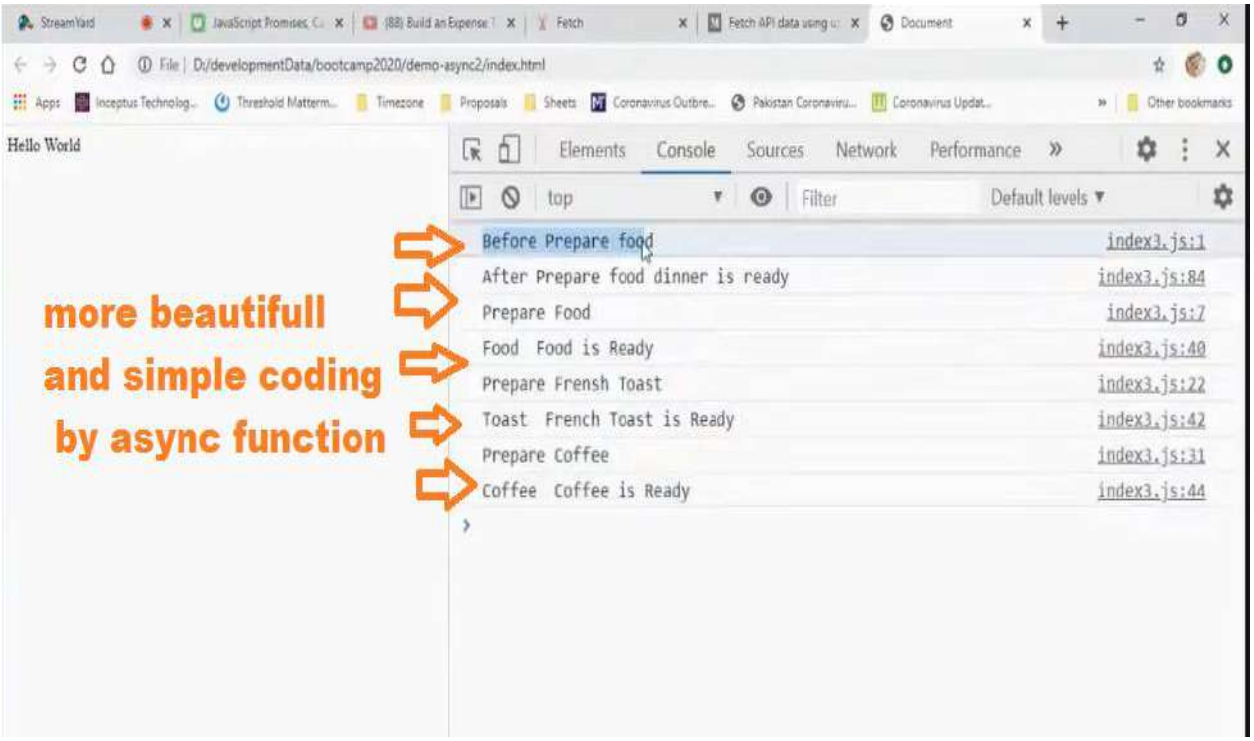
6. Aynchronous function Enhance Code Beauty And Simplicity than Promises:

- In an async way the promise would remain the same it will call all function as in the promise
- Due to the Await function once the prepared food is called it will not hang a function, execution goes to the background when we call synch start process
- instead of" .then" in promise here, we are using await (means the food become ready it will then move towards the next function of french toast and coffee)

6.1. Example

Now let's take an example we want our chief to cook food, French toast, and coffee at different time slots then how async function will help to carry out all events

```
console.log('before prepare food");

function prepareFood(data){
  let promise =new promise(function(resolve,reject){
    (()=>{
      if (data !==2){
        console.log(prepare Food");
          resolve{"Food is ready");
       }
       else{
          reject("Value not acceptable")
       }

     }.10);
    });
return promise;
function prepareFrenchToast(){
let promise="new promise(function (resolve, reject){
setTimeout(()=>{
console.log("prepare frenchToast");
resolve(" french toast is Ready");
},20);
});
  return promise;
function preparecoffee(){
let promise="new promise(function (resolve, reject){
setTimeout(()=>{
console.log(prepare coffee")
resolve("coffee is ready")
},30);
});
  return Promise;
}
async function startProcess(){

let FoodValue=await prepareFood();
console.log ("food",foodvalue);
let FrenchtoastValue=await frenchToastFood();
console.log ("toast",prepareFrenchToast();
let coffeeValue=await preparecoffee();
console.log ("coffee",prepareCoffeeValue);
}
start process();
console.log("after prepare food dinner is ready");
```



That's it how async function upholds callback and promise function

Session 8: Get Data in JavaScript and React

How To Fetch Data

1.1. GetData

create a folder i.e" DEMO-LOCAL-FETCH" and create a component index.js, link it with html as in index.html component:

Index.html

```
<!DOCTYPE html>
<html lang="en">
<haed>
    <title>Document</title>
    <script  src="index.js"></script>
 </head>
   <body>
    Hello World
   </body>
  </html>
```

- Now in index.js for network call use any dummy API i.e jasonplaceholder.typicod.com"

This will gives you data when you call which you can use as a server.

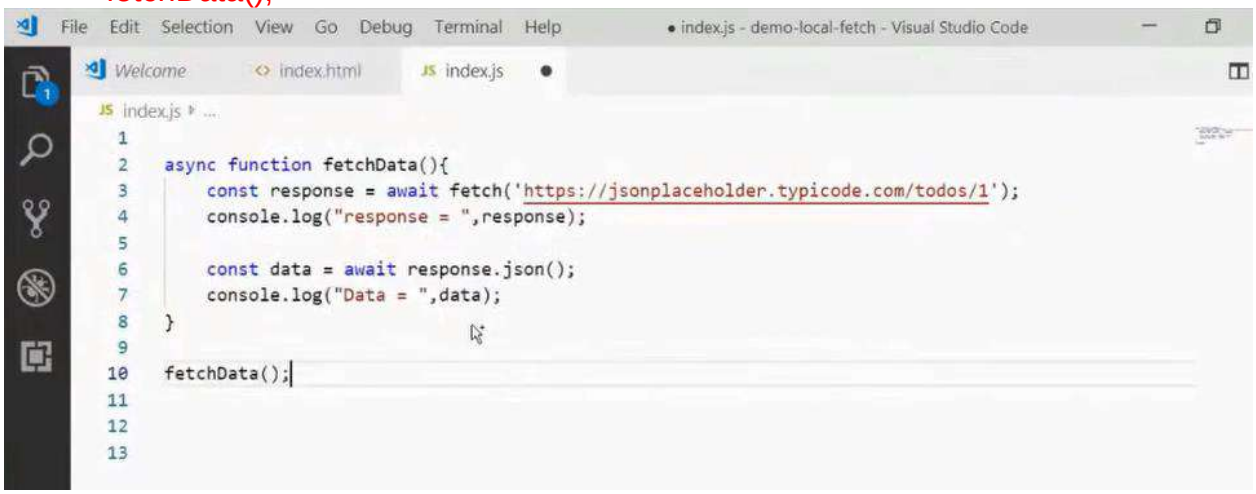
Declare async function fetchData()

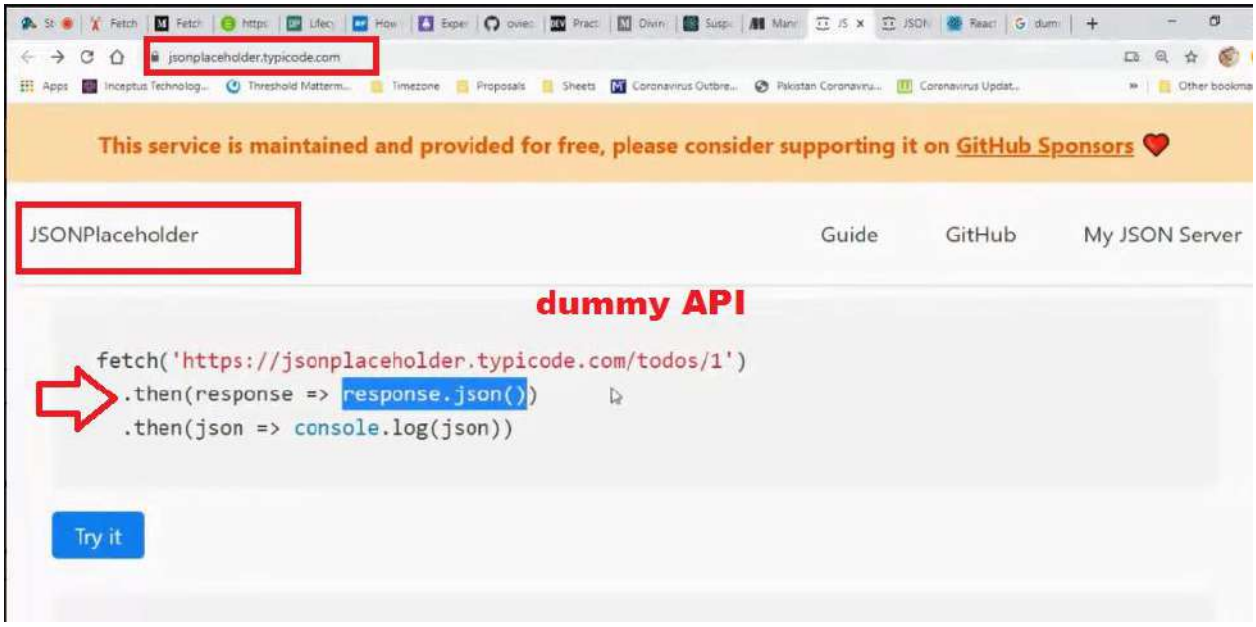
- call data in jason format

  const data=response.json();

  console.log ("Data=",data);
- call async function as

  fetchData();

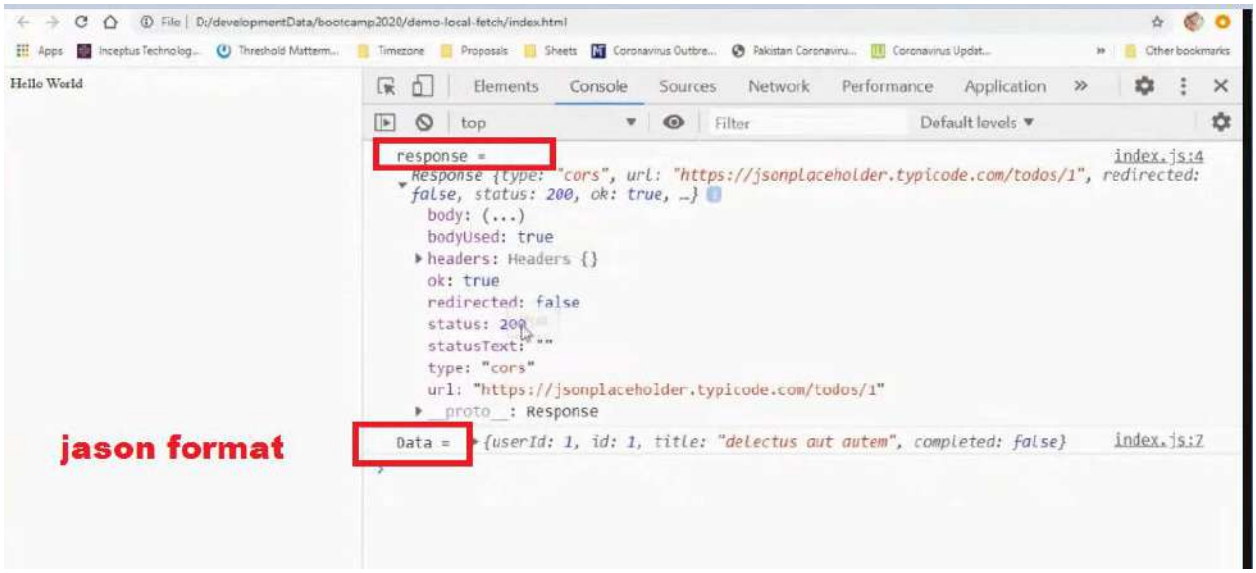




index.js

```
async function fetchData(){
const response = await fetch('https://jasonplaceholder.typicod.com')

console.log("response=",response);
//call data in jason format
const data=response.json();
console.log ("Data=",data);
}

fetchData();
```

- in first response, it creates a response that contains multiple properties
- then it was converted this into Jason formate as shown



NOTE Since fetch data is a type of **getData** that means data will be in the plain text there would be no hide of information everything you call would be displayed on the URL

Now if you want to hide data on the call then you will use post data as shown

1.2. Post Data

- call fetch,API and second argument that includes the JASON object
- Jason object includes

method: "POST"

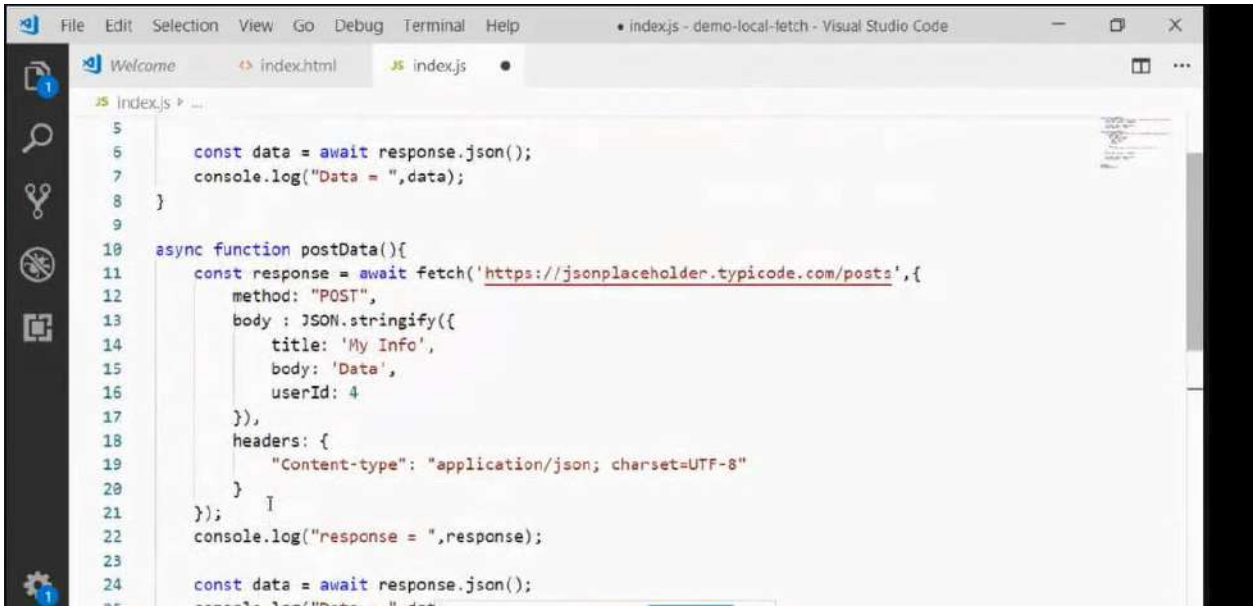
body : JASON.stringify({

title: 'My Info',

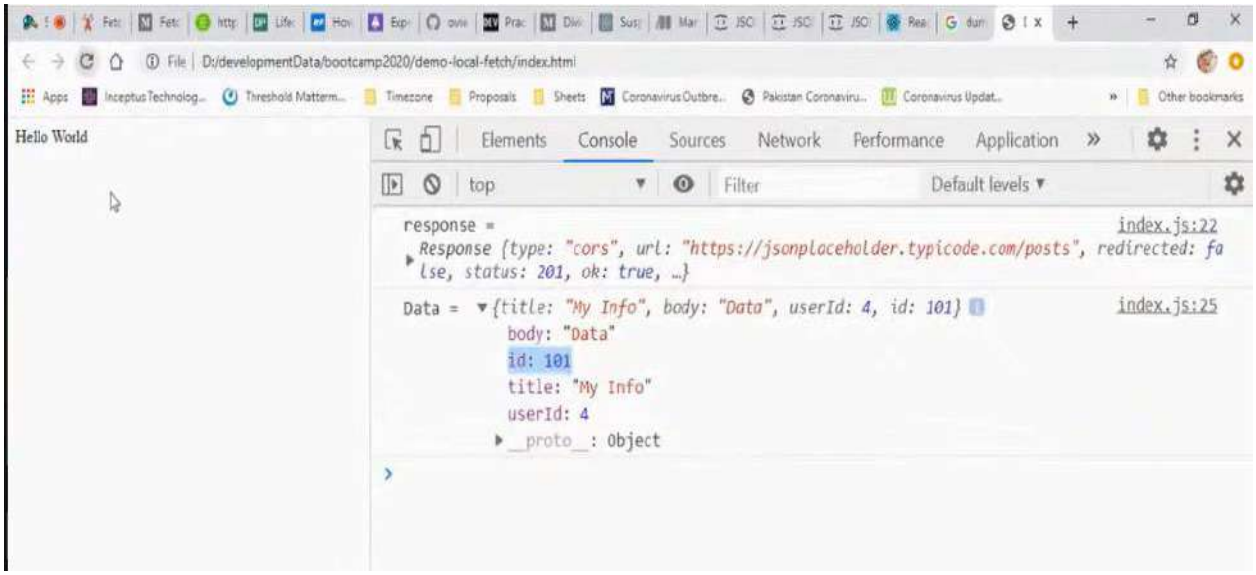
body: 'Data',

userID: 4

- Then it will call header(as a Jason Application or request)
- call postdata

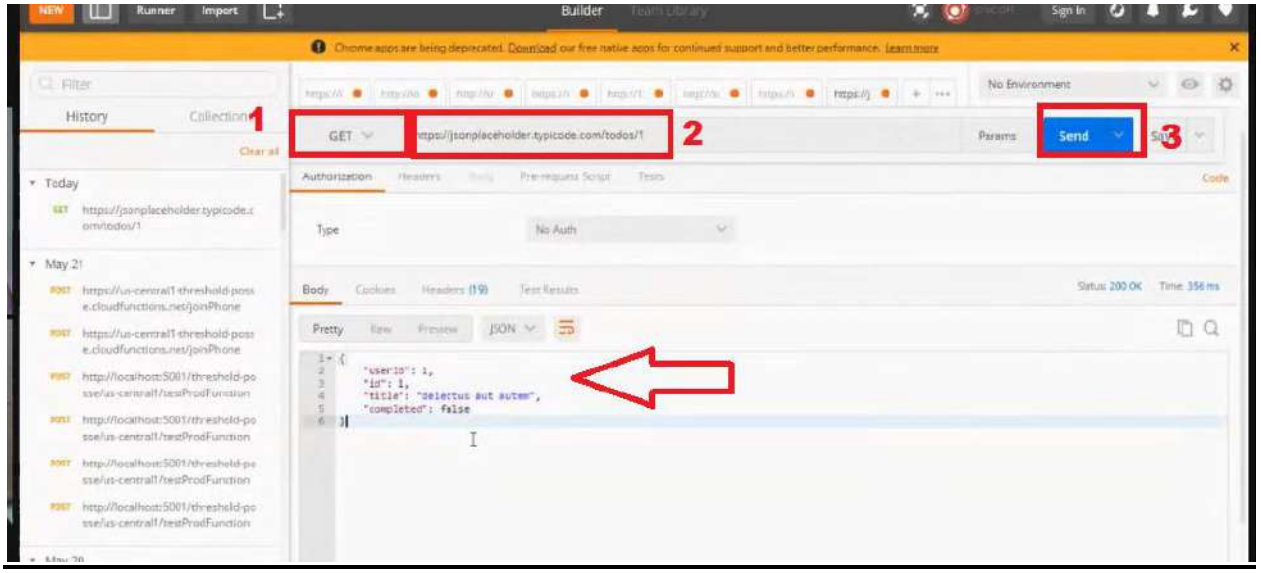



```
async function postData(){
const response = await fetch("https://jasonplaceholder.typicod.com")
method: "POST"
body : JASON.stringify({
title: 'My Info',
body :'Data',
userID:  4
})
headers:{
        "content-type": "application/jason; charset=UFT-8"
    });
const data=response.json();
console.log ("Data= ",data);
}
postData():
```



2. Postman

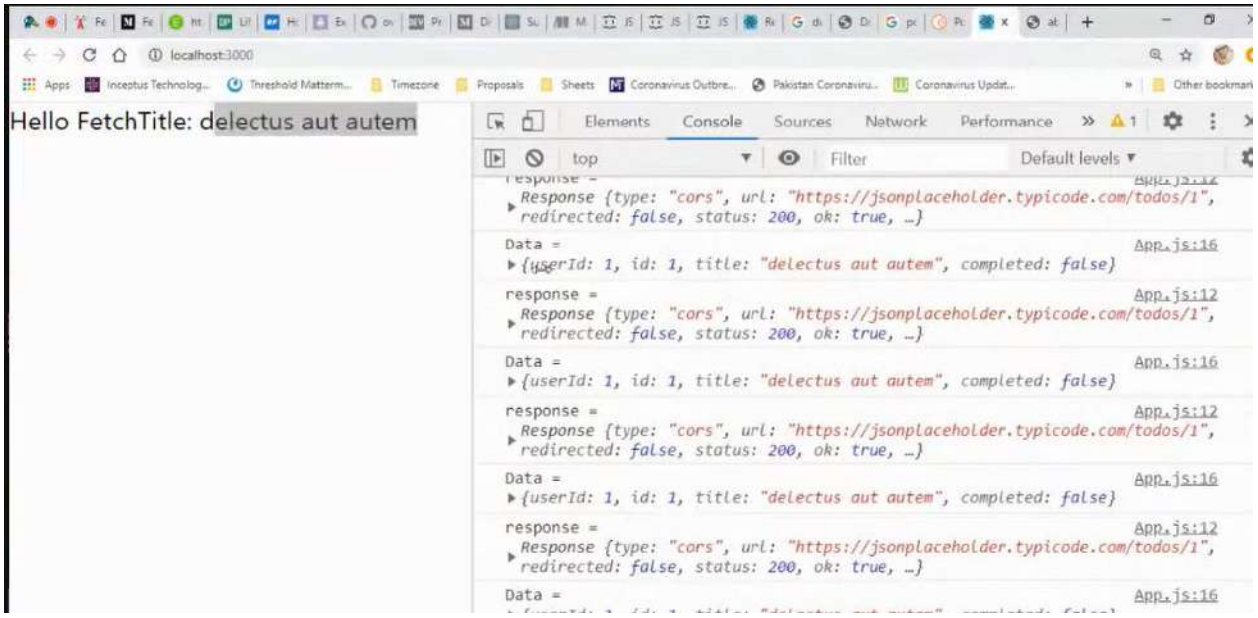
- Open Postman browser Extention
- just select method i.e post, get, etc and provide API and click to send
- Postman automatically verify API And data as shown



3. Fetch API Data Using UseEffect

- Create a react app as we did in our previous classes
- Create a folder and component of index.js in it and follow the given coding.
- call useEffect function and provide data and call toDo as property and to update and set data we need to call **SetTodo** function after we fetch data and convert it into Jason format, call fetch data and call API it will allow the server to work and display.

```
import React, (useeffect} from 'react
import ,'./Apps.css';
function App(){
const data = null;
function App(){
let data ={title: "waiting for data"};
const [todo, setTodo] = useState ({data});

useEffect ()=> {
asyn function fetchData(){
const response = await fetch('https://jasonplaceholder.typicod.com')
console.log("response = ",response);
let data2 = (await response.json())
setTodo(data2);
console.log ("Data= ",data);
}
fetchData();
});
return(
<div>
Hello Fetch
<span>Title: {data.title}</span>
</div>
```



3.1 Component With Multiple Properties

- Create a react app as we did in our previous classes
- Create a folder and component of index.js in it and follow the given coding.
- Call useEffect function and provide data and call toDo as property and to update and set data we need to call **SetTodo** function after we fetch data and convert it into Jason format, call fetch data and call API it will allow the server to work and display.
- Now to re-render the data as in 2 condition

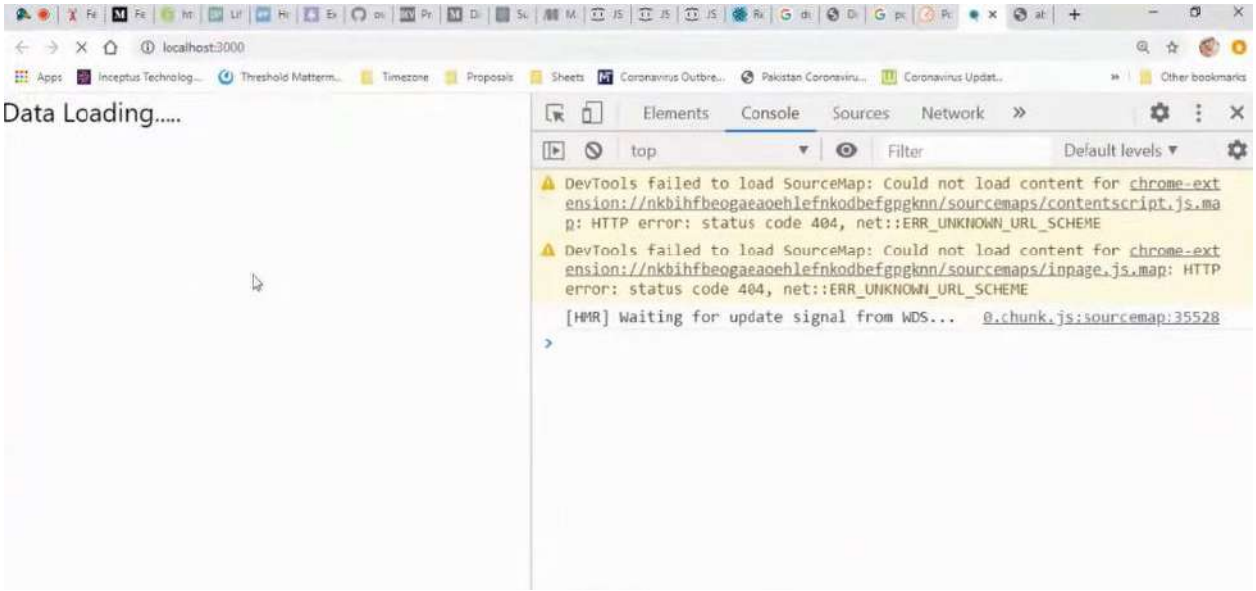
  Data Loading.....

  Hello Fetch…..
- We will declare setFetching(true); in async function so when the condition will be true then it will display "Data Loading"
  - if(isFetching){
  - return Data Loading.....

- if condition is false then it will again re-render API function and function will move to
  - return(
  -
  - Hello Fetch
  - Title: {data.title}
  -
- it will display "Hello fetch"

- return(

```
import React, (useeffect} from 'react

import ,'./Apps.css';

function App(){

const data = null;

function App(){

let data ={title: "waiting for data"};

const [todo.setTodo] = useState (data);

const [isData, setData]=useState(false);

const[isFetching, setFetcing] =usestate(false);


useEffect ()=> {

asyn function fetchData(){

setFetching(true);

const response = await fetch("https://jasonplaceholder.typicod.com")

console.log("response =,response);

let data2 = (await response.json())

setTodo(data2);

setFetching(false);

console.log ("Data=",data);

}

fetchData();

}  [isData]);

});

if(isFetching){
return <div>Data Loading.....</div>
}
return(
<div>
Hello Fetch
<span>Title: {data.title}</span>
```


localhost:3000
Data Loading.....
Elements
Console
Sources
Network
top
Filter
Default levels
DevTools failed to load SourceMap: Could not load content for chrome-extension://nkbihfbeogaeaoehlefnkodbefgpgknn/sourcemaps/contentscript.js.map: HTTP error: status code 404, net::ERR_UNKNOWN_URL_SCHEME
DevTools failed to load SourceMap: Could not load content for chrome-extension://nkbihfbeogaeaoehlefnkodbefgpgknn/sourcemaps/inpage.js.map: HTTP error: status code 404, net::ERR_UNKNOWN_URL_SCHEME
[HMR] Waiting for update signal from WDS...
0.chunk.js:sourcemap:35528

Session 9: Material UI & Charts.js

1. **<u>Material UI Decreases CSS Coding</u>**

- **Material UI decreases CSS coding**

Material-UI, the React component library based on Google Material Design, allows for faster and easier stylized web development. With basic React framework familiarity, you can build a deliciously material app with Material-UI, list of material UI is given as ;

Components

Layout

Box

Container

Grid

Grid List

Hidden

Inputs

Button

Button Group

Checkbox

Floating Action Button

Date / Time

Radio

Select

Slider

Navigation

Bottom Navigation

Breadcrumbs

Drawer

Link

Menu

Stepper

Tabs

Surfaces

App Bar

Paper

Card

Accordion

Feedback

Progress

Component API

- Accordion
- Accordion Actions
- Accordion Details
- Accordion Summary
- Alert
- Alert Title
- App Bar
- Autocomplete
- Avatar
- Avatar Group
- Backdrop
- Badge
- Bottom Navigation
- Bottom Navigation Action
- Breadcrumbs
- Button
- Button Base
- Button Group
- Card
- Card Action Area
- Card Actions
- Card Content
- Card Header
- Card Media
- Checkbox
- Chip
- Circular Progress
- Click Away Listener
- Collapse
- Container
- Css Baseline

Material-UI is available as an npm package.

Note:

You can install material ui source code from the following website

https://material-ui.com/getting-started/installation/

2. UI Material Needed For Installation

npm

To install and save in your package.json dependencies, run:

```
// with npm

npm install @material-ui/core
```

```
// with yarn

yarn add @material-ui/core
```

Roboto Font

```
<link rel="stylesheet"
href="https://fonts.googleapis.com/css?family=Roboto:300,400,500,700&display=swap" />
```

Font Icons

In order to use the font Icon component, you must first add the material icon font.

```
<link rel="stylesheet" href="https://fonts.googleapis.com/icon?family=Material+Icons" />
```

SVG Icons

In order to use prebuilt SVG Material icons, such as those found in the icon demos you must first install the @material-ui.com package:

```
// with npm

npm install @material-ui/icons



// with yarn

yarn add @material-ui/icons
```

Button

Buttons allow users to take actions and make choices, UI
Button communicates actions that users can take. They are typically placed throughout your UI, in places like:

- Dialogs
- Modal windows
- Forms
- Cards
- Toolbars

Contained Buttons

These are high-emphasis, distinguished by their use of elevation and fill. They contain actions that are primary to your app.

```
<Button variant="contained">Default</Button>
<Button variant="contained" color="primary">
  Primary
</Button>
<Button variant="contained" color="secondary">
  Secondary
</Button>
<Button variant="contained" disabled>
  Disabled
</Button>
<Button variant="contained" color="primary" href="#contained-buttons">
  Link
</Button>
```

You can remove the elevation with the disableElevation prop.

DISABLE ELEVATION

```
<Button variant="contained" color="primary" disableElevation>
  Disable elevation
</Button>
```

Text Buttons

Text buttons are typically used for less-pronounced actions, including those located:

- In dialogs
- In cards

In cards, text buttons help maintain an emphasis on card content.

```
<Button>Default</Button>
<Button color="primary">Primary</Button>
<Button color="secondary">Secondary</Button>
<Button disabled>Disabled</Button>
<Button href="#text-buttons" color="primary">
  Link
</Button>
```

Outlined Buttons

These are medium-emphasis buttons. They contain actions that are important but aren’t the primary action in an app.

Outlined buttons are also a lower emphasis alternative to contained buttons or a higher emphasis alternative to text buttons.

```
<Button variant="outlined">Default</Button>
<Button variant="outlined" color="primary">
  Primary
</Button>
<Button variant="outlined" color="secondary">
  Secondary
</Button>
<Button variant="outlined" disabled>
  Disabled
</Button>
<Button variant="outlined" color="primary" href="#outlined-buttons">
  Link
</Button>
```

Handling clicks

All components accept an onClick handler that is applied to the root DOM element.

```
<Button onClick={() => { alert('clicked') }}>Click me</Button>
```

Note that the documentation avoids mentioning native props (there are a lot) in the API section of the components.

Upload button

UPLOAD

Sizes

- ✓ Fancy larger or smaller buttons? Use the size property.
- ✓ SMALLMEDIUMLARGE
- ✓ SMALLMEDIUMLARGE
- ✓ SMALLMEDIUMLARGE

Buttons with icons and label

Sometimes you might want to have icons for a certain button to enhance the UX of the application as we recognize logos more easily than plain text. For example, if you have a delete button you can label it with a dustbin icon.

DELETESENDsendUPLOADTALKSAVESAVE

Icon Buttons

Icon buttons are commonly found in app bars and toolbars.

Icons are also appropriate for toggle buttons that allow a single choice to be selected or deselected, such as adding or removing a star to an item.

3. Creating Button Using UI Materials

- npm install
- UI material install

```
- // with npm
- npm install @material-ui/core
-
- // with yarn
- yarn add @material-ui/core
```

- font icon must be installed In index.html

```
<link rel="stylesheet" href="https://fonts.googleapis.com/icon?family=Material+Icons" />

// with npm

npm install @material-ui/icons



// with yarn

yarn add @material-ui/icons
```

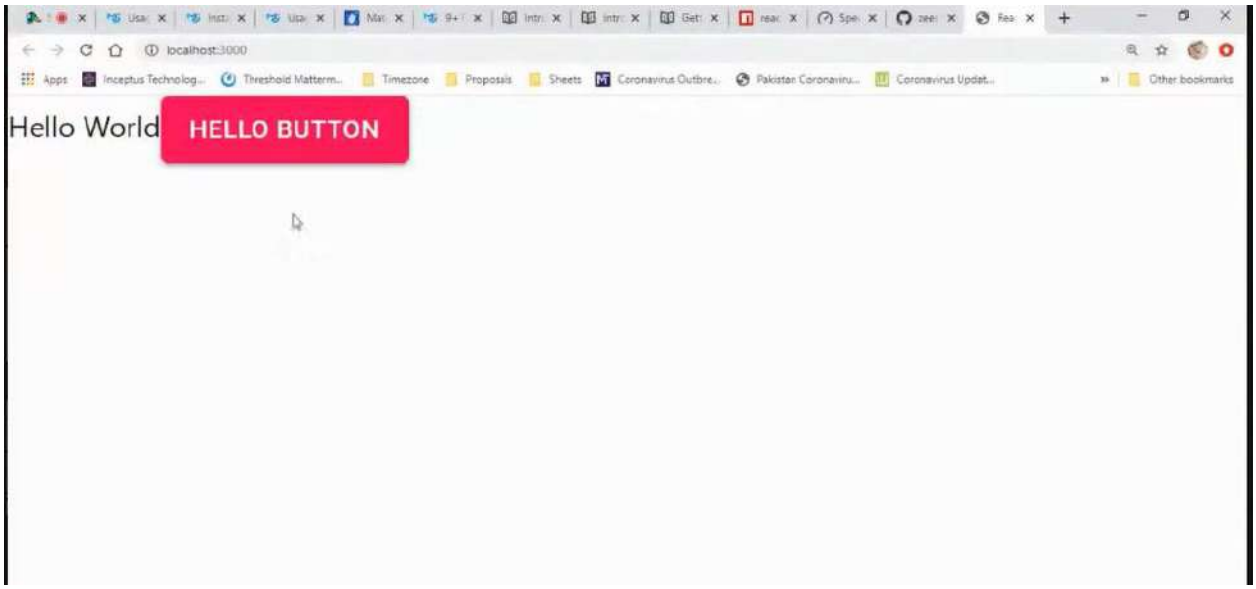
- import {Button} from 'material-ui/core' in App.js

- you can add button variety as primary, secondary, etc as well as in place of contained you can use outline, text whatever you want to look for your button; doing so will decrease CSS coding and implement all CSS style as its suited in App.js

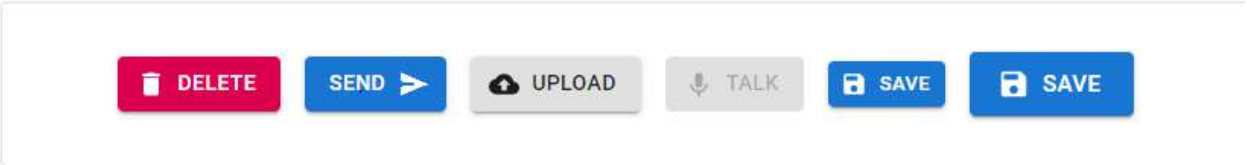
-

App .js

```
import React from 'react';
import './App.css';

import {Button} from 'material-ui/core'

function App()
reurn(
<div>
   Hello World
<Button variant ="contained"  color="secondary">Hello Button</Button>
</div>
);

}
```

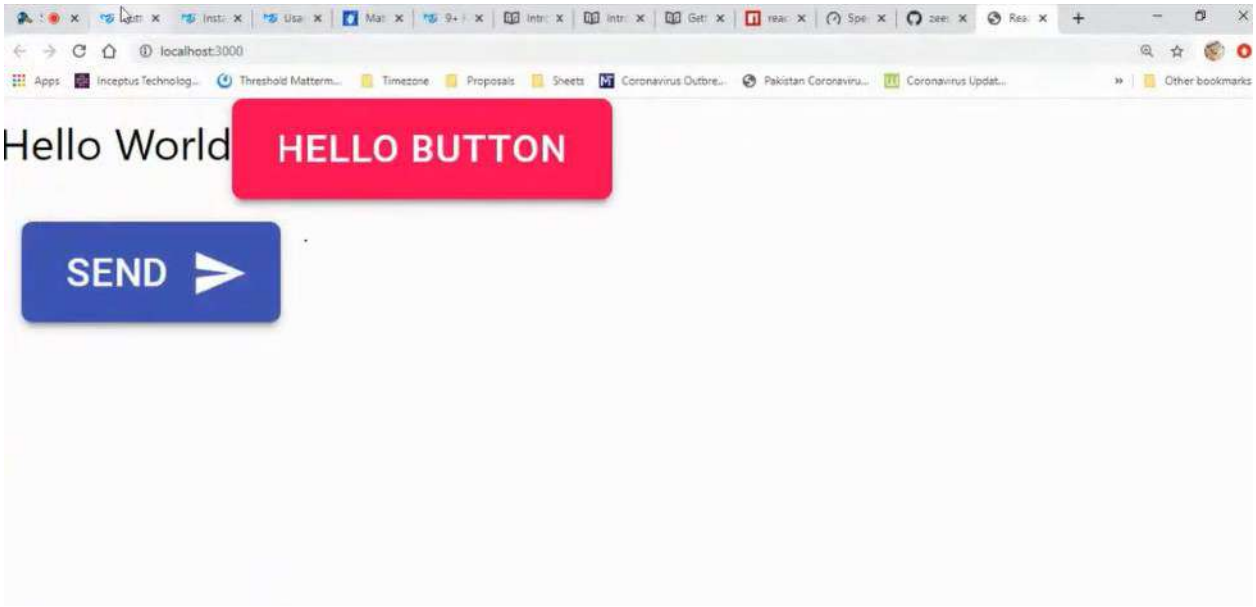


- **Let's create a send button now, make sure you have installed all UI material and npm, font, an icon in index.html ;**



In app.js follow this coding for Hello as well as the Send button

```
import React from 'react';
import './App.css';

import {Button} from 'material-ui/core'

function App()
reurn(
<div>
  Hello World
<Button variant ="container" color="secondary">Hello Button</Button>
<div>

    {/* This Button uses a Font Icon, see the installation instructions in the Icon component docs. */}
    <Button
      variant="contained"
      color="primary"
      className={classes.button}
      endIcon={<Icon>send</Icon>}
    >
      Send
    </Button>
</div>
</div>
);
}
```



4. Grid

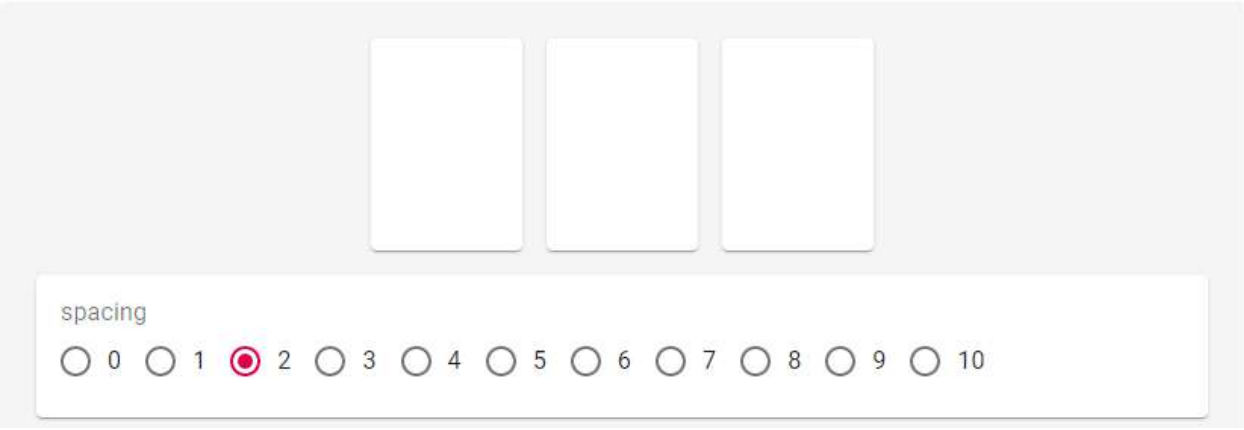
The Material Design responsive layout grid adapts to screen size and orientation, ensuring consistency across layouts.

The grid creates visual consistency between layouts while allowing flexibility across a wide variety of designs. Material Design's responsive UI is based on a 12-column grid layout.

Spacing

The responsive grid focuses on consistent spacing widths, rather than the column width. Material Design margins and columns follow an **8px** square baseline grid. The spacing property is an integer between 0 and 10 inclusive. By default, the spacing between two grid items follows a linear function: output(spacing) = spacing * 8px, e.g. spacing={2} creates a 16px wide gap.

This output transformation function can be customized using the theme.



Fluid grids

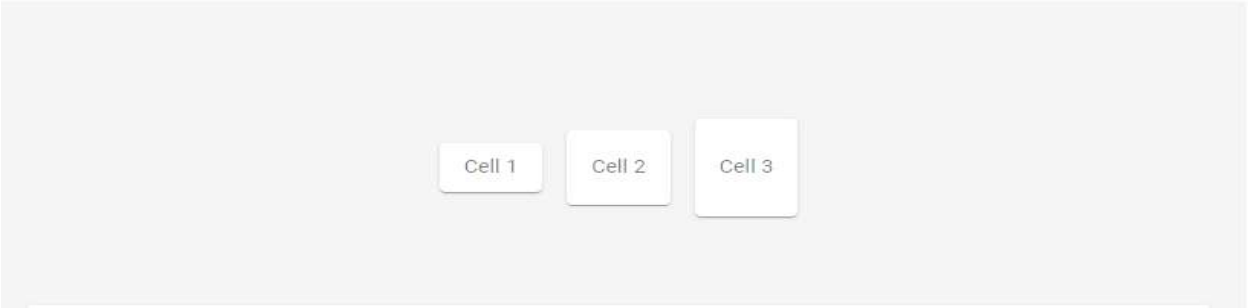
Fluid grids use columns that scale and resize content. A fluid grid's layout can use breakpoints to determine if the layout needs to change dramatically.

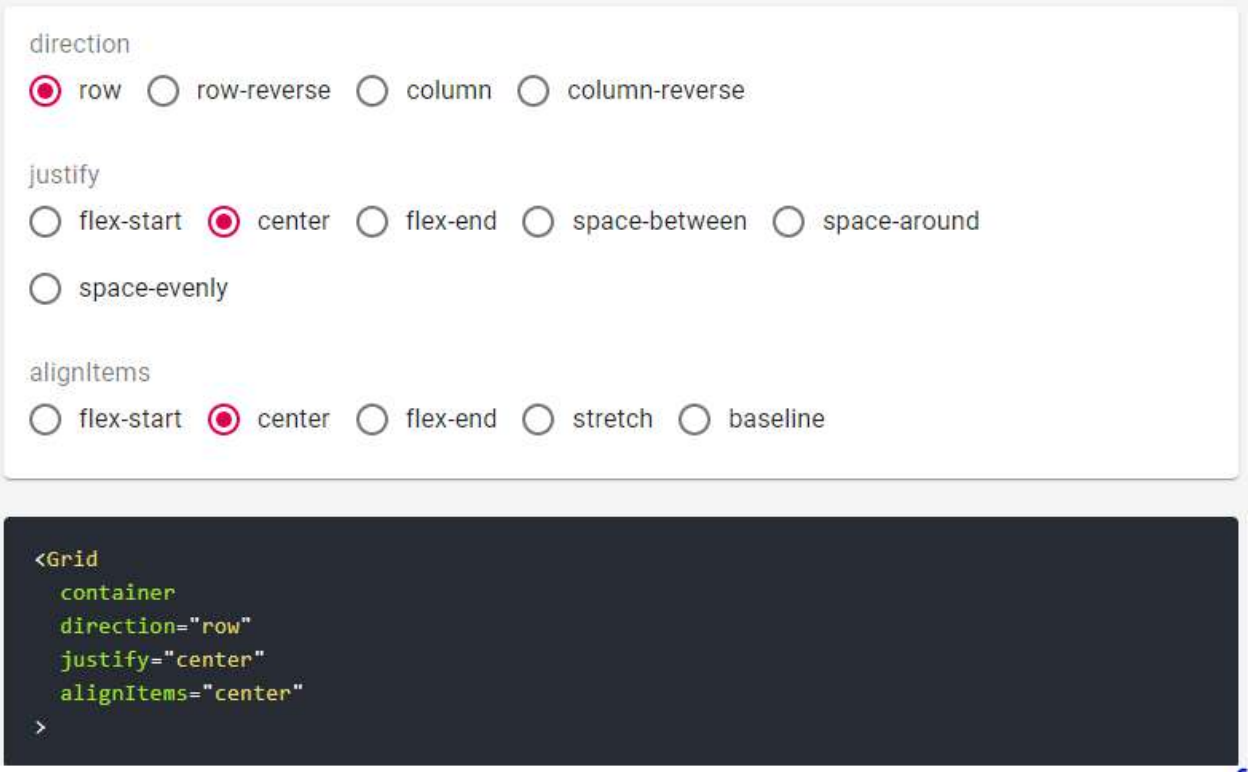
Basic grid

The column widths apply at all breakpoints (i.e. xs and up).

Grid with breakpoints

Some columns have multiple widths defined, causing the layout to change at the defined breakpoint.

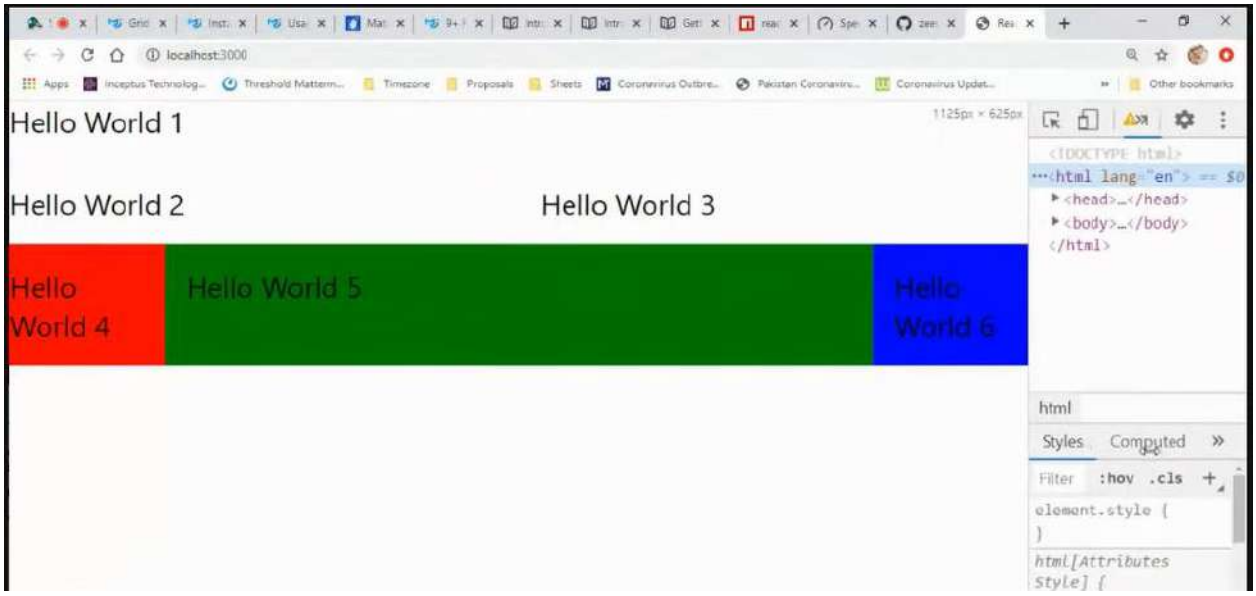




4.1. Basic Grid Creation

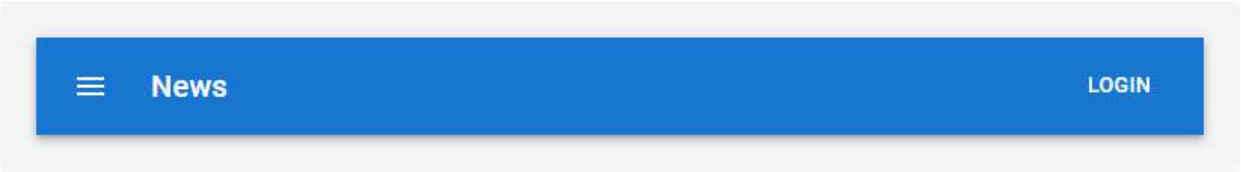
- create container type grid with spacing to properly show items with spaces in between them and inside grid provide columns or rows

```
import React  from 'react';
import './App.css';
import{ button.Icon, makeStyle, Grid} from '@material-ui/core'
const usestyles =makeStyle((theme)=>({
button:{
margin theme.spacing(1),
}));
function app(){
const classes =useStyle();
return(
<div>
<Grid container spacing={3}>
<Grid item xs={12}>
Hello World 1
</Grid>
<Grid item xs={6}>
Hello World 2
</Grid>
<Grid item xs={2} style ={{backgroundColor:"red"}}>
Hello World 4
</Grid>
<Grid item xs={8} style ={{backgroundColor:"green"}}>
Hello World 5
</Grid>
<Grid item xs={4} style ={{backgroundColor:"blue"}}>
Hello World 6
</Grid>
</div>

);
```



5. App Bar

- It provides content and actions related to the current screen. It's used for branding, screen titles, navigation, and actions.
- It can transform into a contextual action bar or be used as a navbar.
- pp Bar displays information and actions relating to the current screen.



- Now in App.js import "import{ button.Icon, makeStyle, Grid,IconButton,AppBar,Toolbar,Typography} from '@material-ui/core'
- Provide function with different classes as:

```
const useStyle  = makeStyles((theme) => ({
root : {
  flexGrow: 1,
},
title: {
  flexGrow: 1,
},
```

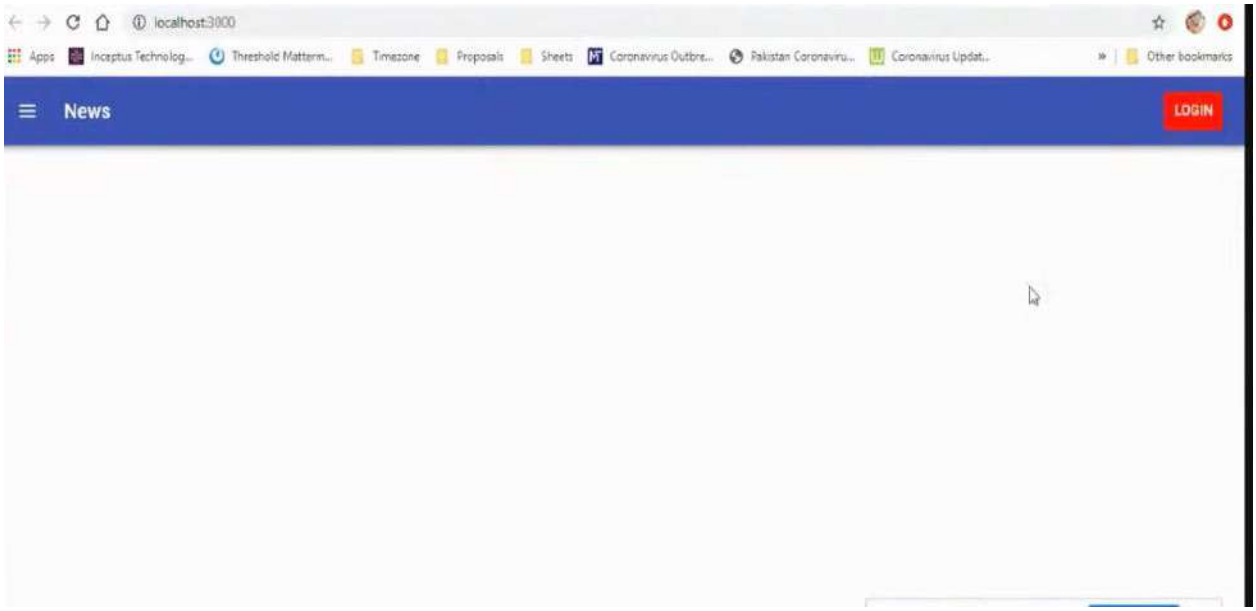
- For red login button :

```
testButton: {
   backgroundColor : "red"

}
```

- Also in function app declare classes and different properties as;

```
function App(){
const classess = useStyle();
return(
<div className = { class.root }>
 <AppBar position ="static">
  <Toolbar>
      <iconButton edge ="start" className ={classes.menuButton} color = "inherit"
aria-label="


<MenuIcon/>
<typography variant ="h6" className = {classes.title}>
News </Typography>
<Button color ="inherit" className = {classes.testButton>Login</Button>
</Toolbar>
</AppBar>
</div>);
```

```
import React from 'react';
import './App.css';
import{ button.Icon, makeStyle, Grid,IconButton,AppBar,Toolbar,Typography} from
'@material-ui/core'

const useStyle =makeStyles((theme)=>({
root:{
  flexGrow:1,
},
title: {
  flexGrow: 1,
},
testButton,:{
   backgroundColor :"red"
}

}));

function App(){
const classess = useStyle();
return(
<div className={class.root}>
 <AppBar position ="static">
  <Toolbar>
      <iconButton edge ="start" className ={classes.menuButton}color = "inherit" aria-
label="


<MenuIcon/>
<typography variant ="h6" className={classes.title}>
News'</Typography>
<Button color ="inherit" className ={classes.testButton>Login</Button>
</Toolbar>
</AppBar>
</div>

);
```



6. Bar chart

A **bar chart** or **bar** graph is a **chart** or graph that presents categorical data with rectangular **bars** with heights or lengths proportional to the values that they represent. The **bars** can be plotted vertically or horizontally. A vertical **bar chart** is sometimes called a **column chart**.

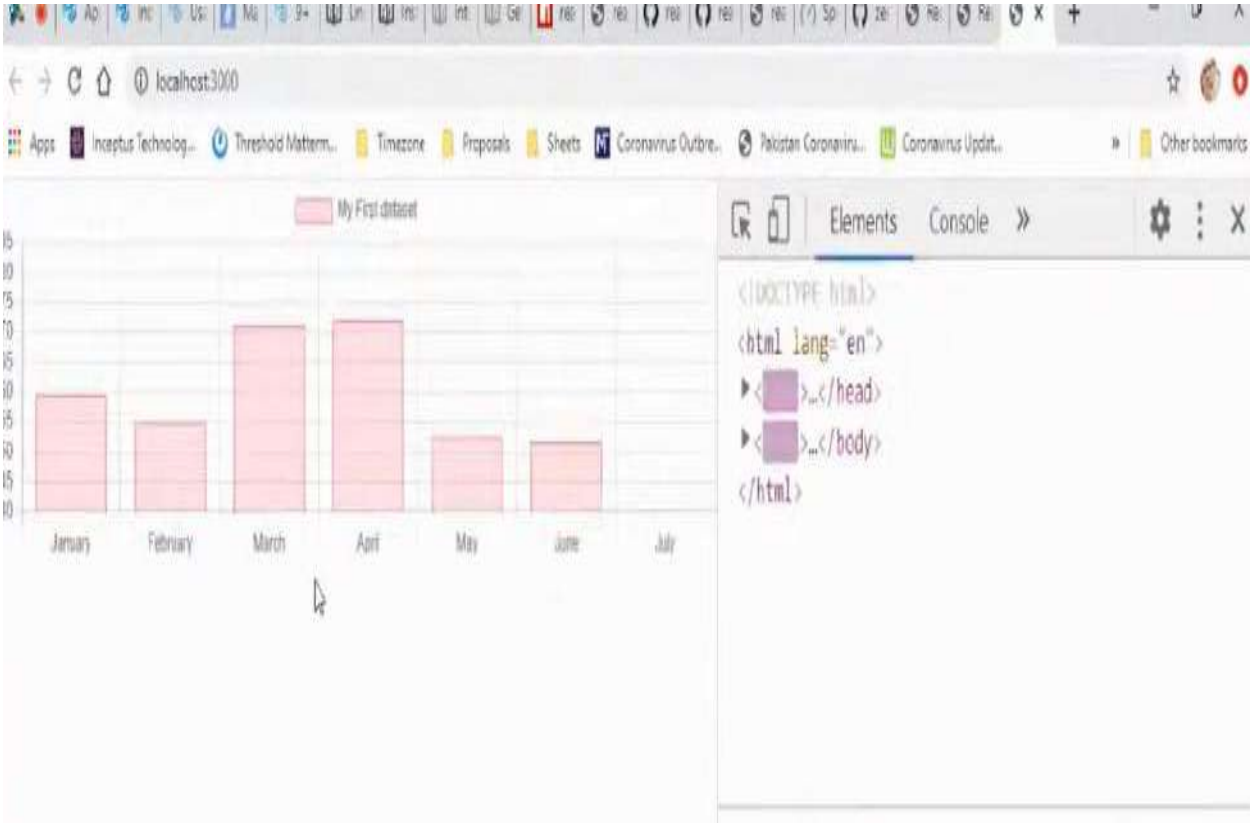
- Go to App.js, set seven labels

'january', 'Fevruary', 'March', 'April', 'May', 'june', 'july']

- the dataset that has the following properties array

```
  database:[
{
        label: 'My First dataset',
        backgroundColor:'rgba(255,99,132,0,2)',
        borderWidth:1,
        hoverBackgroundColor:'rgba(255,99,132,0,4)',
        hoverBorderColor: rgba(255,99,132,1)',
        data:[65,59,80,81,56,55,40]
        }
```

```
import React from 'react';
import {Bar} from 'react-chartjs-2';
const data =['january', 'Fevruary', 'March', 'April', 'May', 'june', 'july']
database:[
{
        label: 'My First dataset',
        backgroundColor:'rgba(255,99,132,0,2)',
        borderWidth:1,
        hoverBackgroundColor:'rgba(255,99,132,0,4)',
        hoverBorderColor : rgba(255,99,132,1)',
        data:[65,59,80,81,56,55,40]
        }
    ]
 }
;

  export default React.createClass({
    displayName: 'BarExample",

    render(){
   return(){
   <div>
    <h2>Bar Example (custom size)</h2>
   <Bar
     data-{data]
     width={100}
     height={200}
      options={(
          maintainAspectRatio: false
     )}
   />

  </div>
```


localhost:3000
Apps
Inceptus Technolog...
Threshold Matterm...
Timezone
Proposals
Sheets
Coronavirus Outbre...
Pakistan Coronaviru...
Coronavirus Updat...
Other bookmarks
My First dataset
January
February
March
April
May
June
July
Elements
Console
<!DOCTYPE html>
<html lang="en">
<head></head>
<body></body>
</html>

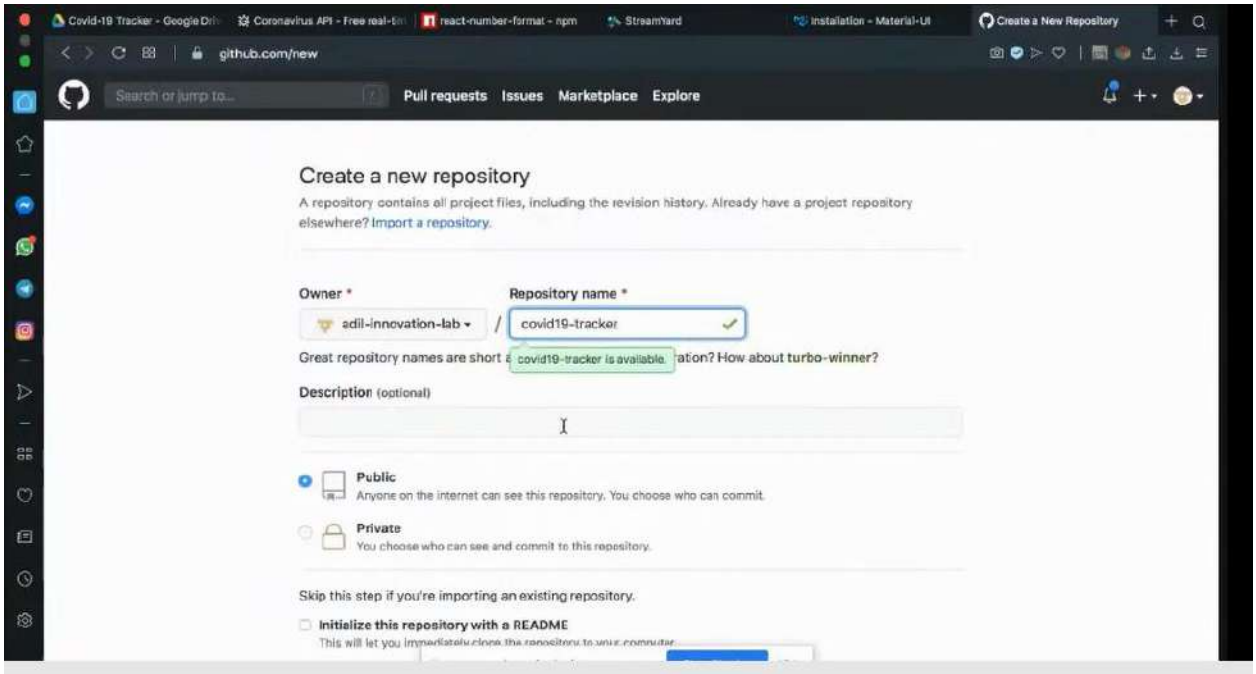
Session 10: Material UI & Charts.js

1. Development Of COVID 19 Tracker App

- set up your project with the following commands for creating Coviid-19 Tracker App
- install all API materials as well as install npm

```
Success! Created covid19-tracker at /Users/adil/Documents/GitHub/covid19-t
racker
Inside that directory, you can run several commands:

  yarn start
    Starts the development server.

  yarn build
    Bundles the app into static files for production.

  yarn test
    Starts the test runner.

  yarn eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd covid19-tracker
  yarn start
```

- Go to GitHub.com
- create a new repository of name Covid 19 Tracker as we did in previous classes
- publish it



- now in App.js import Header component
- Create a component Header.js and follow up this coding

```
import React from 'react';
import { makeStyles } from '@material-ui/core/styles';
import AppBar from '@material-ui/core/AppBar';
import Toolbar from '@material-ui/core/Toolbar';
import Typography from '@material-ui/core/Typography';

const useStyles = makeStyles((theme) => ({
  root: {
    flexGrow: 1,
  },
  menuButton: {
    marginRight: theme.spacing(2),
  },
  title: {
    flexGrow: 1,
  },
}));

export default function ButtonAppBar() {
  const classes = useStyles();

  return (
    <div className={classes.root}>
      <AppBar position="static">
        <Toolbar>
          <IconButton edge="start" className={classes.menuButton} color="inherit" aria-label="menu">
            <MenuIcon />
          </IconButton>
          <Typography variant="h6" className={classes.title}>
            News
          </Typography>
          <Button color="inherit">Login</Button>
        </Toolbar>
      </AppBar>
    </div>
  );
}
```

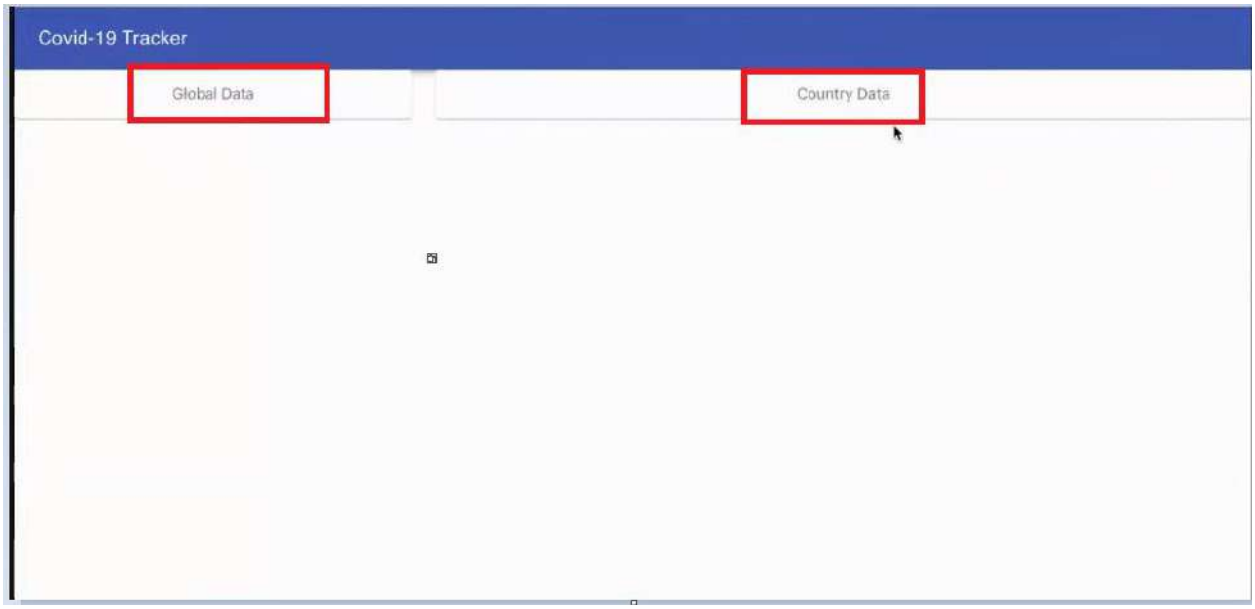
- so we have a simple and nice App bar with the heading “Covid-19 Tracker”;

- Now create another component “MainGrid” and import this into the App.js component
- In MainGrid.jss create two grids with 4 and 8 columns respectively
- In one grid will be globe data and in other grid will be country data as :

```
App.js
import React from 'react';
import logo from './logo.svg';
import ./App.css';
import Header from ./components/Header'
import  MianGrid from ./components/MainGrid'

function App(){
return (
  <div>
    Hello World
    <Header/>
   <MainGrid/>
   </div>
  );

}

export default App;
```

MainGrid.jss

```
import React from 'react';
import { makeStyles } from '@material-ui/core/styles';
import Paper from '@material-ui/core/Paper';
import Grid from '@material-ui/core/Grid';

const useStyles = makeStyles((theme) => ({
  root: {
    flexGrow: 1,
  },
  paper: {
    padding: theme.spacing(2),
    textAlign: 'center',
    color: theme.palette.text.secondary,
  },
}));

export default function MainoGrid() {
  const classes = useStyles();

  return (
    <div className={classes.root}>
      <Grid container spacing={3}>
        <Grid item xs  {4}>
          <Paper className= {classes.paper}>
             GlobalData                                    ***[<GlobalData/>]

          </Paper>
        </Grid>
        <Grid item xs{8}>
          <Paper className={classes.paper}>
            Country data
         </Paper>

      </Grid>

    </div>
  );
}
```

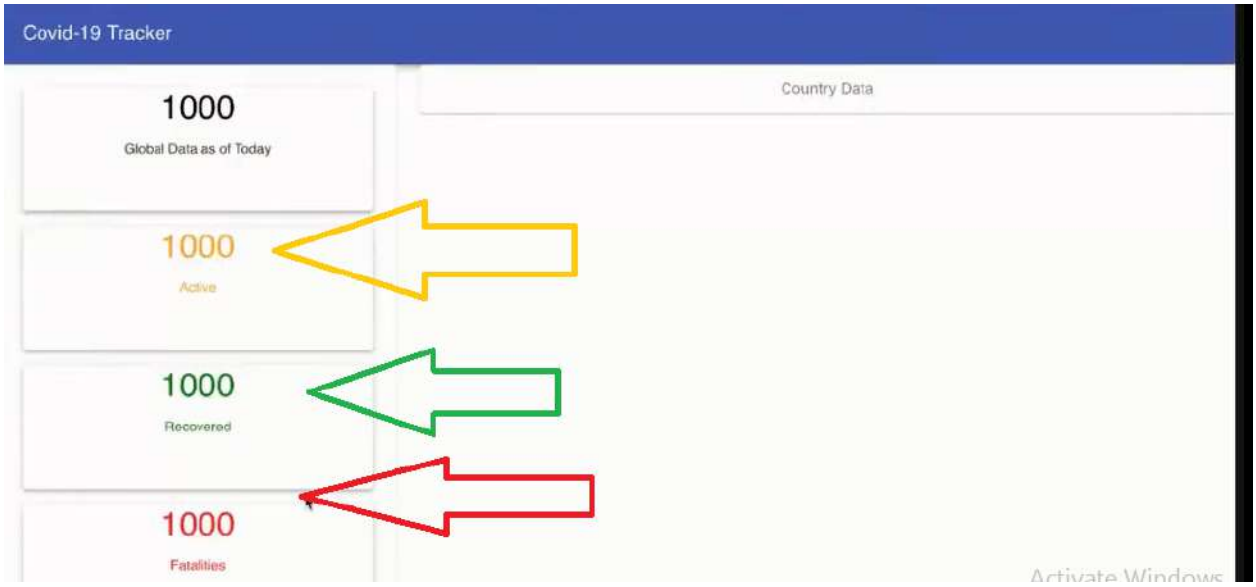


- Now create another component “GlobalData” and import this into the “MainGrid”
- In GlobalData.js use paper to shape out the different section for writing the Global Data as:
- For coloring text, we have use typography with some CSS style of our wish as shown in coding.
- Later look at the beautiful display of Global data entry

GlobalData.js

```
import React from 'react';
import { makeStyles } from '@material-ui/core/styles';
import Paper from '@material-ui/core/Paper';

const useStyles = makeStyles((theme) => ({
  root: {
    display: 'flex',
    flexWrap: 'wrap',
    '& > *': {
      margin: theme.spacing(1),
      width: 100%
      height: theme.spacing(16),
    },
  },
}));

export default function GlobalData() {
  const classes = useStyles();

  return (
    <div className={classes.root}>
      <Paper elevation={3} >
<Typography variant="h4" gutterBottom>
        1000
<Typography variant="subtitle2" gutterBottom  style ={{color 'orange'}}>
              Global Data as of Today
       </Typography>
      <Paper />
      <Paper elevation={3} >
  <div ClassName= {ClassTypography.root}>
 <NumberFormat value= <Typography variant="h4" guttreBottom  style ={{color 'orange'}}>
    1000
  </Typography>
<Typography variant="subtitle2" gutterBottom style ={{color 'orange'}}>

       Active
     </Typography>
  <div/>
      <Paper />
     <Paper elevation={3} >
   <div ClassName= {ClassTypography.root}>
 <NumberFormat value=<typography variant="h4" guttreBottom style ={{color 'green'}}>
    1000
   </Typography>
<Typography variant="subtitle2" gutterBottom style ={{color 'green'}}>
       Recovered
       </Typography>
    <div/>
     <Paper />
      <Paper elevation={3} >
<div ClassName= {ClassTypography.root}>
 <NumberFormat value= <typography variant="h4" guttreBottom style ={{color 'red'}}>
    1000
</Typography>
```

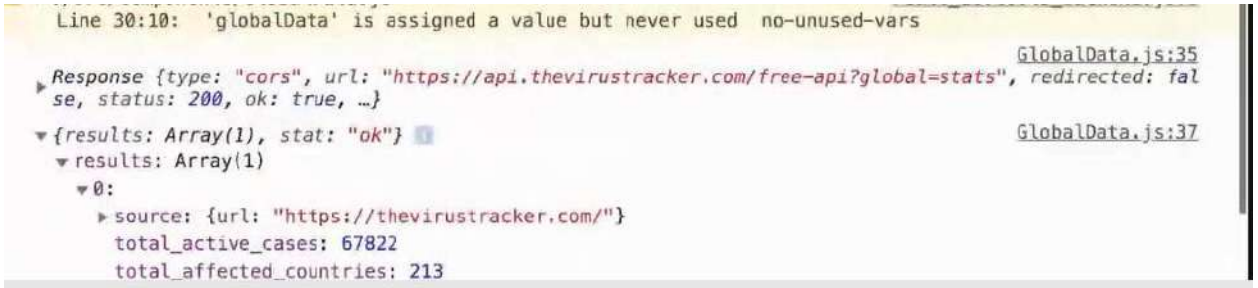
```
<Typography variant="subtitle2" gutterBottom  style ={{color 'red'}}>
      Fatalities
     </Typography>
    </div>
     <Paper />

   </div>
  );
}
```

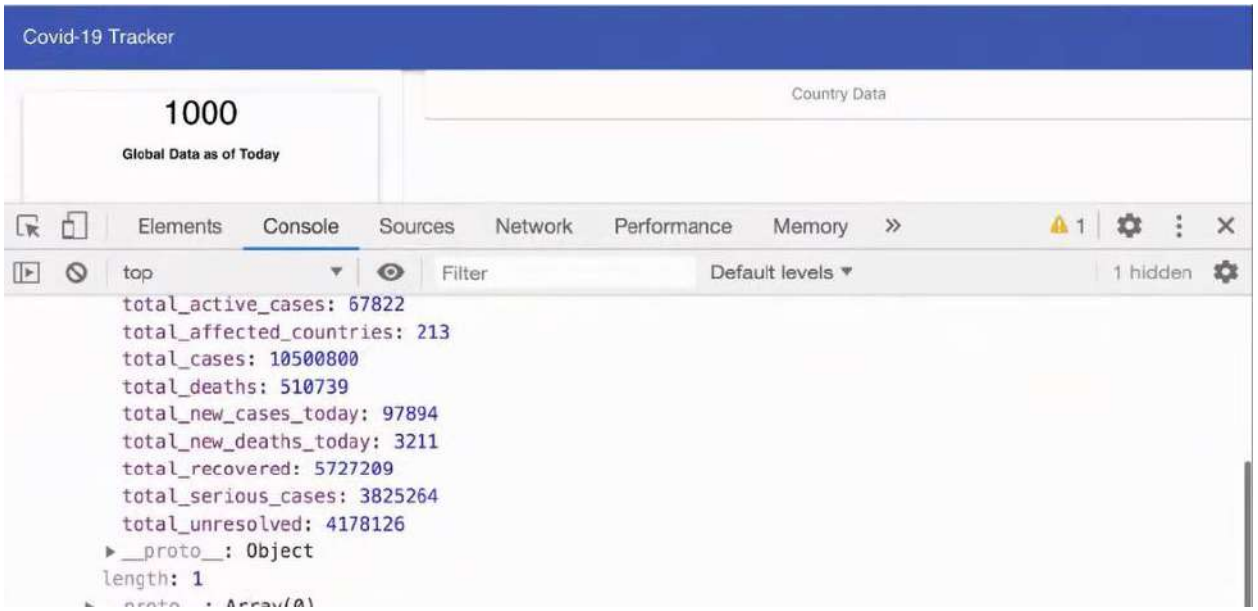


- Now to fetch data create an async function to fetch data at the background, use useState, useEffects
- Create states in the main Global data function and then use it in the usestate.
- Provide different states as;
  - {globaldata && globalData.results && globalData.results [0].total-class}
  - {globaldata && globalData.results && globalData.results [0].total-unresolved+total-Active-results}
  - {globaldata && globalData.results && globalData.results [0].total-recovered}
  - {globaldata && globalData.results && globalData.results [0].total-deaths}

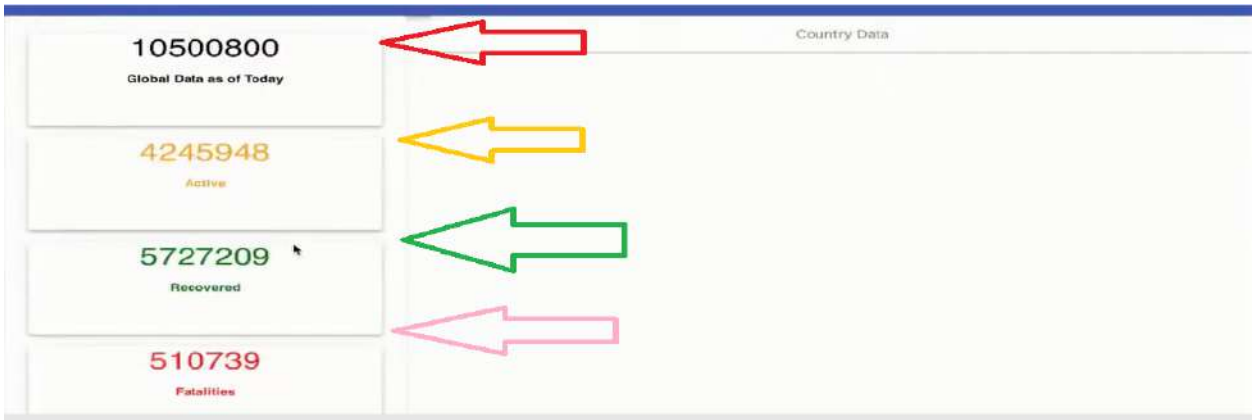
```
import React{useState,useeffect} from 'react';
import { makeStyles } from '@material-ui/core/styles';
import Paper from '@material-ui/core/Paper';

const useStyles = makeStyles((theme) => ({
  root: {
    display: 'flex',
    flexWrap: 'wrap',
    '& > *': {
      margin: theme.spacing(1),
      width: 100%
      height: theme.spacing(16),
    },
  },
}));

  export default function GlobalData() {
   const Classtypography=useStyletypography();
   const classes = useStyles();
   const [globalData, setGlbalData]=useState();


    useEffect(()  =>
    aync function fetchGlobalData(){
    const apiResponse = await fetch ('https://api.
    console.log(apiResponse);
    const dataFromAPI = await apiResponse.json();
    console.log(dataFromAPI);
    setGlobalData(dataFromAPI);
  }
  fetchGlobalData();
})

  return (
    <div className={classes.root}>
      <Paper elevation={3} >
<Typography variant="h4" gutterBottom>
        1000
<Typography variant="subtitle2" gutterBottom  style ={{color 'orange'}}>
              Global Data as of Today
{globaldata && globalData.results && globalData.results [0].total-class}
       </Typography>
      <Paper />
      <Paper elevation={3} >
  <div ClassName= {ClassTypography.root}>
  <typography variant="h4" guttreBottom  style ={{color 'orange'}}>
    {globaldata && globalData.results && globalData.results [0].total-unresolved+total-Active-
results}
  </Typography>
<Typography variant="subtitle2" gutterBottom style ={{color 'orange'}}>

       Active
     </Typography>
  <div/>
      <Paper />
```

```
    <Paper elevation={3} >
   <div ClassName= {ClassTypography.root}>
   <typography variant="h4" guttreBottom style ={{color 'green'}}>
    {globaldata && globalData.results && globalData.results [0].total-recovered}
   </Typography>
<Typography variant="subtitle2" gutterBottom  style ={{color 'green'}}>
       Recovered
       </Typography>
    <div/>
     <Paper />
      <Paper elevation={3} >
<div ClassName= {ClassTypography.root}>
  <typography variant="h4" guttreBottom style ={{color 'red'}}>
    {globaldata && globalData.results && globalData.results [0].total-deaths}
   </Typography>
<Typography variant="subtitle2" gutterBottom  style ={{color 'red'}}>
       Fatalities
      </Typography>
     </div>
      <Paper />

    </div>
  );
}
```





- Now you can add number format so that it put commas and it will look easier to read and also import Number format in “GlobalData.js”



Session 11: React Router v6 and Web Animation API

1. **Add Router Into Application (Developing An Application)**

One of the quickest ways to add React and React Router to a website is to use good ol' <script> tags and global variables. React Router is compatible with React 16.8+. Just add the following <script> tags to your HTML, just before the closing </body> tag:

```
  <!-- Other HTML for your app goes here -->

  <!-- The node we will use to put our app in the document -->
  <div id="root"></div>

  <!-- Note: When deploying to production, replace "development.js"
       with "production.min.js" in each of the following tags -->

  <!-- Load React and React DOM -->
  <!-- See https://reactjs.org/docs/add-react-to-a-website.html to learn more -->
  <script src="https://unpkg.com/react@>=16.8/umd/react.development.js" crossorigi
  <script src="https://unpkg.com/react-dom@>=16.8/umd/react-dom.development.js"
crossorigin></script>

  <!-- Load history -->
  <script src="https://unpkg.com/history@5/umd/history.development.js" crossorigin>

  <!-- Load React Router and React Router DOM -->
  <script src="https://unpkg.com/react-router@6/umd/react-router.development.js"
crossorigin></script>
  <script src="https://unpkg.com/react-router-dom@6/umd/react-router-dom.developm
crossorigin></script>

  <!-- A simple example app -->
  <script>
  var e = React.createElement;
  var Router = ReactRouterDOM.BrowserRouter;
  var Routes = ReactRouterDOM.Routes;
  var Route = ReactRouterDOM.Route;

  ReactDOM.render(
    (
      e(Router, null, (
        e(Routes, null, (
          e(Route, {
            element: e('div', null, 'Hello, React Router!')
          })
        ))
      ))
    ),
    document.getElementById('root')
  );
  </script>

</body>
```

2nd method is to install with a package manager as given:

Before using a bundler for your project, you'll first need to install React Router to your local node_modules directory using a JavaScript package manage

$ npm install history@5react-router-dom@6

We are going to use 2nd method

Create React App

- Once your project is set up and React Router is installed as a dependency, open the src/index.js in your text editor. Import BrowserRouter from react-router-dom near the top of your file.
- To make sure your app has the necessary context to handle state and logic controlled by React Router, you need to wrap the entire app inside the BrowserRouter component. Where ReactDOM.render is called to load your app, update the first argument so that <App /> is nested accordingly: import { BrowserRouter as Router } from "react-router-dom";

```
ReactDOM.render(
 <React. strictMode>
 <Router>
    <App />
  </Router>
  document.getElementById('root')
);
```

- in our index.js follow the following coding

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker'; import { BrowserRouter as Router } from "react-
router-dom";
 import { BrowserRouter as Router } from "react-router-dom";

 ReactDOM.render(
  <React. strictMode>
  <Router>
     <App />
   </Router>
   document.getElementById('root')
 );
```

- create an App.js and inside it create two-component, about component and home component:

App.js

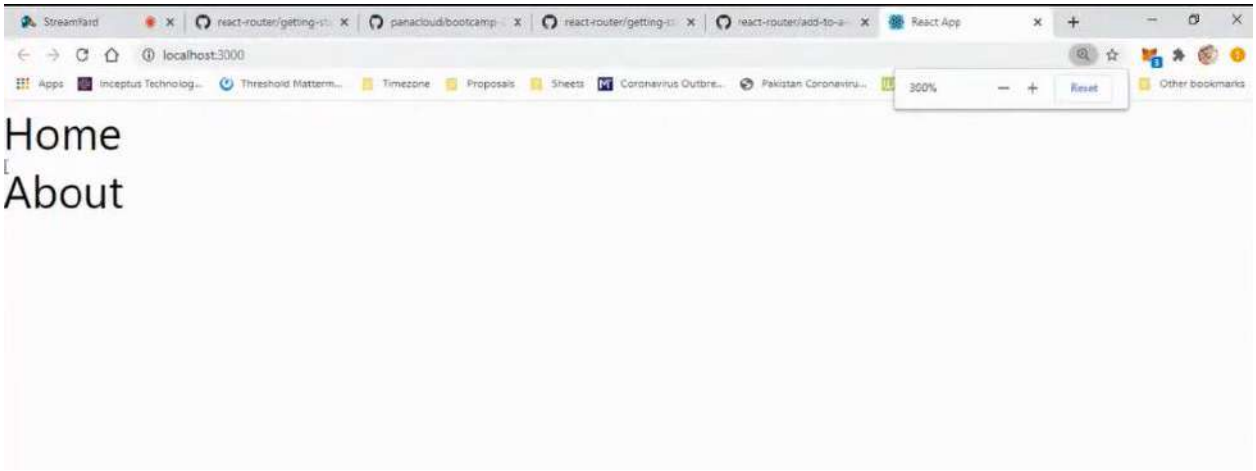
```
import react from 'react';
function App(){
<div>
   <Home></Home>
   <About></About>
</div>
);
}
export default App;
```

About.js

```
import react from 'react';
function about(){
<div>
</div>
);
}
export default about;
```

Home.js

```
import react from 'react';
function Home(){
<div>
</div>
);
}
export default Home;
```

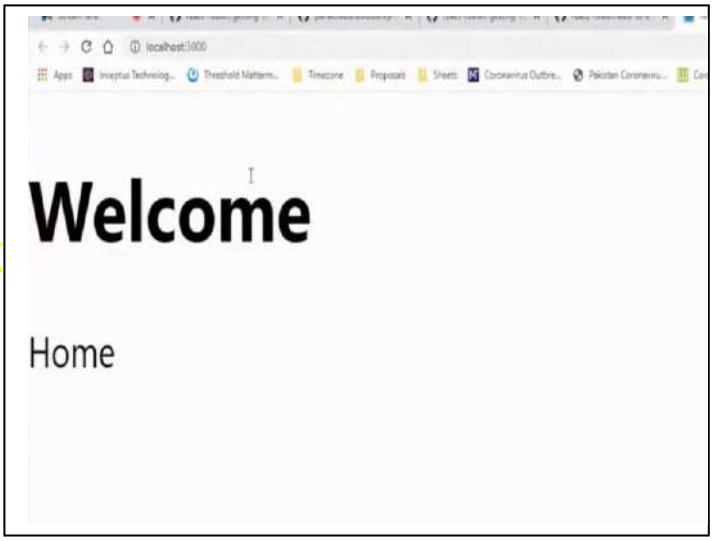


- inside App.js use route component, provide a path that is being displayed by switch

About.js

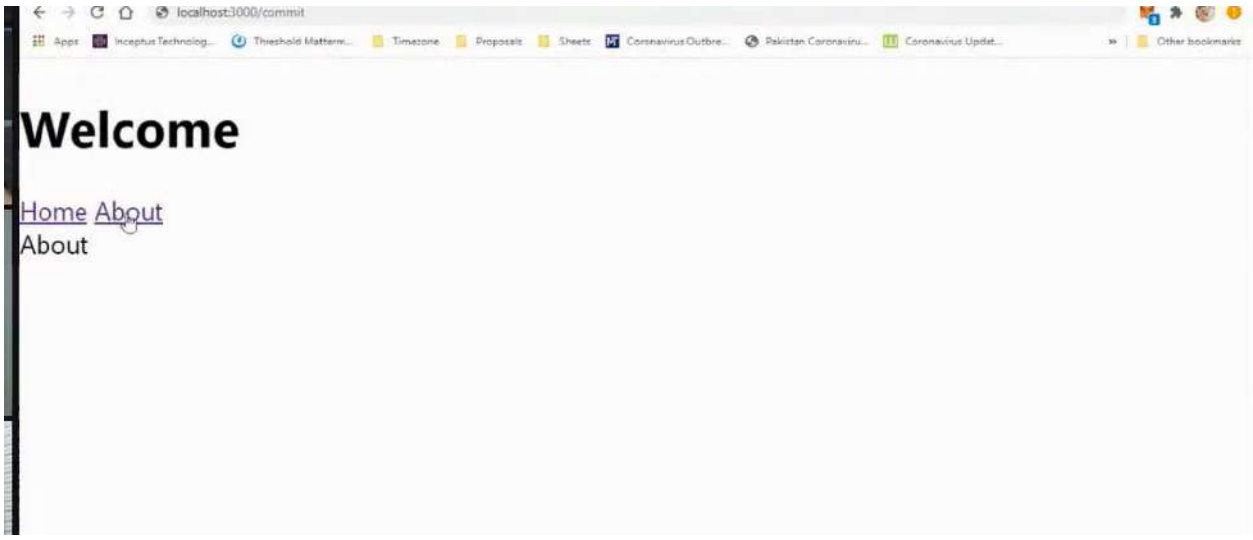
```
import react from 'react';
function about(){
<div>
  <h1>welcome<h1>
   <Routes>
      <Route path="/" element={<Home />}></Rou
       <Route path="about" element={<About />}>
   </Routes>
</div>
);
}
export default about;
```

output



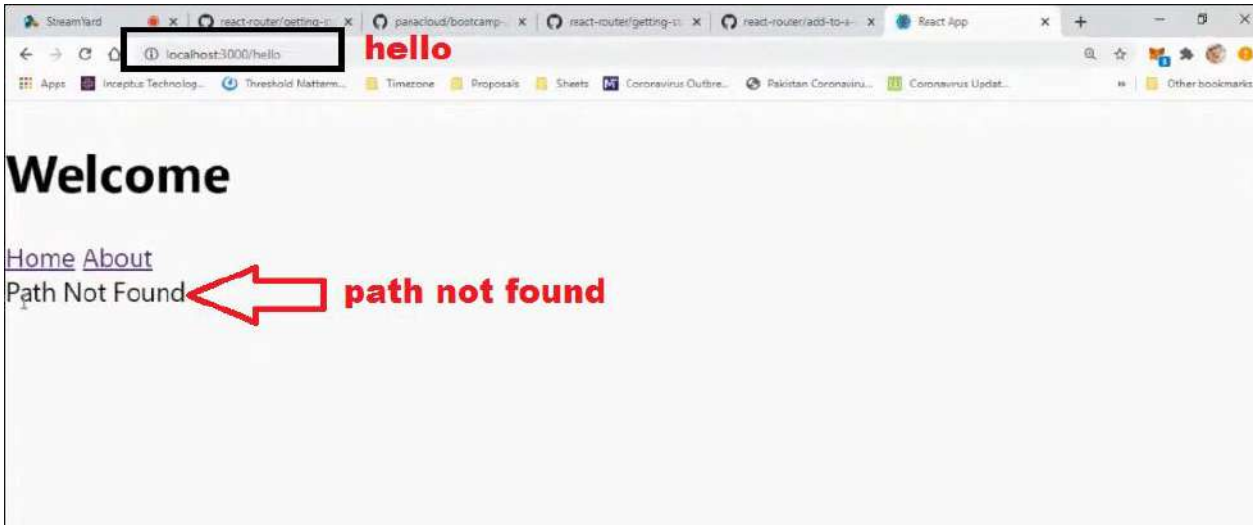
- FOR CLICKS: Now import routes as well as links in the App.js to link home and about components to be displayed on the screen from routes.
- You can click either to home or about to get the respective component detail

```
import react from 'react';
import './App.cs
import about from'/About';
import Home from '/Home';
import React from 'react';
  import { Routes, Route, Link } from 'react-router-dom';

function App(){
<div>
  <h1>welcome<h1>
  <div>
      <Link to ="/">Home</Link> {' '}
       <Link to ="about">About</Link>

    <Routes>
      <Route path="/" element={<Home/>}></Route>
       <Route path="about" element={<About/>}></Route>
   </Routes>
</div>
```



- If we write anything else instead of hello or about in the URL then it should display nothing for that purpose we will use another tag of Route with ‘ *’ and import function of “page not found” as shown :
- It will help you to know you have entered an invalid entry in the URL.

```
import react from 'react';
import './App.cs';
import about from '/About';
import Home from '/Home';
import React from 'react';
 import { Routes, Route, Link } from 'react-router-dom';
function NotFound() {
 return <div> Path not found</div>

function App(){
<div>
  <h1>welcome<h1>
  <div>
      <Link to ="/">Home</Link> {' '}
       <Link to ="about">About</Link>

   <Routes>
      <Route path="/" element={<Home/>}></Route>
       <Route path="about" element={<About/>}></Route>
       <Route Path= '*'element={<NotFound/>}</Route>
   </Routes>
</div>
```



Sheo store

- Create another component “product.js” and import in the App.js
- Provide product route and links as;

Product.js

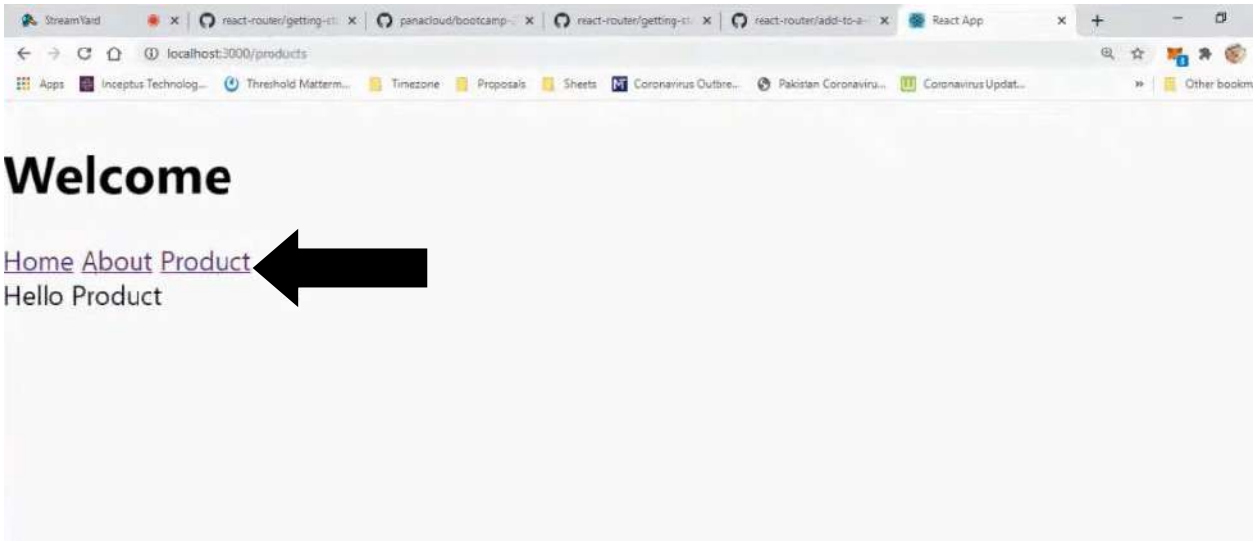
```
import react from 'react';
function product() {
return (
   <div>
    Hello product
   </div>
   );
}
export default product;
```

App.js

```
import react from 'react';
import './App.cs ';
import about from'/About';
import Home from '/Home';
import prodduct from '/products';

import React from 'react';
   import { Routes, Route, Link } from 'react-router-dom';
      function NotFound() {
       return <div> Path not found</div>

function App() {
<div>
  <h1>welcome<h1>
  <div>
      <Link to ="/">Home</Link>  {'  '}
       <Link to ="about">About</Link> {' '}
       <Link to ="products">product</Link>

   <Routes>
      <Route path="/" element<{<Home/>}></Route>
       <Route path="about" element<{<About/>}></Route>
       <Route Path= '*'element={NotFound}</Route>
       <Route path="products" element<{<product/>}></Route>
</Routes>
```

- **When you will write product in the URL you will get product data as “Hello Product”.**

Product link:



- **To** display a variety of show useparams as a parameter in product.js as well as get element ID in div
- In app.js in route provide product name as variable and its ID: **<Route path="products/:productId" element<{<product/>}></Route>**

Product.js

```
import React from 'react';
import { useparams } from 'react-router';

function product() {
  const {productID} = useParams();
  return (
  <div>
     Hello Product{ProductID}
  </div>
 );
}
```

```
import react from 'react';
import './App.cs';
import about from'/About';
import Home from '/Home';
import product from '/products';

import React from 'react';
   import { Routes, Route, Link } from 'react-router-dom';
      function NotFound() {
       return <div> Path not found</div>

function App() {
<div>
  <h1>welcome<h1>
  <div>
      <Link to ="/">Home</Link> {' '}
       <Link to ="about">About</Link>{' '}
       <Link to ="products">product</Link>

   <Routes>
      <Route path="/" element={<Home/>}></Route>
       <Route path="about" element={<About/>}></Route>
       <Route Path= '*'element={NotFound}</Route>
       <Route path="products/:productId" element<{<product/>}></Route>

</Routes>
```

- **this is how you can get the provided URL product because of useparams**

Nested Route

- to display particular pages with their details insides we use a nested route as shown
- in App.js provides links of products as mobile, laptop
- then in Routes provide route path, product element, and productdetails

- create an outlet in prouduct.js, in div section in place of outlet the productDetail will be displayed
- this productDetail will use useparams to get the id of that product

App.js

```
import react from 'react';
import './App.cs';
import about from'/About';
import Home from '/Home';
import prodduct from '/products';

import React from 'react';
   import { Routes, Route, outlet  ,Link } from 'react-router-dom';
      function NotFound(){
       return <div> Path not found</div>

function App(){
<div>
  <h1>welcome<h1>
  <div>
      <Link to ="/">Home</Link> {' '}
       <Link to ="about">About</Link>{' '}

       <Link to ="products">product</Link>{' '}

     <Link to ="products"/mobile">Mobile</Link>{' '}
```

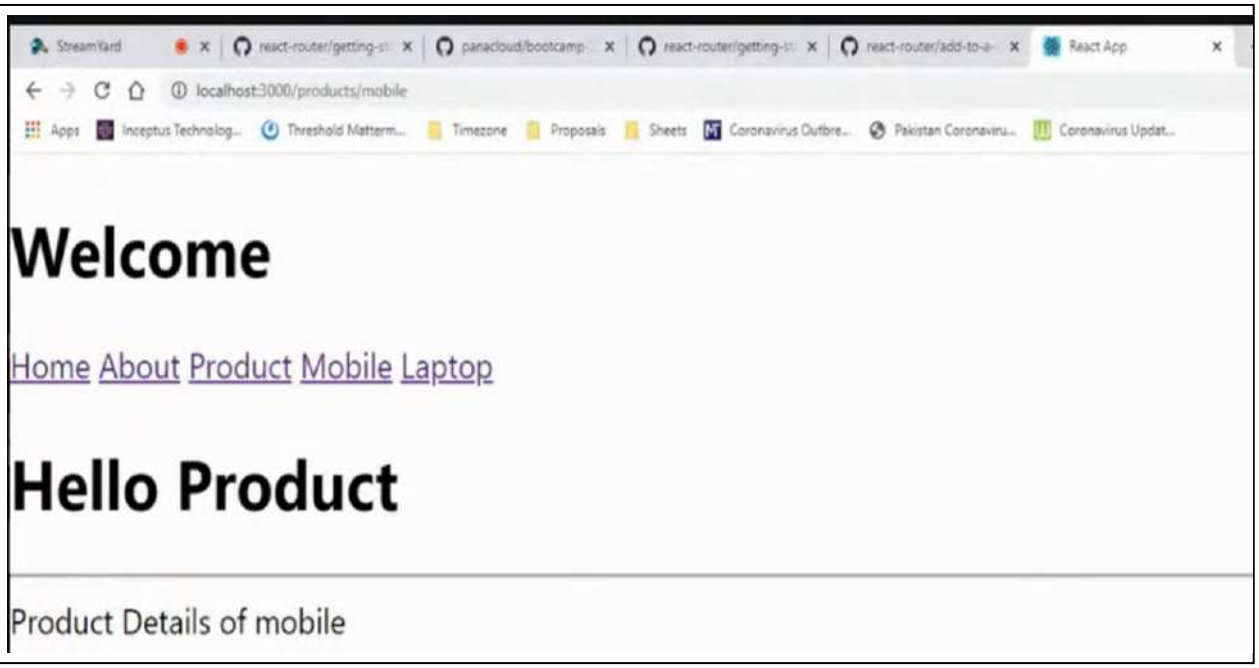
```
   <Routes>
      <Route path="/" element<{<Home/>}></Route>
       <Route path="about" element<{<About/>}></Route>
       <Route Path= '*'element={NotFound}</Route>
       <Route path="products/:productId" element={<product/>}></Route>
    <Route path=":productId" element={<ProductDetails/>}></Route>


</Routes>
```

productDetail.js

```
import React from 'react';
import {useparams} from 'react-ro

function productDetails(){
   const {productID} = useParams
   return (
   <div>
       product Details of {ProductID
   </div>
  );
 }
export default ProductDetails:
```

product.js

```
import React from 'react';
import {useparams, outletts} from
'react-router';

function product(){
   const {productID} = useParams();
   return (
   <div>
      <h1>Hello product</h1>
      <hr/>
     <div>
         <outlet/>
     </div>
   </div>
  );
```

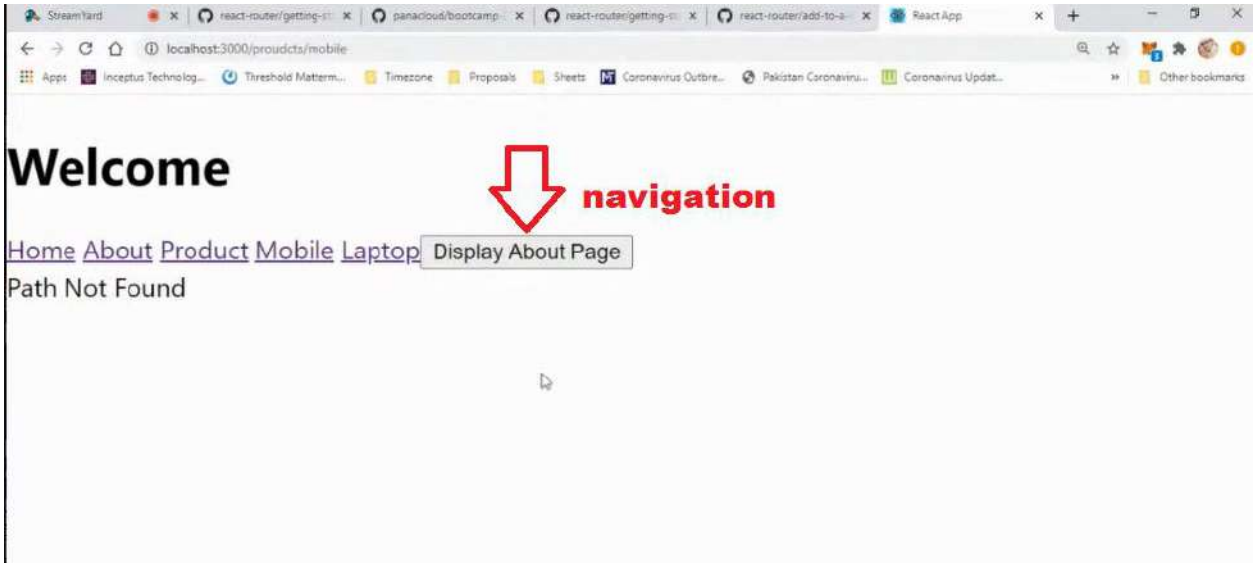


For Navigation

- **create useNavigate from the route router**
- **call its function with navigate('/about')**
- **on onclick, it will display the navigation of data dynamically**

App.js

```
const navigate=useNavigate();import react from 'react';
import './App.cs'/
import about from'/About';
import Home from '/Home';
import prodduct from '/products';

import React from 'react';
   import { Routes, outlet, useNavigate, Link } from 'react-router-dom';
      function NotFound() {
       return <div> Path not found</div>

function App(){
const navigate=useNavigate();
<div>
  <h1>welcome<h1>
  <div>
      <Link to ="/">Home</Link> {'  '}
       <Link to ="about">About</Link>{'  '}

       <Link to ="products">product</Link>{'  '}

     <Link to ="products"/mobile">Mobile</Link>{' '}

       <Link to ="products"/laptop">Laptop</Link>
                <button onClick={ ()=>  {
            navigate( '/about' )
          }}>
          Display About Page
           </button>

   <Routes>
      <Route path="/" element<{<Home/>}></Route>
       <Route path="about" element<{<About/>}></Route>
       <Route Path= '*'element={NotFound}</Route>
       <Route path="products/:productId" element<{<product/>}></Route>
    <Route path=":productId" element={<ProductDetails/>}></Route>
</Routes>


</Route>
```



List of products using navigation

- **create components folder**
  - **App.js**
  - **Header.js**
  - **ProductDetail.js**
  - **Product.js**
  - **Home.js**
  - **PageNotfound.js**

In App.js import the following for their respective functions

- **import logo from'./logo.svg';**
- **import {BrowserRouter,Routes,Route,link,Router} from "react-router-dom"; for providind routes and links**
- **import {Header} from '.components/Header'; for haeding**
- **import {Home} from '.component/Home';as component**
- **import {product} from '.component/product'; as component**
- **import {productDetails} from './productDetai'; to get product details**
- **import './App.css';**

- **create Route with an nested Route to provide the particular page link detail in multiple pages;**

```
 <Routes
<Route path ="/" element={<Home/>}></Route>
<Route path="products"element={<products/}>
  <Route path ="/" element={<products/>}></Route>
  <Route path =":productID" element={<productDetails/>}></Route>
 <Route>
```

- **Route for If any component we provided does not match**

```
<Route path ="*"element={<Not Found/>}></Route>
```

- **create a component haeder.js and product.js with their respective headings inside div and then import these components to the App.js**

App.js

```
import React from 'react';
import logo from'./logo.svg';
import {BrowserRouter,Routes,Route,link,Router} from "react-router-dom";
import { Header } from '.components/Header';
import {Home} from '.component/Home';
import {product} from '.component/product';
import {productDetails} from './productDetai';
import './App.css';
  function App() {
   return (
   <div>
    <Browserouter>
     <Header/>
     <Routes
     <Route path ="/" element= {<Home/>}></Route>
     <Route path="products"element={<products/}>
        <Route path ="/" element={<products/>}></Route>
        <Route path =":productID" element={<productDetails/>}></Route>
          <Route path ="*"element={<Not Found/>}></Route>
              <Route>
    <Routes>
</Browserouter>
   </div>
   );
export default App;
```

Header.js

```
import React from 'react'
import {Link} from "react-router-dom";
export const Header =()=>{
  return{
    <nav>
          <Link
to="home">Home</Link>{" | "}
          <Link
to="product">products</Link>
    </nav>

   )

 }

export default Header;
```

product.js

```
 import React from'react';
export const products=()=>{
  return(
    <div>
      <h4>products<h4>
    </div>
   )

 }

export default products;
```

pageNotFound.js

```
import React from 'react';
export const NotFound = () => {
   return(
     <div>
     <h2> page Not Found<h2>
    </div>
  )
 }
export default page not Found;
```

Home.js

```
import React from 'react';
export const Home=()=>{
   return(
     <div>
     <h2> Home<h2>
    </div>
  )
 }
export default Home;
```

2. Shoe Store

- create a component app.js
- import in it :

```
import React from 'react';

import logo from'./logo.svg';

import {BrowserRouter,Routes,Route,link,Router} from "react-router-dom";

import {Header} from '.components/Header';

import {Home} from '.component/Home';

import {product} from '.component/product';

import {productDetails} from './productDetai';

import {productindex}from './productindex';

import './App.css';
```

- inside browserouter tag insert haeder, routes ,nested routes to proovide particular page detail from multiple pages ,"*" element for invalid entry

```
return (
<div>
 <Browserouter>
  <Header/>
  <Routes
  <Route path ="/" element={<Home/>}></Route>
  <Route path="products"element={<products/}>
    <Route path ="/" element={<productIndex/>}></Route>
    <Route path ":productID" element={<productDetails/>}></Route>
      </Route>
      <Route path ="*"element={<Not Found/>}></Route>
         <Route>

 <Routes>

</Browserouter>

  </div>
```

- now create another component productindex.js
- export the following shoe images detail with thier location and URL as

```
export const ProuctIndex =()=>{

const shoes = {

  "air-jordan-3-valor-blue": {
```

```
    name: "VALOUR BLUE",
    img:
      "https://secure-
images.nike.com/is/image/DotCom/CT8532_104_A_PREM?$SNKRS_COVER_WD$&align
=0,1"
  },
  "jordan-mars-270-london": {
    name: "JORDAN MARS 270 LONDON",
    img:
      "https://secure-
images.nike.com/is/image/DotCom/CV3042_001_A_PREM?$SNKRS_COVER_WD$&align
=0,1"
  },
  "air-jordan-1-zoom-racer-blue": {
    name: "RACER BLUE",
    img:
      "https://secure-
images.nike.com/is/image/DotCom/CK6637_104_A_PREM?$SNKRS_COVER_WD$&align
=0,1"
  }
};
```

- then create an object entry map to display the list of all shoe object with their names and images in unorder list and provide heading of name of object and link it with productId as

```
return(
      <div>
  <ul>
      {object.entries(shoes).map((productID,{name,img}])=.
    (
     <li key{productID}>
      <Link to="/productID}
        <h2>{name}</h2>
         <img src={img} alt={name}/>
      </Link>

     </li>
    )
   )}
  </ul>
```

- in productDetail.js provide the product img detail, location and URL
- use useparams declare function of const product console lo as

```
const{productId}=useparams();
const produuct = shoes[productId];
const{nam.img}=product;
console.log({productId});
console.log(shoes);
```

- so far create Home.js and Product.js as well
- when you call the object with its provided URL it will display that object list with name and images as shown

app.js

```
import React from 'react';
import logo from'./logo.svg';
import {BrowserRouter,Routes,Route,link,Router} from "react-router-dom";
import {Header} from '.components/Header';
import {Home} from '.component/Home';
import {product} from '.component/product';
import {productDetails} from './productDetai';
import {productindex}from './productindex';
import './App.css';
  function App(){
   return (
   <div>
    <Browserouter>
     <Header/>
     <Routes
     <Route path ="/" element={<Home/>}></Route>
     <Route path="products"element={<products/}>
        <Route path ="/" element={<productIndex/>}></Route>
        <Route path =":productID" element={<productDetails/>}></Route>
          </Route>
          <Route path ="*"element={<Not Found/>}></Route>
              <Route>
    <Routes>
</Browserouter>
  </div>

  );

export default App;
```

productIndex.js

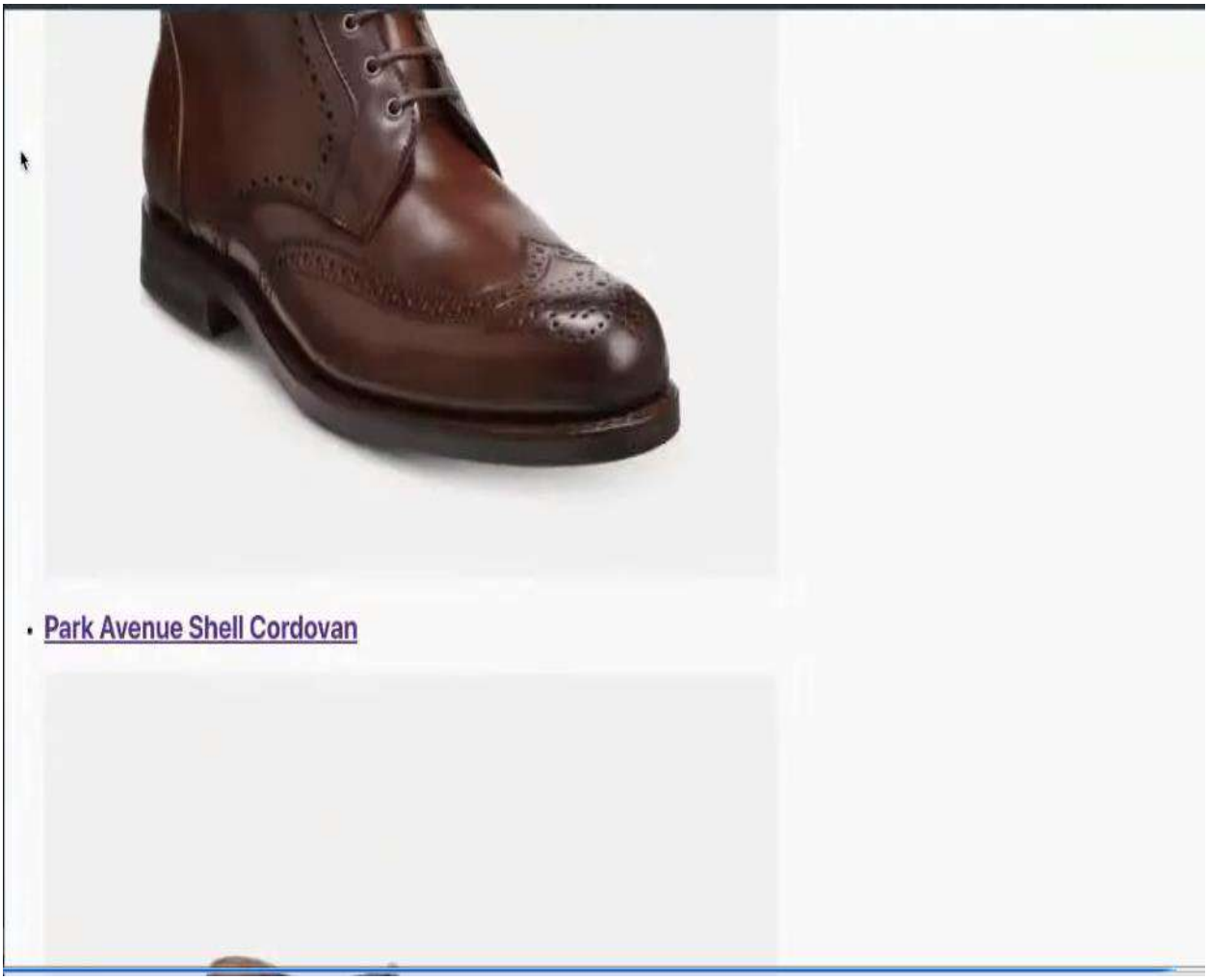
```
import React from 'react'
export const ProuctIndex =()=>{
const shoes = {
  "air-jordan-3-valor-blue": {
    name: "VALOUR BLUE",
    img:
      "https://secure-
images.nike.com/is/image/DotCom/CT8532_104_A_PREM?$SNKRS_COVER_
WD$&align=0,1"
  },
  "jordan-mars-270-london": {
    name: "JORDAN MARS 270 LONDON",
    img:
      "https://secure-
images.nike.com/is/image/DotCom/CV3042_001_A_PREM?$SNKRS_COVER_
WD$&align=0,1"
  },
  "air-jordan-1-zoom-racer-blue": {
    name: "RACER BLUE",
   img:

     "https://secure-
images.nike.com/is/image/DotCom/CK6637_104_A_PREM?$SNKRS_COVER_WD$&align
=0,1"

  }
};
   return(
        <div>
  <ul>
      {object.entries(shoes).map((productID,{name,img}])=.
     {
      <li key{productID}>
        <Link to="/productID}
          <h2>{name}</h2>
         <img src={img} alt={name}/>

        </Link>

       </li>
     )
   )}

  </ul>
```

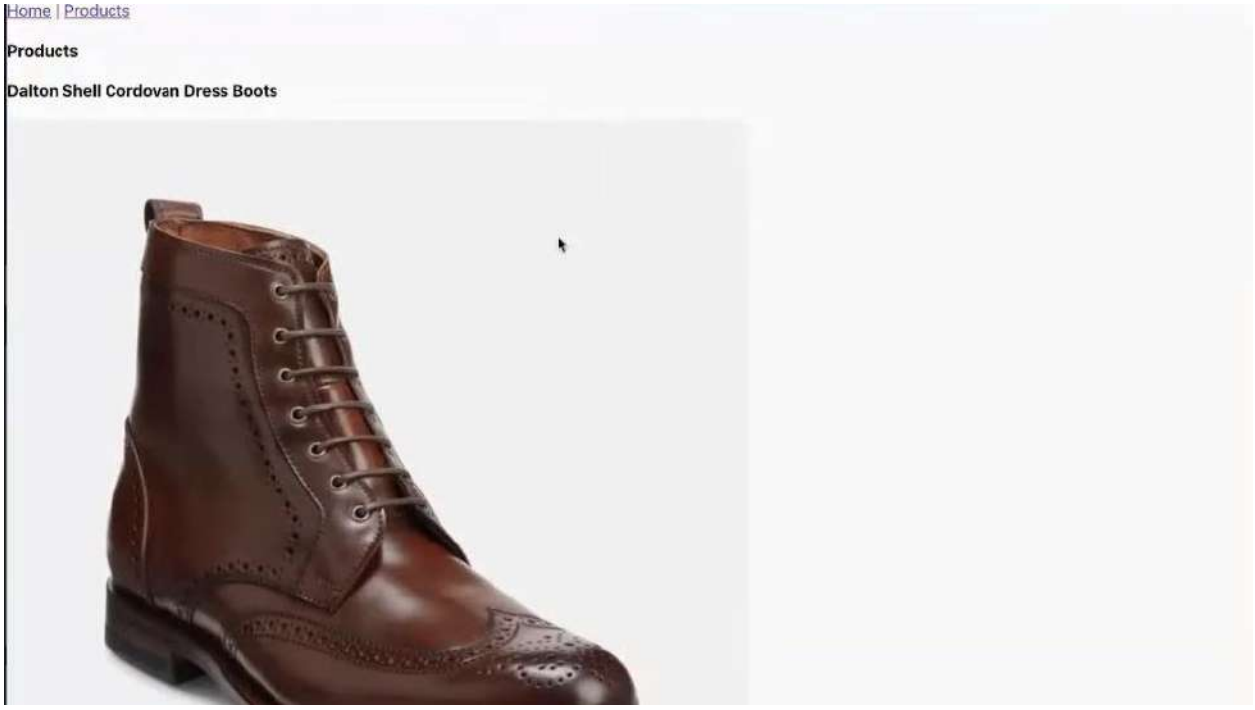
productDetail.js

```
import React from 'react'
import {OUTlet,Link} from 'react-router-dom'
export const productDetails =()=>{
const shoes = {
  "air-jordan-3-valor-blue": {
    name: "VALOUR BLUE",
    img:
      "https://secure-
images.nike.com/is/image/DotCom/CT8532_104_A_PREM?$SNKRS_COVER_WD$&ali
gn=0,1"
  },
  "jordan-mars-270-london": {
    name: "JORDAN MARS 270 LONDON",
    img:
      "https://secure-
images.nike.com/is/image/DotCom/CV3042_001_A_PREM?$SNKRS_COVER_WD$&ali
gn=0,1"
  },
  "air-jordan-1-zoom-racer-blue": {
    name: "RACER BLUE",
    img:
      "https://secure-
images.nike.com/is/image/DotCom/CK6637_104_A_PREM?$SNKRS_COVER_WD$&ali
gn=0,1"
  }
};
const{productId}=useparams();
const produuct = shoes[productId];
const{nam.img}=product;
console.log({productId});
console.log(shoes);

  return(
   <div>
     <h4>{name}</h4>
     <img src={img} alt={name}/>
     <ul>
       {object.entries(shoes).map((productID,{name,img}])=.
     (
      <li key{productID}>
        <Link to="/productID}
          <h2>{name}</h2>
           <img src={img} alt={name}/>
        </Link>

       </li>
     )
    )}
   </ul>
 </div>
)
```

product.js

```
import React from 'react'
import {OUTlet,Link} from 'react-router-dom'
export const products =()=>{
return(
    <div>
       <h4>products</h4>
       <outlet/>

   </div>

  )
```




Home | Products
Products
Dalton Shell Cordovan Dress Boots

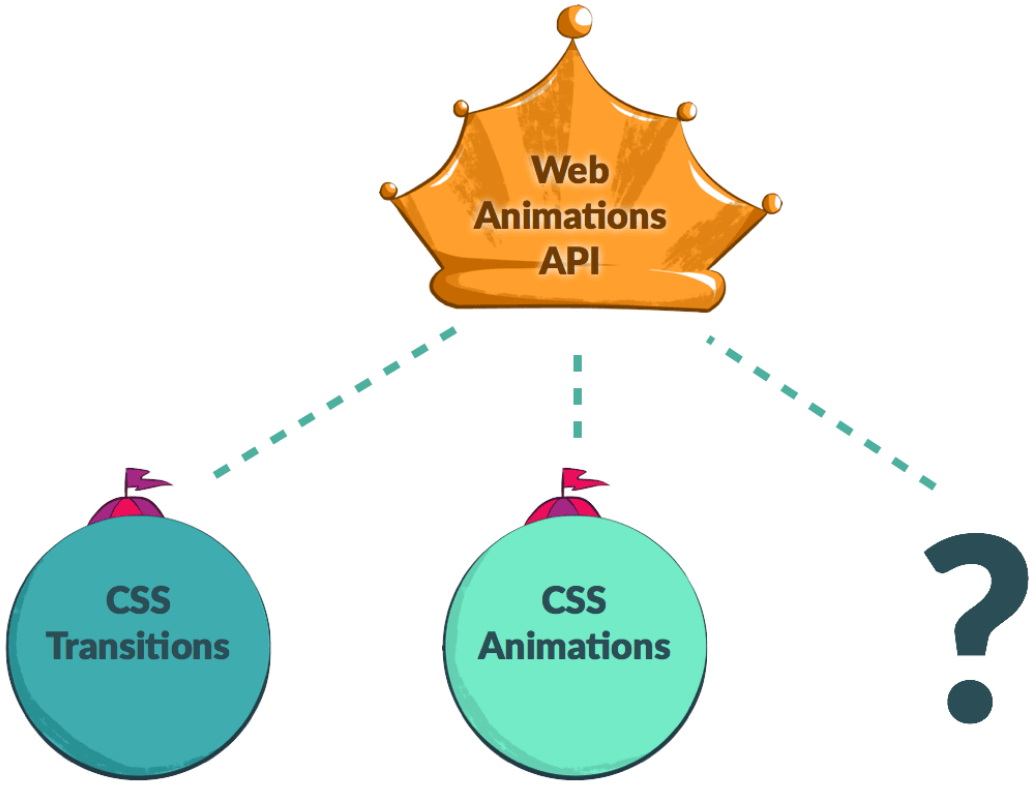
Session 12: Project 4A and 4B without React

1. What Is Animation?

Animation is a method of photographing successive drawings, models, or even puppets, to create an illusion of movement in a sequence. Because our eyes can only retain an image for 1/16 of a second, when multiple images appear in fast succession, the brain blends them into a single moving image. In traditional animation, pictures are drawn or painted on transparent celluloid sheets to be photographed and shown on film. Early cartoons are examples of this, but today, most animation is made with computer-generated imagery or CGI.

To create the appearance of smooth motion from these drawn, painted, or computer-generated images, frame rate, or the number of consecutive images that are displayed each second, is considered. Moving characters are usually shot “on twos” which just means one image is shown for two frames, totaling in at 12 drawings per second. 12 frames per second allows for motion but may look choppy. In the film, a frame rate of 24 frames per second is often used for smooth motion animation.

2. Web Animations API Concepts



The Two Models: Timing and Animation

The Web Animations API runs on top of two models, one that handles time—Timing—and one that handles visual change over time—Animation. The Timing Model keeps track of how far along a set timeline we've come. The Animation Model determines what the animated object should look like at any given time.

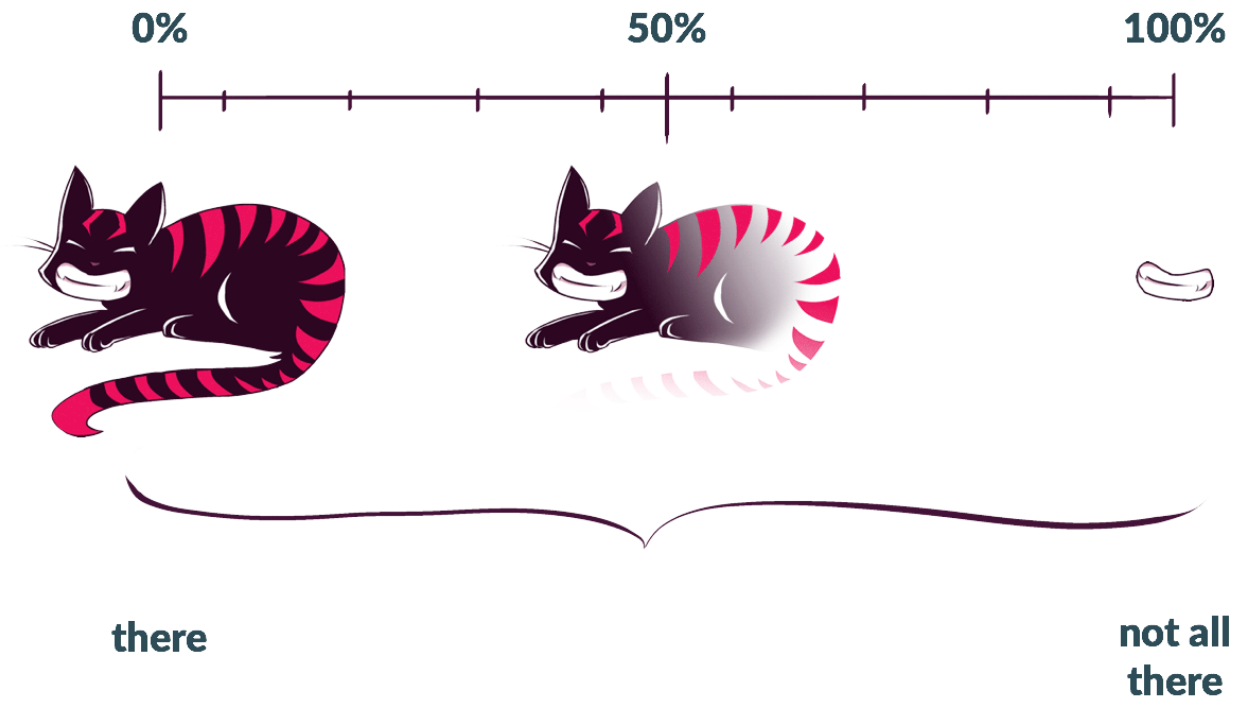
Timing

The Timing Model is the backbone of working with the WAAPI. Each document has a master timeline, Document.timeline, which stretches from the moment the page is loaded to infinity — or until the window is closed. Spread along that timeline according to their durations are our animations. Each animation is anchored to a point in the timeline by its startTime, representing the moment along the document's timeline when the animation starts playing.

All the animation's playback relies on this timeline: seeking the animation moves the animation's position along the timeline; slowing down or speeding up the playback rate condenses or expands its spread across the timeline; repeating the animation lines up additional iterations of it along the timeline. In the future, we might have timelines based on gestures or scroll position or even parent and child timelines. The Web Animations API opens up so many possibilities!

Animation

The animation model can be thought of as an array of snapshots of what the animation could look like at any given time, lined up along the duration of the animation.



2.1. Core Concepts

Web animations consist of Timeline Objects, Animation Objects, and Animation Effect Objects working together. By assembling these disparate objects, we can create animations of our own.

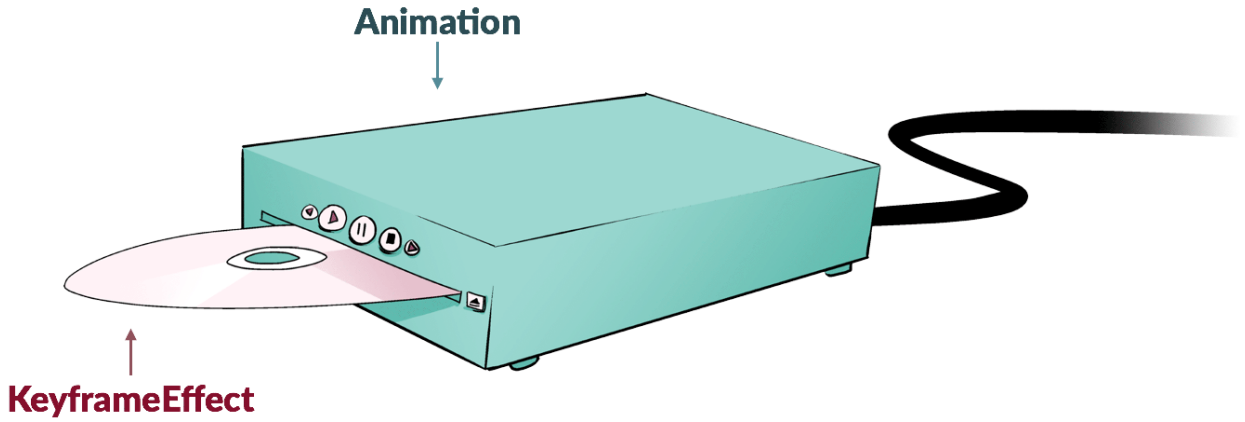
Timeline

Timeline objects provide the useful property currentTime, which lets us see how long the page has been opened for: it's the "current time" of the document's timeline, which started when the page was opened. As of this writing, there's only one kind of timeline object: the one based on the active

document's timeline. In the future, we may see timeline objects that correspond to the length of the page, perhaps a ScrollTimeline, or other things entirely.

Animation

Animation objects can be imagined as DVD players: they're used for controlling media playback, but without media to play, they don't do anything. Animation objects accept media in the form of Animation Effects, specifically Keyframe Effects (we'll get to those in a moment). As a DVD player, we can use the Animation Object's methods to play, pause, seek, and control the animation's playback direction and speed.



An illustration likening how an Animation plays a KeyframeEffect like a DVD player plays a DVD.

Animation Effect

If Animation objects are DVD players, we can think of Animation Effects, or Keyframe Effects, like DVDs. Keyframe Effects are a bundle of information including at the bare minimum a set of keys and the duration they need to be animated over. The Animation Object takes this information and, using the Timeline Object, assembles a playable animation we can view and reference.

We currently have only one animation effect type available: KeyframeEffect. Potentially we could have all kinds of Animation Effects in the future—e.g. effects for grouping and sequencing, not unlike features we had in Flash. Group Effects and Sequence Effects have already been outlined in the currently-in-progress level 2 spec of the Web Animations API.

Assembling the Animation from disparate pieces

We can assemble all these pieces to create a working animation with the Animation() Constructor or we can use the Element.animate() shortcut function. (Read more about how to use Element.animate() in Using the Web Animations API.)

3. Uses

The API allows for the creation of dynamic animations that can be updated on the fly as well as more straightforward, declarative animations like those CSS creates. It can be used in automated tests to ensure that your UI animations are running correctly. It opens up the browser's rendering

engine for building animation development tools like timelines. It is also a performant base on which to build a custom or commercial animation library. (See Animating like you just don't care with Element.animate.) In some instances, it may negate the need for a fully-fledged library entirely in the same way Vanilla JavaScript can be used without jQuery for many purposes.

- **animation is basically a manipulating of ny object transform property**

create an index.html without using react Lets us use API Animation

- provide element a shape frame and provide this frame different background transition colors as

```
var ShapeElement= document.querySelector(".shape");
var frames =[
  {background: "red"},
  {background: "blue"},
  {background:"yellow"},
];
```

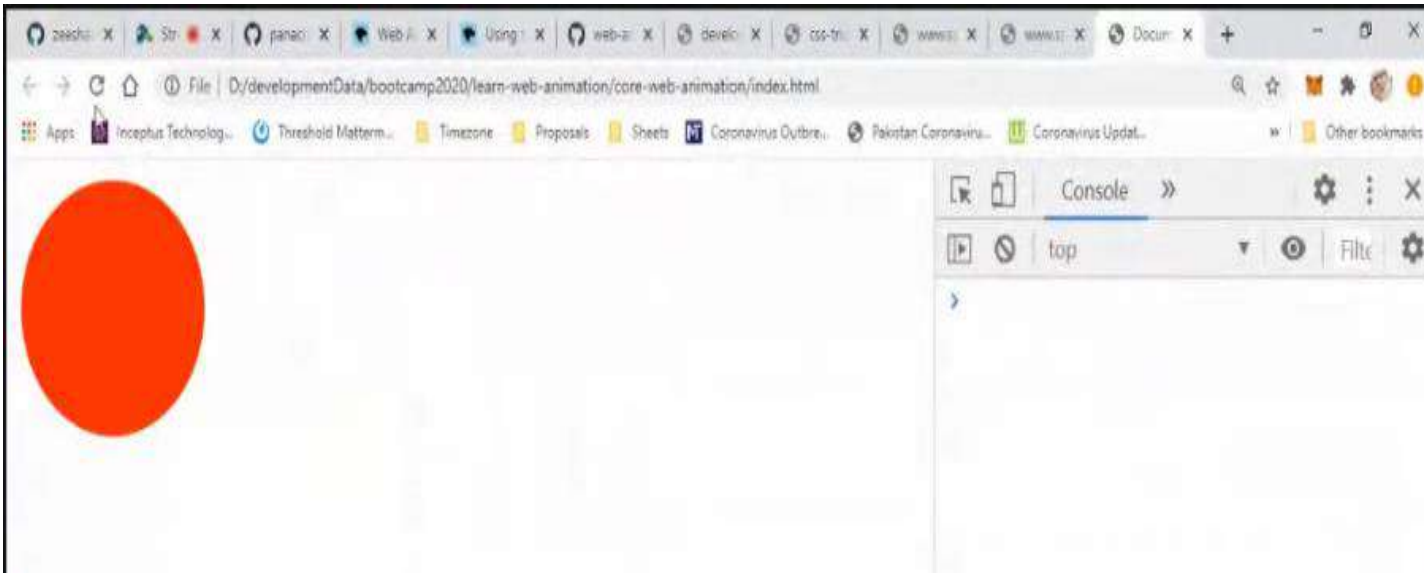
4. <u>Project 4A and 4B Without React</u>

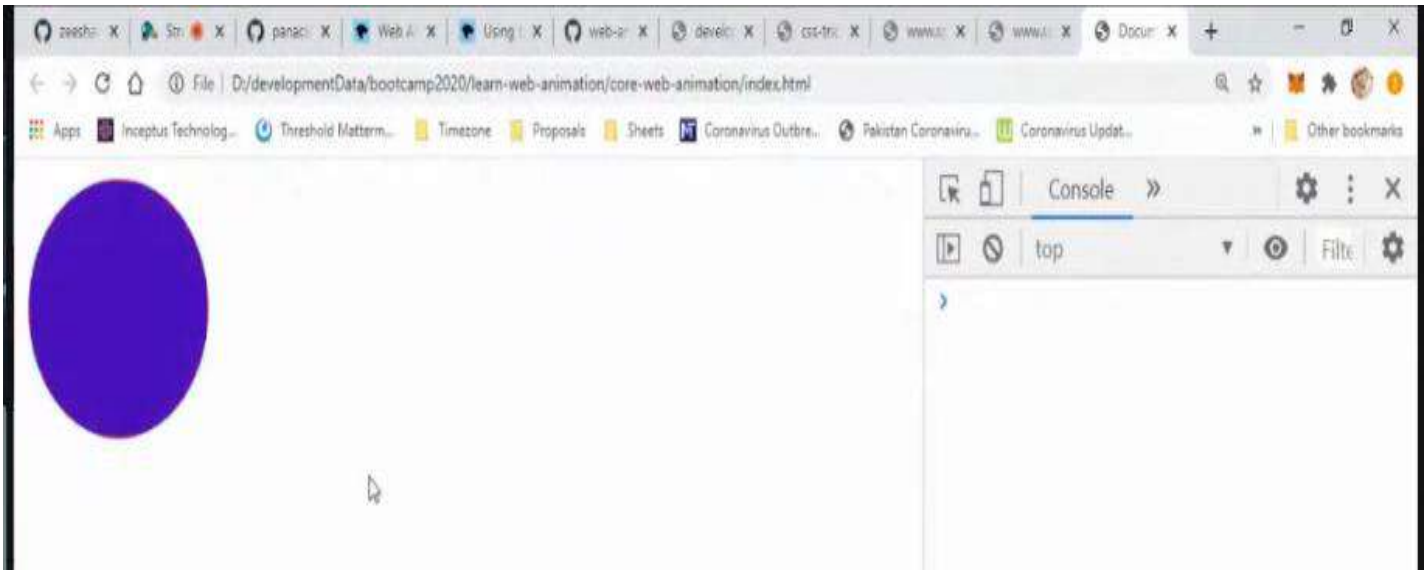
- provide shape and frames to a particular area, provide background color
- provide timing duration

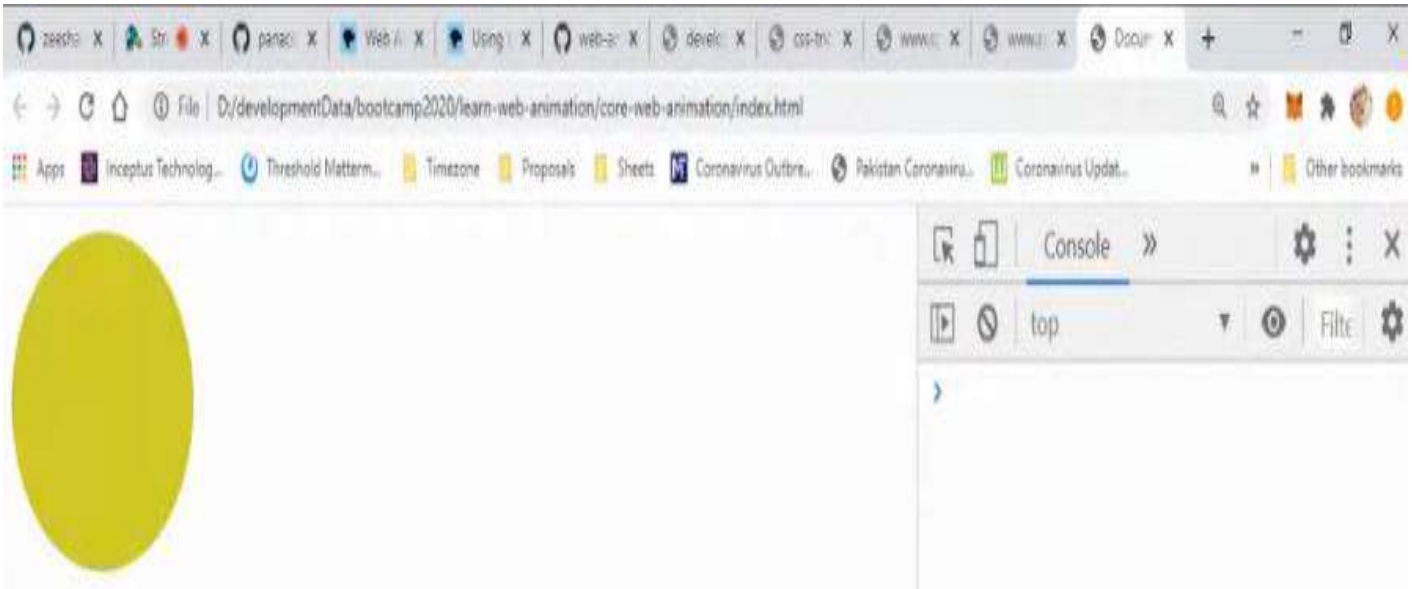
  ShapeElement.animate (frames,timing);

- Open up your browser and you will look circle with different colors transition at different times as

```
index.html
<!DOCTYPE html>
<html lang="eng">
<head>
    <title>Document</title>
  <style>

 <script>
   window.AddEventListener("load, function(){
 var ShapeElement= document.querySelector(".shape");
 var frames =[
   {background: "red"},
   {background: "blue"},
   {background:"yellow"},
 ];
var timing={
  duration:2000
}
ShapeElement.animate(frames,timing);
  </script>
 </head>
 <body>

<div class="shape">
  Hello World
 </div>
</body>

</html>
```


File | D:/developmentData/bootcamp2020/learn-web-animation/core-web-animation/index.html
Console
top

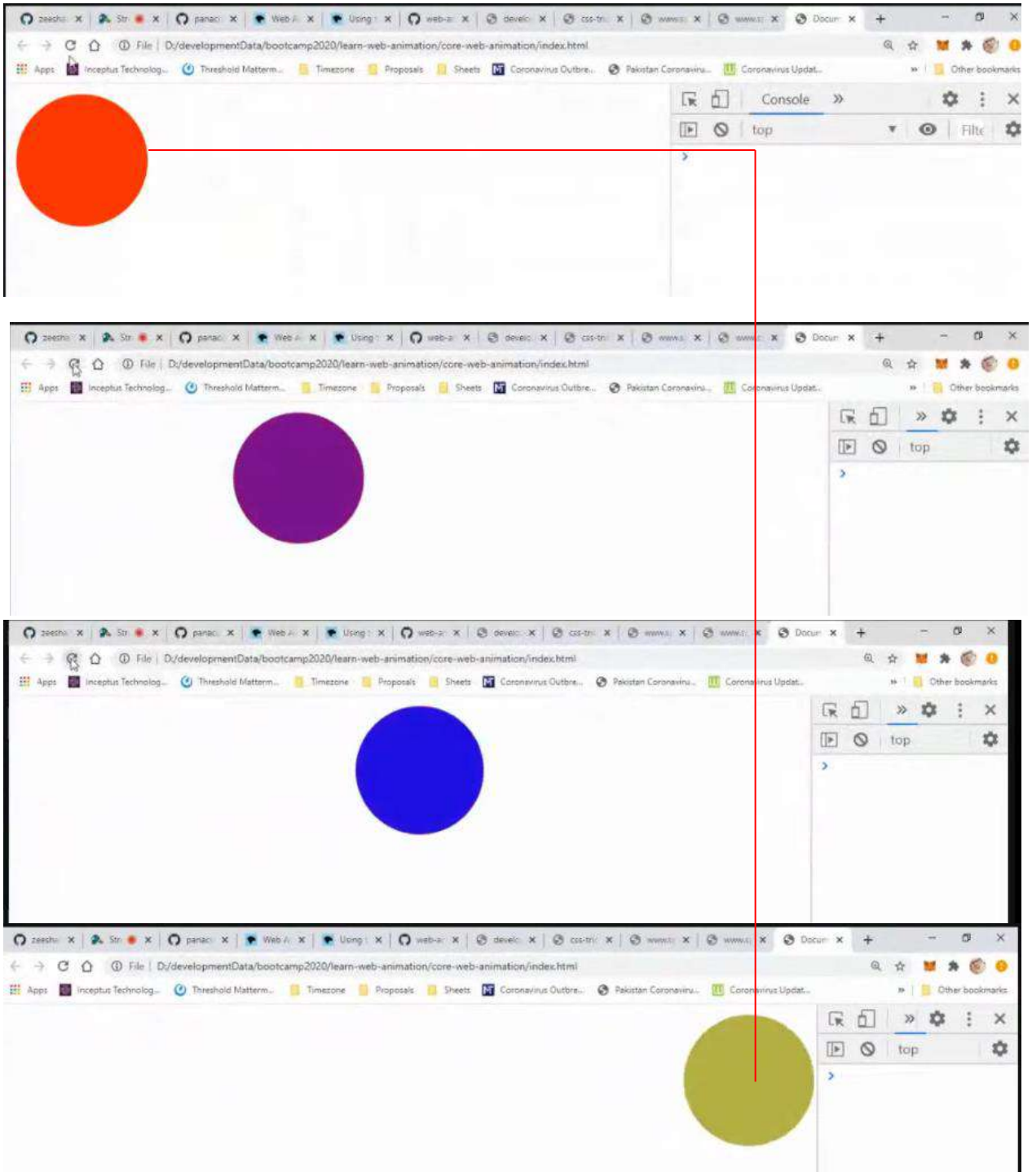

File | D:/developmentData/bootcamp2020/learn-web-animation/core-web-animation/index.html
Console
top


File | D:/developmentData/bootcamp2020/learn-web-animation/core-web-animation/index.html
Console
top

- animate the property of translate (circule moves from 0px to 600px left to right

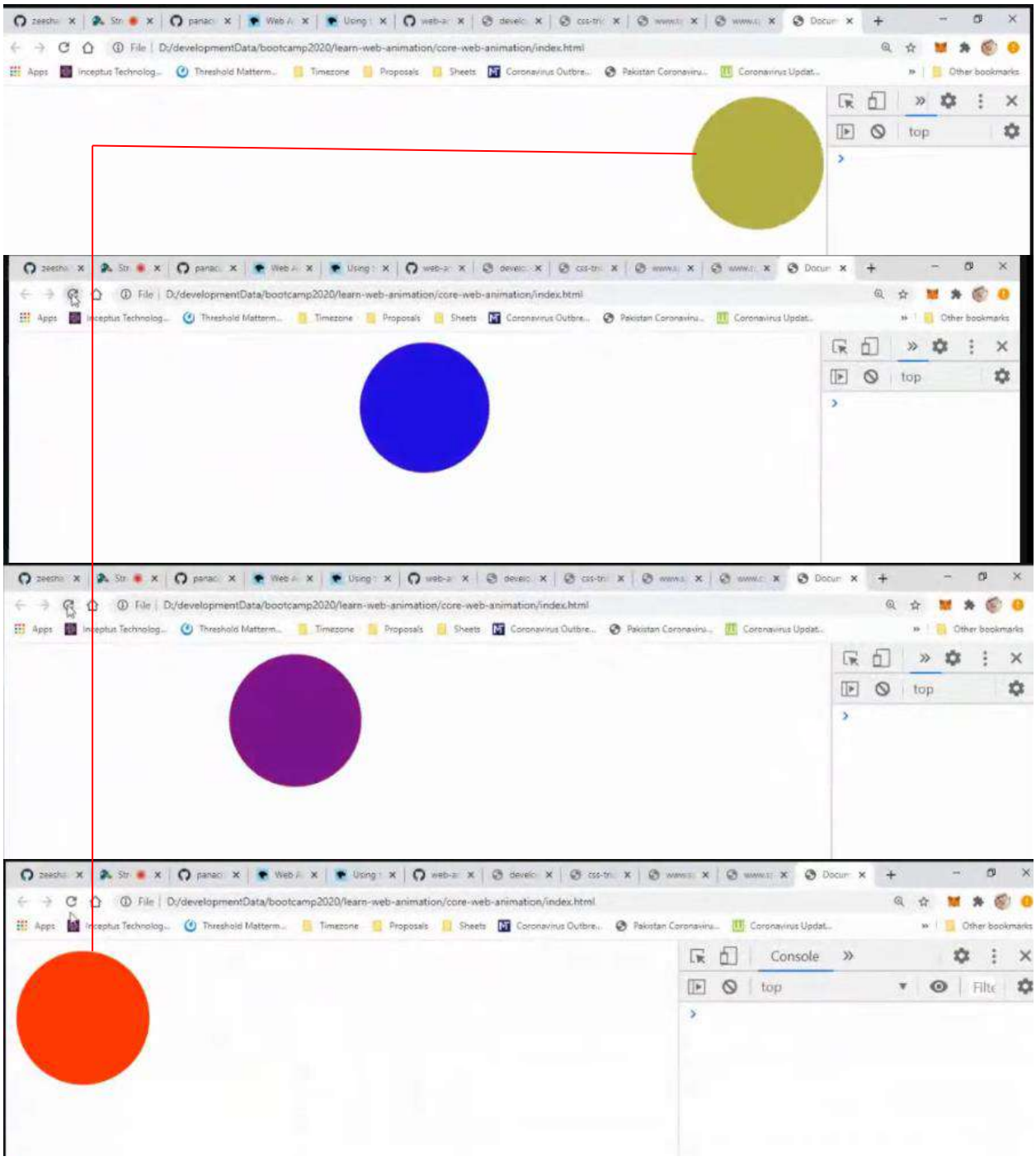
```
var ShapeElement= document.querySelector(".shape");
var frames =[
  {background: "red", transform: "translate(0px)"},
  {background: "red"},
  {background:"yellow", transform: "translate(600px)"},
```

```
index.html
<!DOCTYPE html>
<html lang="eng">
<head>
    <title>Document</title>
  <style>




  <script>
    window.AddEventListener("load, function(){
  var ShapeElement= document.querySelector(".shape");
  var frames =[
    {background: "red", transform: "translate(0px)"},
    {background: "red"},
   {background:"yellow", transform: "translate(600px)"},

 ];
   var timing={
            duration:2000,
            iteration:infinity ,
             direction; “reverse”
}
ShapeElement.animate(frames,timing);
   </script>
  </head>
```



forward moving with transition in color

- **Similarly, we can provide the direction as reverse, alternate as well as can decrease or increase the duration of this transition**



- **Since the accelaration of bject is not in control , to control animation we will ad the function**

```
var shapAnimate =ShapeElement.animate(frames,timing);
```

- **With buttons in div sections as**

```
<div class="shape">

<button class="play">Play</button>

<button class="pause">Pause</button>

<button class="reverse">Reverse</button>

<button class="updatePalybackRate">UpdatePalybackRate</button>
```

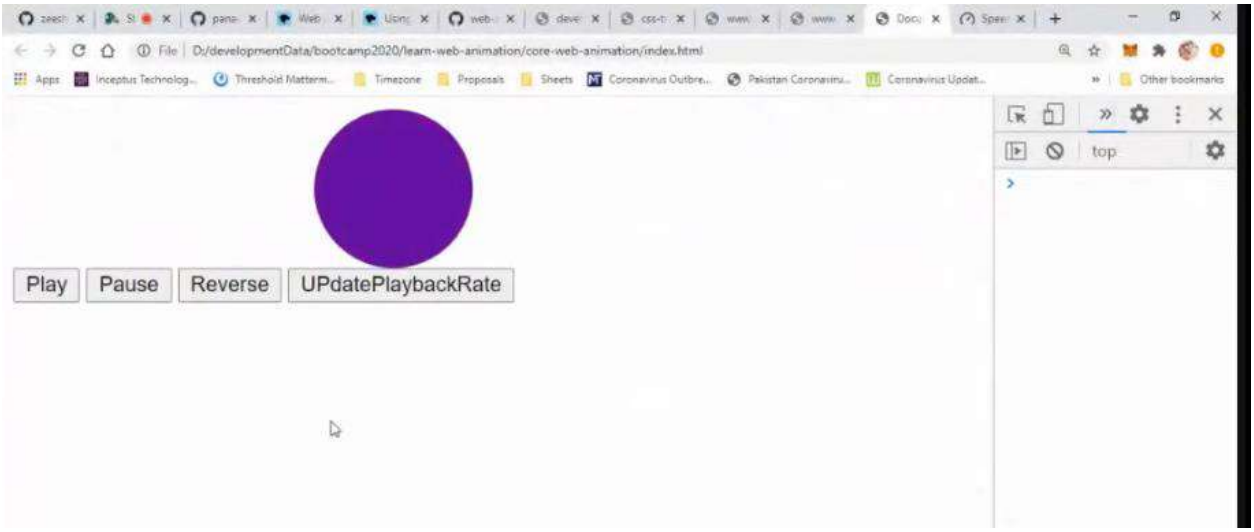
- **Implement these control put event as**

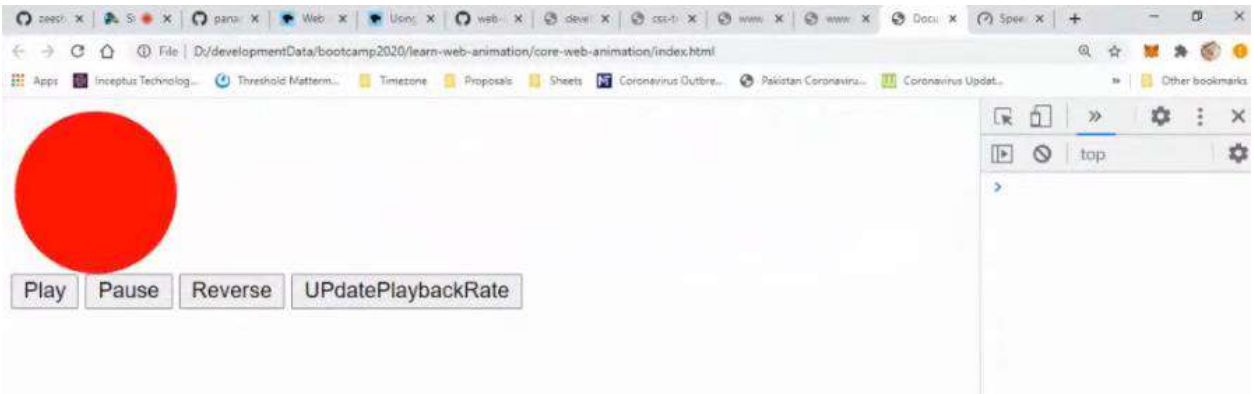
```
document.querySelector(".pause").addEventListner("click",function{}{
  shapeAnimate.pause();
    });
var shapAnimate =ShapeElement.animate(frames,timing);
 document.querySelector(".reverse").addEventListner("click",function{}{
  shapeAnimate.reverse();
     });
var shapAnimate =ShapeElement.animate(frames,timing);
 document.querySelector(".updatePalybackRate").addEventListner("click",function{}{
  shapeAnimate.updatePalybackRate();
     });
```

This function will help to control the playback rate with the speed of 1*1 or whatever you set

```
  shapeAnimate.updatePalybackRate(ShapeAnimate.playbackRate* 1.2);
```

```
index.html
<!DOCTYPE html>
<html lang="eng">
<head>
     <title>Document</title>
   <style>


  <script>
    window.AddEventListener("load, function(){
  var ShapeElement= document.querySelector(".shape");
  var frames =[
    {background: "red", transform: "translate(0px)"},
    {background: "red"},
    {background:"yellow", transform: "translate(600px)"},
 ];
var timing={
   duration:2000
      iteration:infinity ,
      direction: “reverse"
}
 var shapAnimate =ShapeElement.animate(frames,timing);
             document.querySelector(".pause").addEventListner("click",function{}{
  shapeAnimate.pause();
});
var shapAnimate =ShapeElement.animate(frames,timing);
 document.querySelector(".reverse").addEventListner("click",function{}{
  shapeAnimate.reverse();
});
var shapAnimate =ShapeElement.animate(frames,timing);
 document.querySelector(".updatePalybackRate").addEventListner("click",function{}{
  shapeAnimate.updatePlaybackRate(ShapeAnimate.playbackRate* 1.2);
});
   </script>
  </head>
 <body>

<div class="shape">
<button class="play">Play</button>
<button class="pause">Pause</button>
<button class="reverse">Reverse</button>
<button class="updatePlaybackRate">UpdatePalybackRate</button>
   Hello World
 </div>

</body>
</html>
```





- lets put an image Gif of lion that has been already downloaded in our pc
- in div section provide the class name of that img GIFas

```
<img class="shape"  src="lion.gif">
```

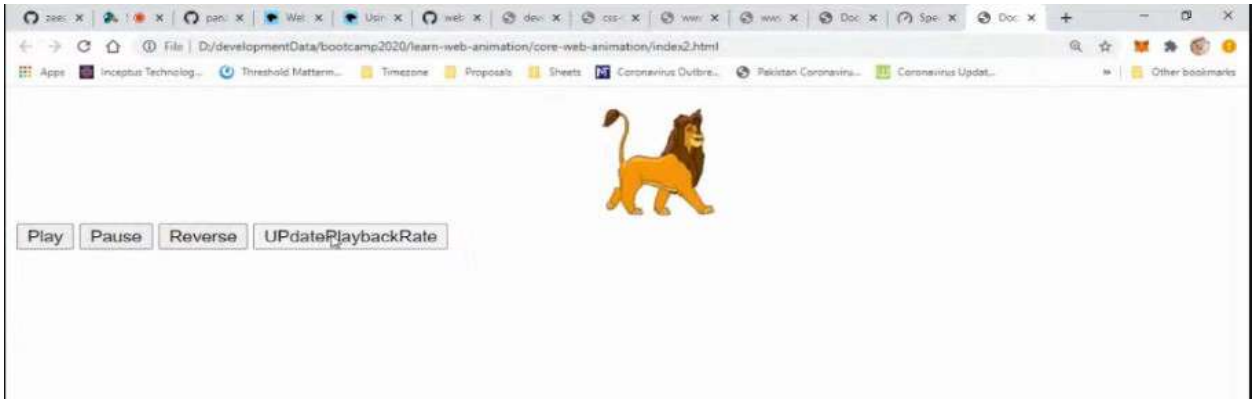
- Change background color

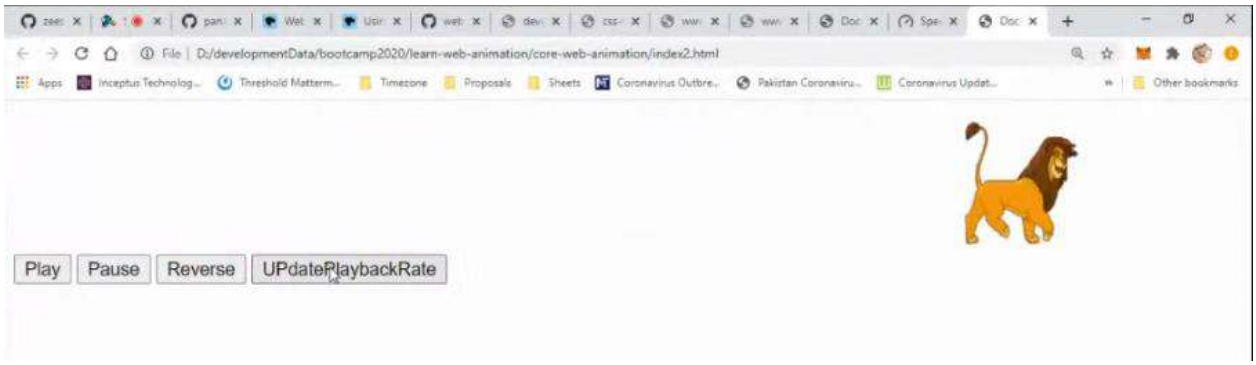
```
{transform:"translateX(0px)"},
{transform: translateX(600px)"},
```

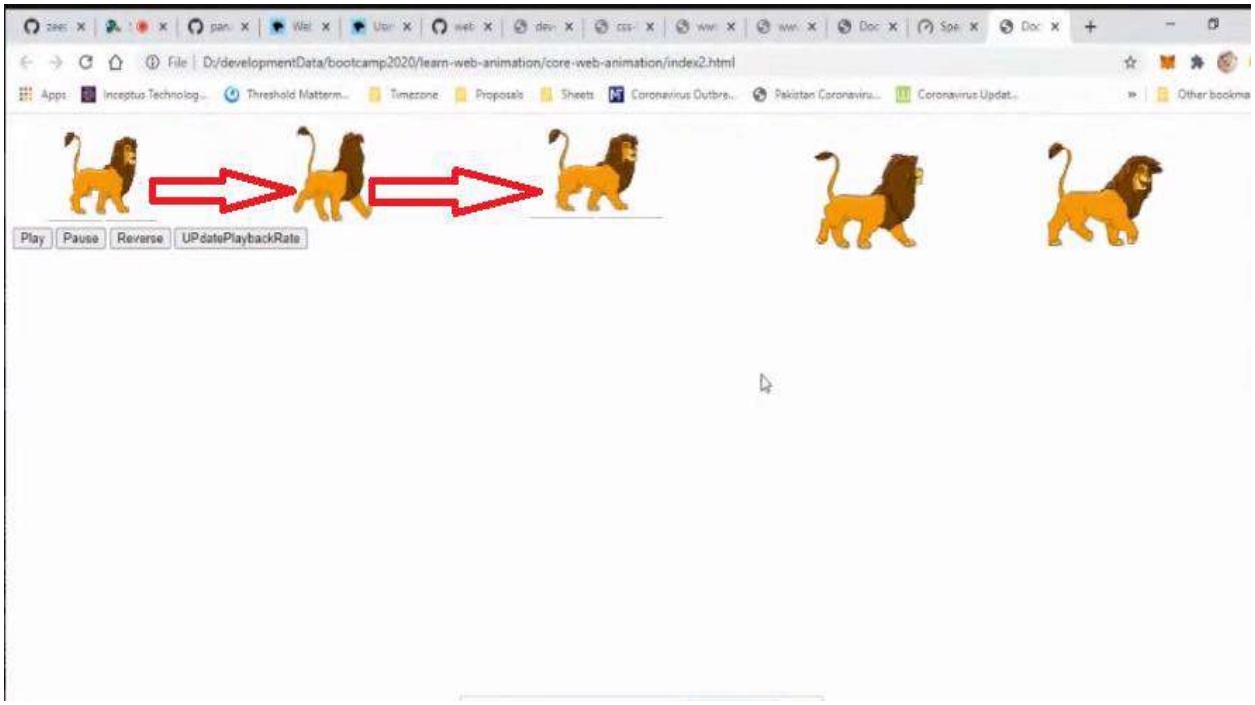
- Provide hight and width

Now you can make lion gift move, pause, reverse and speed up its playback rate

```
index.html
<!DOCTYPE html>
<html lang="eng">
<head>
     <title>Document</title>
   <style>
   .shape{
              hight=100px;
            width=100px;


  <script>
    window.AddEventListener("load, function(){
  var ShapeElement= document.querySelector(".shape");
  var frames =[
    {transform:"translateX(0px)"},
    {transform: translateX(600px)"},

 ];
    var timing={
       duration:2000
      iteration:infinity ,
}

var shapAnimate =ShapeElement.animate(frames,timing);
 document.querySelector(".play").addEventListner("click",function{}{
  shapeAnimate.play();
});
var shapAnimate =ShapeElement.animate(frames,timing);
 document.querySelector(".pause").addEventListner("click",function{}{
  shapeAnimate.pause();
});
var shapAnimate =ShapeElement.animate(frames,timing);
 document.querySelector(".reverse").addEventListner("click",function{}{
  shapeAnimate.reverse();
});
var shapAnimate =ShapeElement.animate(frames,timing);
 document.querySelector(".updatePalybackRate").addEventListner("click",function{}{

  shapeAnimate.updatePalybackRate(ShapeAnimate.playbackRate* 1.2);
});

  </script>
  </head>
 <body>
```


File | D:/developmentData/bootcamp2020/learn-web-animation/core-web-animation/index2.html
Play
Pause
Reverse
UPdatePlaybackRate


File | D:/developmentData/bootcamp2020/learn-web-animation/core-web-animation/index2.html
Play
Pause
Reverse
UPdatePlaybackRate


File | D:/developmentData/bootcamp2020/learn-web-animation/core-web-animation/index2.html
Play
Pause
Reverse
UPdatePlaybackRate

Session 12(A): Project 4A and 4B without React

1. **Web Animation API Without Using React**
   - open up a new window, create a new folder, and code all projects there.
   - copy any image like clouds and birds
   - add CSS and javascript
   - now in the div add src image of bird Gif that has been downloaded by you
   - now in main.css add body and wrapper height and width plus add birds location as absolute to maintain it in the center
   - add background-color

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset ="UFT-8">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <title>Animation App</title>
    <link rel="stylesheet" href="main.cc">
</head>
</body>
   <div class="wrapper">
   <div class="mybird">
   <img src="images/mybird.gif"/>

  </div>


  <!--Add js->
  <script src ="main.js"></script>

 </body>

</html>
```

main.css

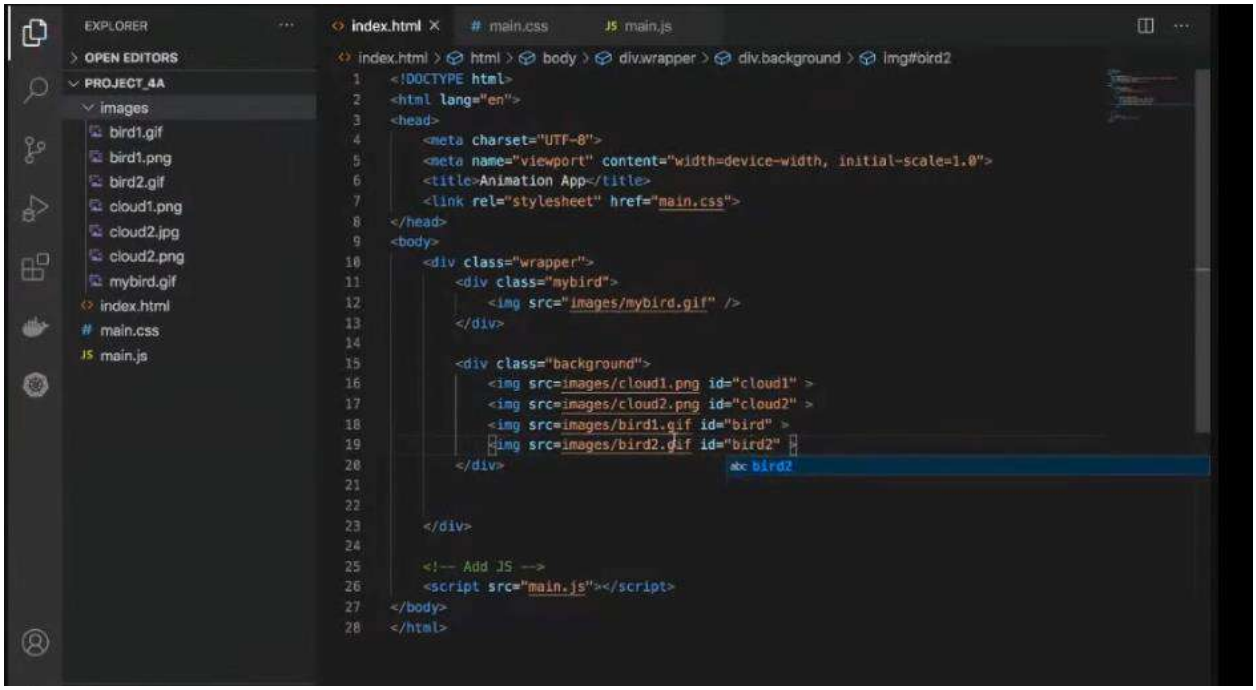
```
body{
    width:100%:
    height:100%;
      background-color:3b5d9e8;
}
.wrapper{
    width:100%;
    height:100%;
}
.mybird{
   position:absolute;
  top:20%;
 left:35%;
}
```

- add more object in the background with eagle in index.html as
- provide each object id as well

```
<div class="background">
        <img src=images/cloud1.png id="cloud1">
        <img src=images/cloud2.png id="cloud2">
        <img src=images/bird1.png id="bird1">
        <img src=images/bird2.png id="bird2">
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset ="UFT-8">
   <meta name="viewport" content="width=device-width,initial-scale=1.0">
   <title>Animation App</title>
   <link rel="stylesheet" href="main.cc">
</head>
</body>
  <div class="wrapper">
  <div class="mybird">
          <img src="images/mybird.gif"/>
 </div>
         <div class="background">
           <img src=images/cloud1.png id="cloud1">
           <img src=images/cloud2.png id="cloud2">
           <img src=images/bird1.png id="bird1">
           <img src=images/bird2.png id="bird2">
         </div>


 <!--Add js->
 <script src ="main.js"></script>

</body>

</html>
```



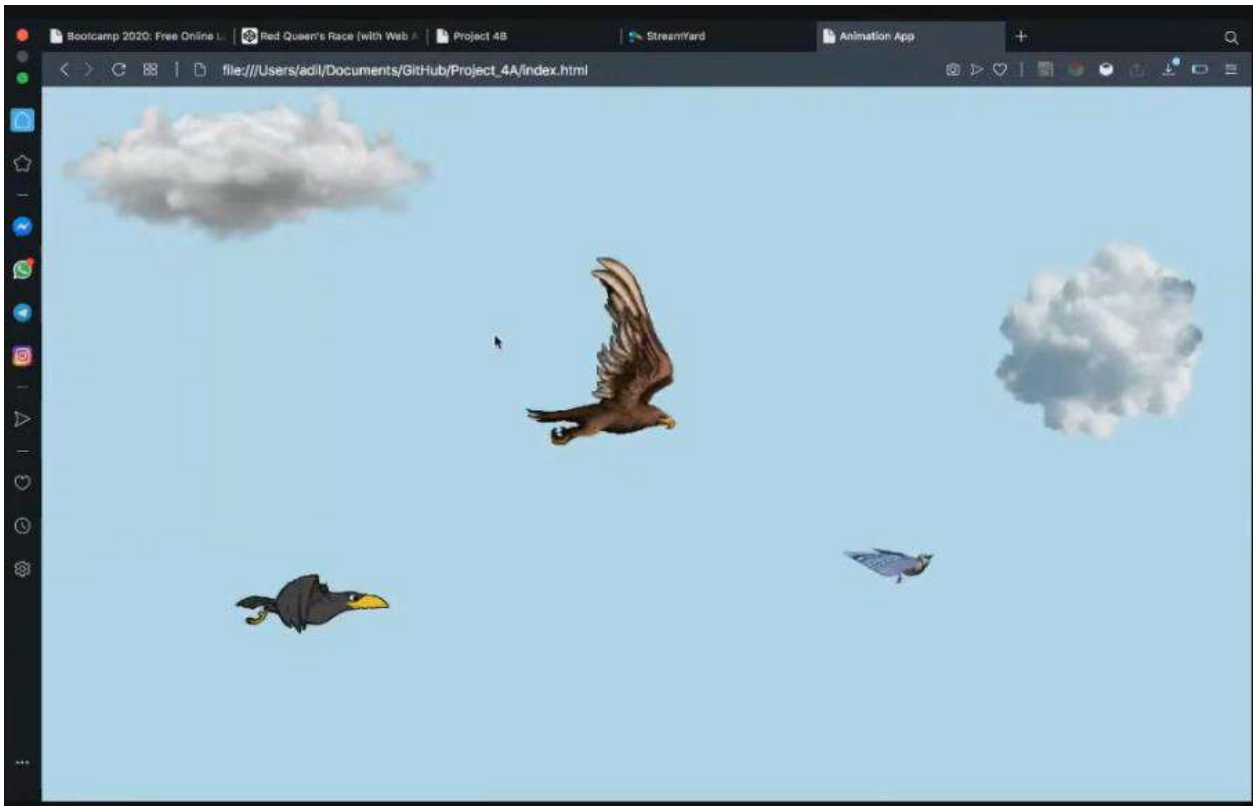
- in CSS.main provide the object CSS style of your choice as;

```
#cloud1{
   position:absolut;
   width:100px;
   top:0;
   left: 5%;
 }
#cloud2{
   position:absolut;
   width:300px;
   top:20%;
   left: 80%;
 }
#bird1{
   position:absolut;
   width:100px;
   top:68%;
   left:78%;
 }
#bird2{
   position:absolut;
   width:100px;
   top:68%;
   left:15%;
 }
```

main.css

```
body{
     width:100%:
     height:100%;
       background-color:3b5d9e8;
}

.wrapper{
     width:100%;
     height:100%;
}

.mybird{
    position:absolute;
   top:20%;
 left:35%;
}

#cloud1{
    position:absolut;
    width:100px;
    top:0;
    left: 5%;
 }

#cloud2{
    position:absolut;
    width:300px;
    top:20%;
    left: 80%;
 }

#bird1{
    position:absolut;
    width:100px;
    top:68%;
    left:78%;
 }

#bird2{
    position:absolut;
    width:100px;
    top:68%;
    left:15%;
 }
```

- **This is how clouds within sky blue background, 2 clouds and 3 birds of provided CSS style properties have been shown**



Background Movement

- create 2 variables
  - time
  - frame
- after that create a variable background from index.html by using element id

  as var background= document.getElementById("background");

- next, create a variable for background element movement to animate the background that will be just an illusion and pass two arguments of frame and timing;

  var backgroundMovement=background.animate(sceneryFrames,sceneryTiming)

- set background movement timing
- in order call a move make a variable of flyfaster or of your choice name nd provide it a function (in this function update the palyback rate)

  var flyFaster=function(){
  backgroundMovement.playbackRate=1.1 (10 times increase playback rate upon clicking)

- update background time

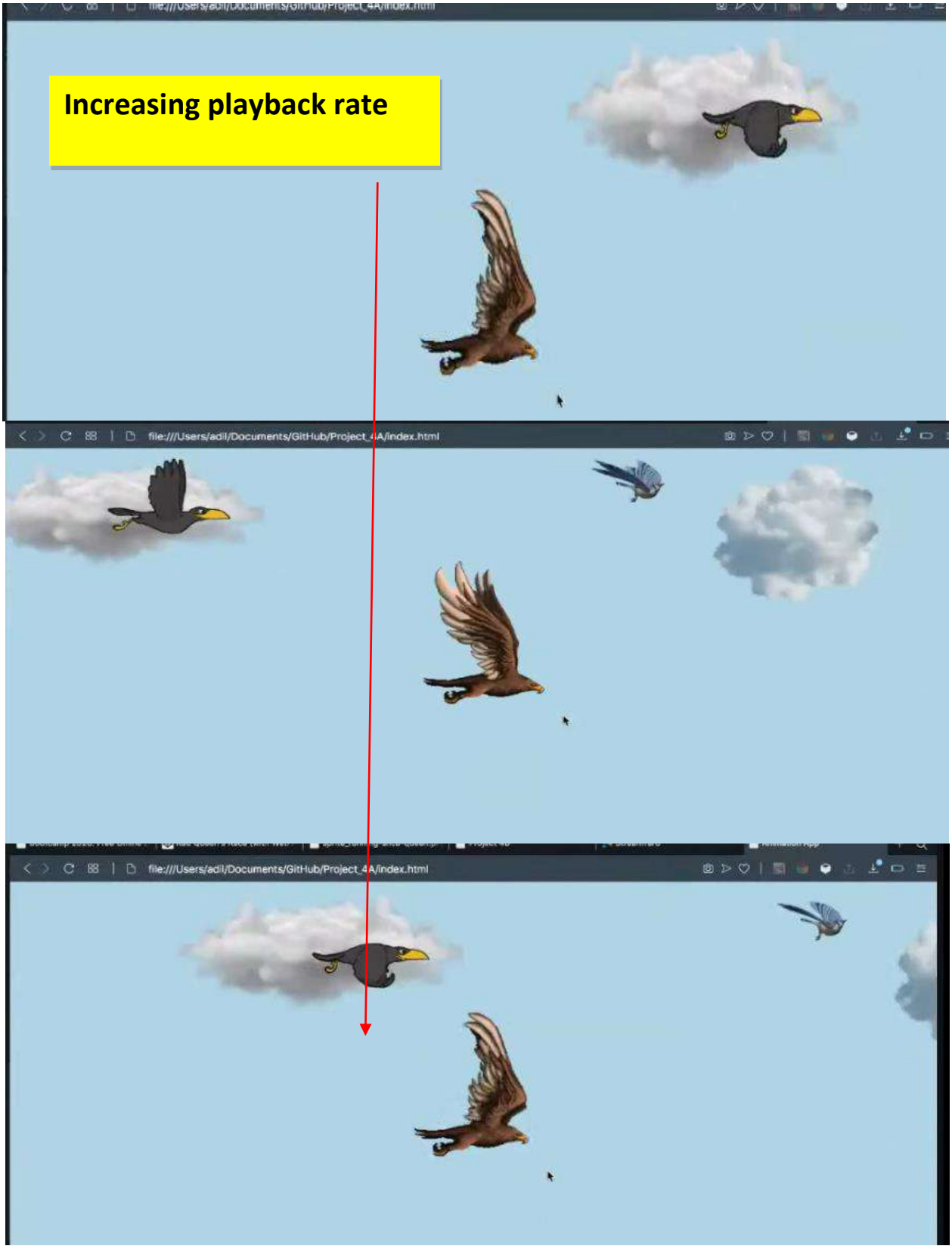
```
backgroundMovement.currentTime=backgroundMovement.effect.getTiming().duration/2
```

- also provide id to the background in div section in index.html

```
<div class="background" id=" background">
    <img src=images/cloud1.png id="cloud1">
    <img src=images/cloud2.png id="cloud2">
    <img src=images/bird1.png id="bird1">
    <img src=images/bird2.png id="bird2">
  </div>
```

main.js

```
var sceneryFrames=[
  {transform: 'translatex(100)'},
  {transform: 'translatex(100)'},
 }:
 var sceneryTiming=[
  duration:36000,
  iteration:infinity,
  playbakRate:-2
};
 var background= document.getElementById("background");
 var
backgroundMovement=background.animate(sceneryFrames,sceneryTiming):
backgroundMovement.currentTime=backgroundMovement.effect.getTiming().dur
ation/2

 var flyFaster=function(){

  backgroundMovement.playbackRate=1.1

}
```

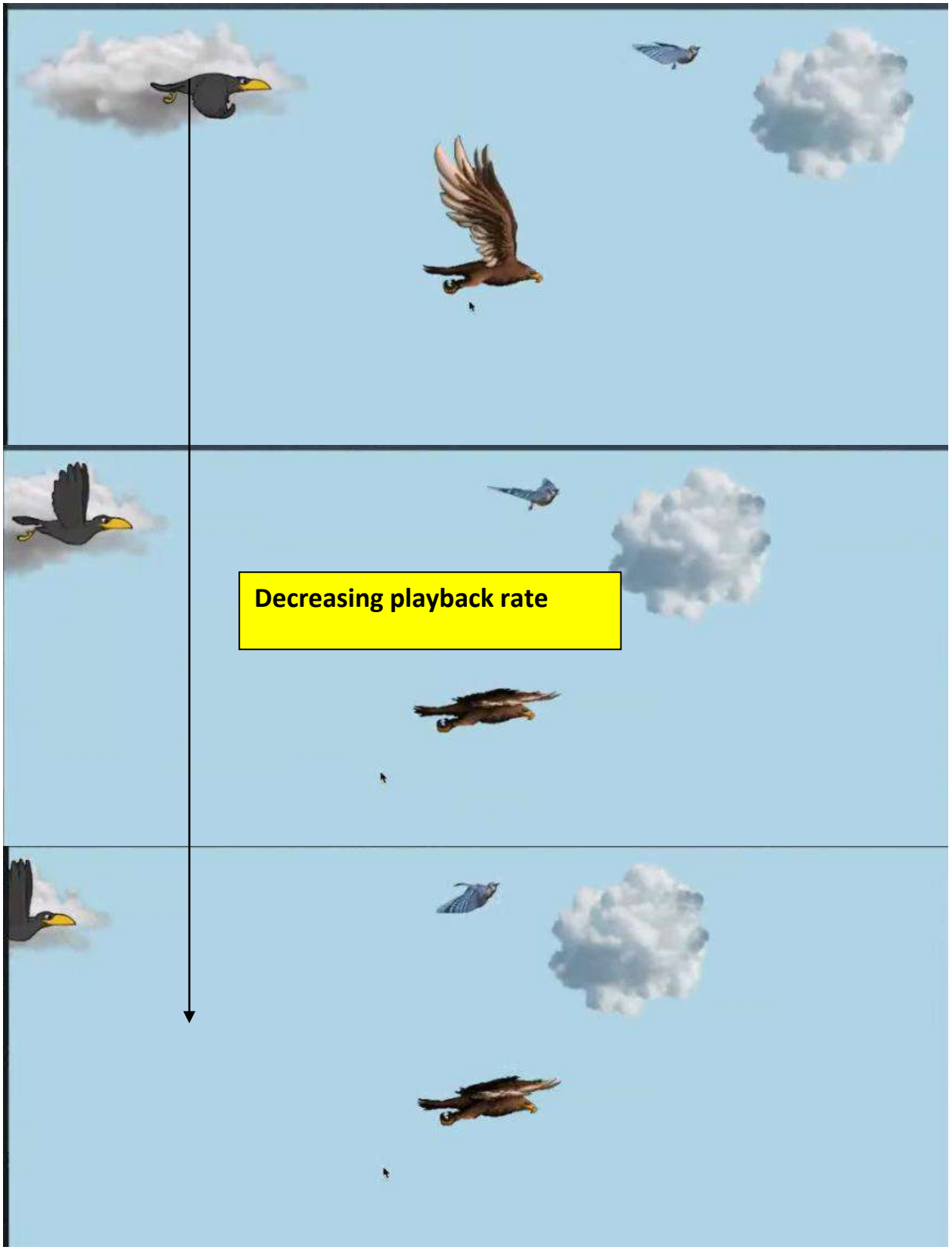

Increasing playback rate
file:///Users/adil/Documents/GitHub/Project_4A/index.html
file:///Users/adil/Documents/GitHub/Project_4A/index.html

- To decrease down the playback rate of background use set intervals

```
setInterval(function(){
   if(bacgroundMovement.playbackRate>0.4){
      backgroundMovement.playbackRate*=.9
  }
},3000)
```

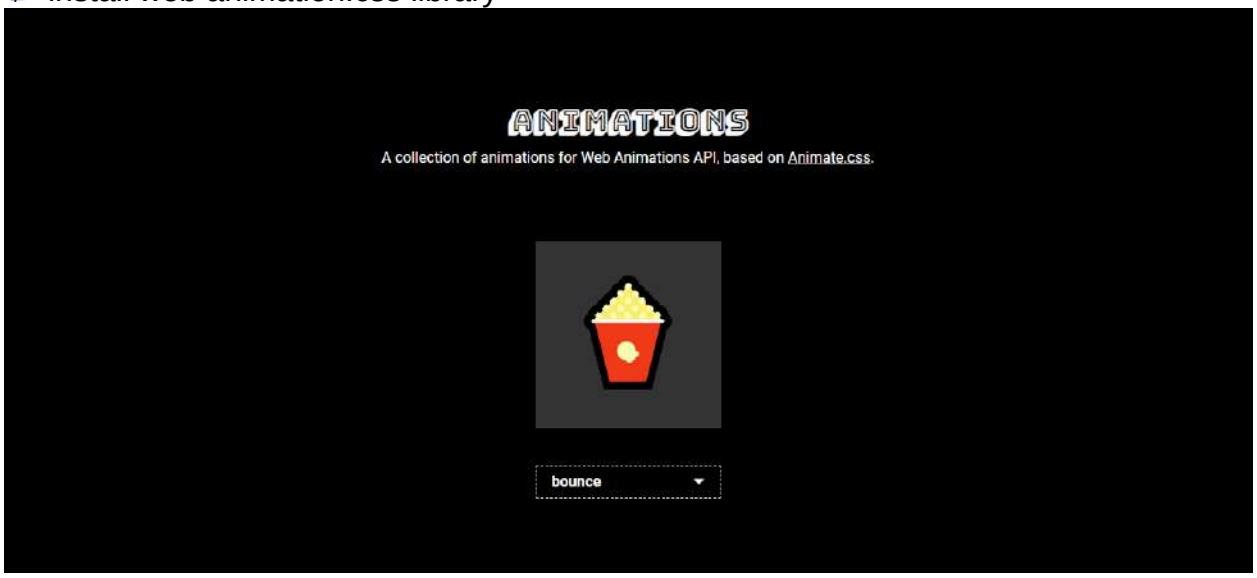
main.js

```
var sceneryFrames=[
   {transform: 'translatex(100)'},
   {transform: 'translatex(100)'},
 }:
 var sceneryTiming=[
  duration:36000,
  iteration:infinity,
  playbakRate:-2
};
 var background= document.getElementById("background");
 var
backgroundMovement=background.animate(sceneryFrames,sceneryTiming)
:
backgroundMovement.currentTime=backgroundMovement.effect.getTiming(
).duration/2

setInterval(function(){
   if(bacgroundMovement.playbackRate>0.4){
      backgroundMovement.playbackRate*=.9
  }
},3000)
 var flyFaster=function(){
   backgroundMovement.playbackRate=1.1
}
```


Decreasing playback rate

2. Web Animation Using App React

- install web animation.css library



- Install with npm:

```
$ npm install animate.css --save
with yarn:

$ yarn add animate.css
```

We gonna use hook WeAnimation const {ref, playState, getAnimation]=useWebAnimation Using

- **useRef**
- **playState**
- **getAnimation**

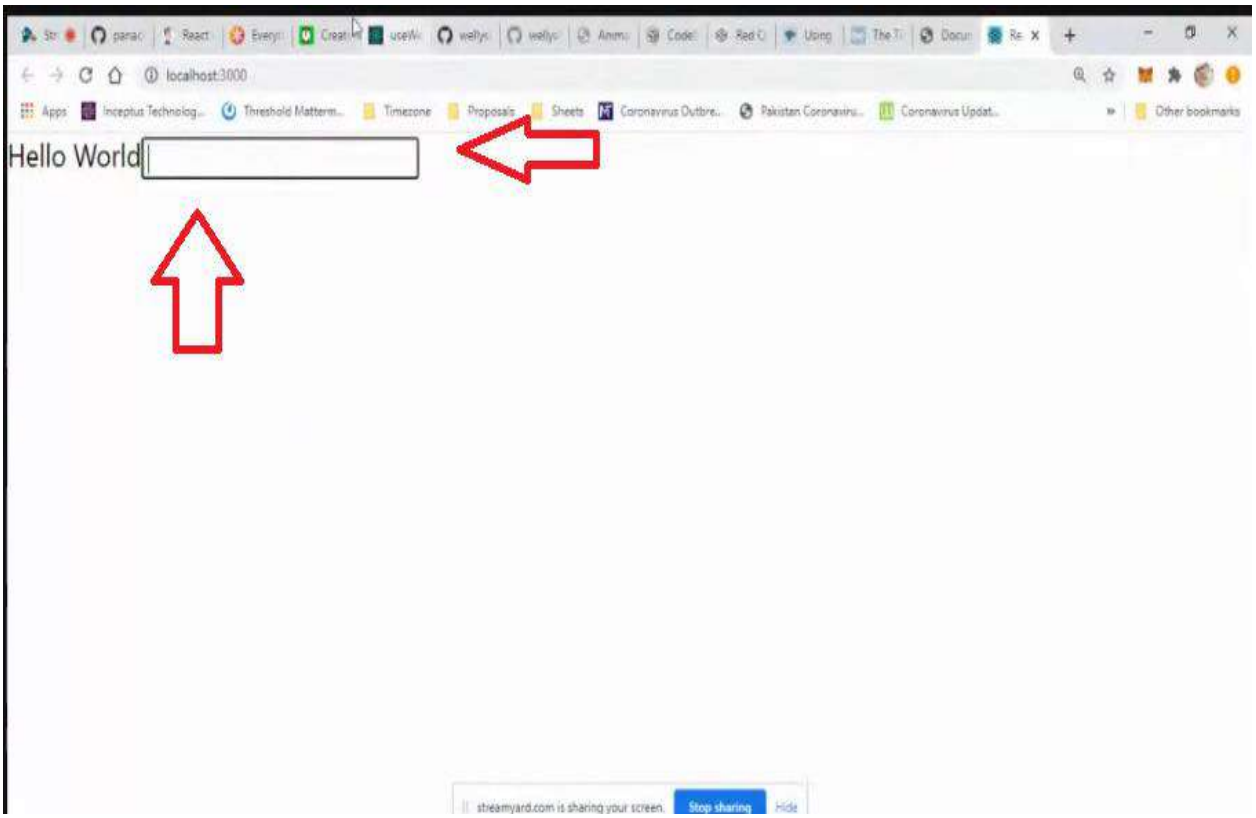
- using useRef let you not to rerender your elements when you update any element

  allow you direct Access to dom functionality of your element

- now use useEffect for focusing effect to get cursor in the box

```
useEffect(()=>{
  inputEl.current.focus();
 },[])
```

```
import React, {useRef} from 'react';
import './App.css';



  function App(){
  const inputEl=useRef(null);

          useEffect(()=>{
            inputEl.current.focus();
   },[])
       return(
        <div>
       Hello World
      <input ref=(inputEl) type="text"/>
     </div>
    );
}
export default App;
```



- now import "import useWebAnimationfrom "@wellsyshen/use-web-animation"
- use keyframes with transform of 0px to 500px or of your choice

- in timing provide

```
timing:{
        duration:3000
       iteration:infinity
        direction:"alternate",
        easing"ease in"
```

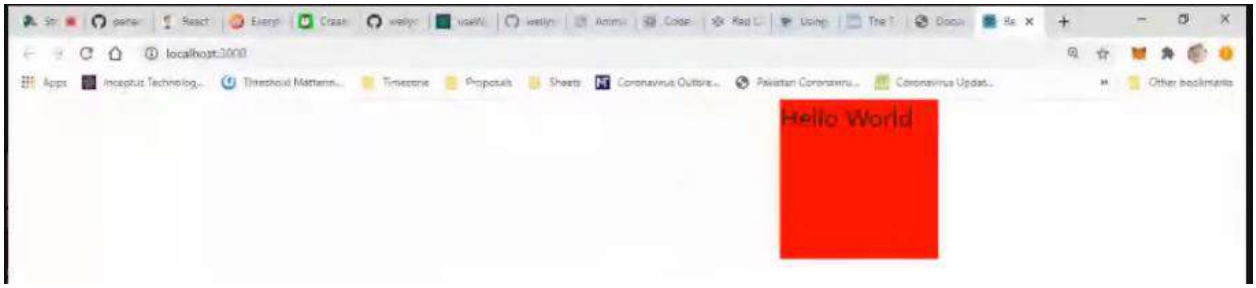
- look at display it HelloWorld will move forward backward with ifnity iteration and duration os 3000

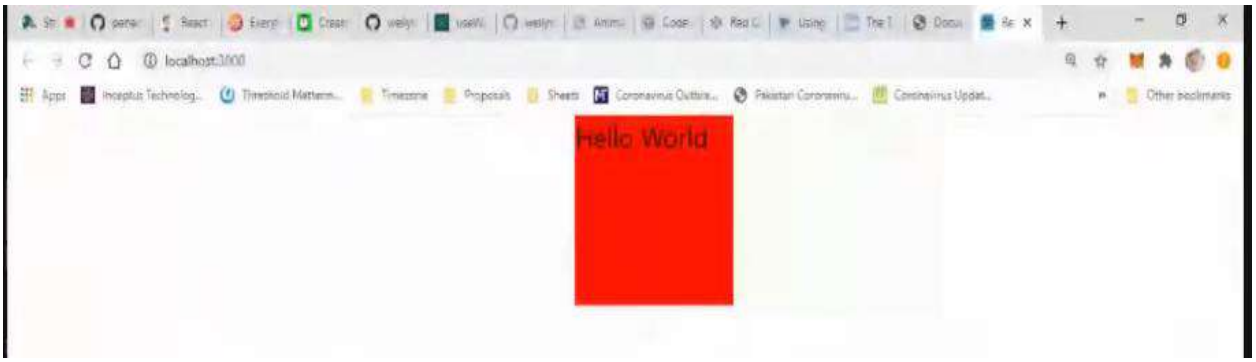
```
import React, {useRef} from 'react';
import './App.css';
import useWebAnimationfrom "@wellsyshen/use-web-animation

function App(){
   const{ref,playState,getnimation]=useWebAnimation
    keyframes:[
       {transform: "translate(0px)"}
       {transform: "translate(500px)"}
  ],
 timing:{
    duration:3000
   iteration:infinity
    direction:"alternate",
    easing"ease in"
 }
});
   return(
     <div ref=(ref)>
   Hello World
   <input ref=(inputEl) type="text"/>
   </div>
  );
}

export default App;
```

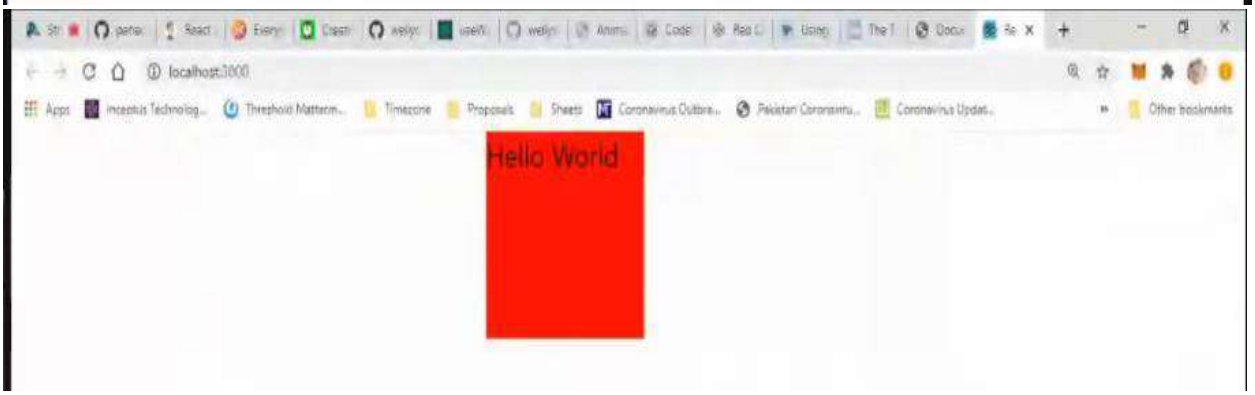
localhost:3000
Hello World
Hello World
Hello World
Hello World

- **Now apply css using hook , provide all properties like the background color, width and height insisde the ref as** <div ref=(ref)style={{background:"red",width:"100",height:"100">

```
  import React, {useRef} from 'react';
import './App.css';
import useWebAnimationfrom "@wellsyshen/use-web-animation

function App(){
   const{ref,playState,getnimation]=useWebAnimation
    keyframes:[
       {transform: "translate(0px)"}
       {transform: "translate(500px)"}
  ],
 timing:{
    duration:3000
     iteration:infinity
    direction:"alternate",
    easing"ease in"
 }
});
   return(
     <div ref=(ref)style={{background:"red", width:"100",height:"100">
   Hello World
   <input ref=(inputEI) type="text"/>
   </div>
  );
}

export default App;
```





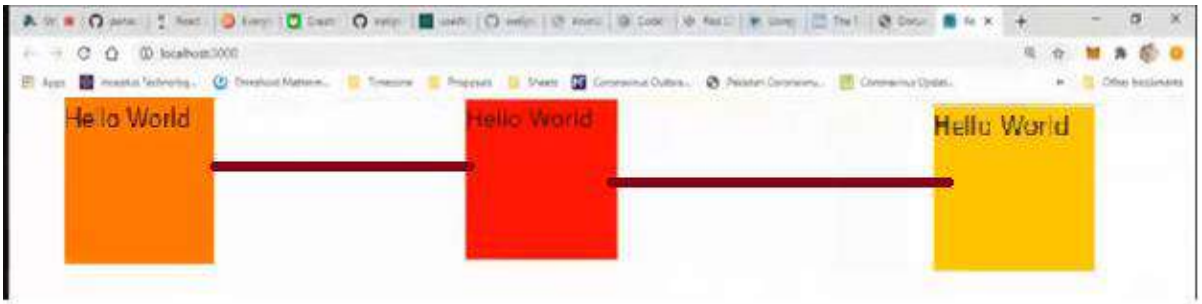


- Now in keyframe provide different tranform colors as ;

  {transform: "translate(0px)", background:"red"}

  {transform: "translate(500px)", background: "yellow"}

- **For animation after steps** easing"step<4, end)"



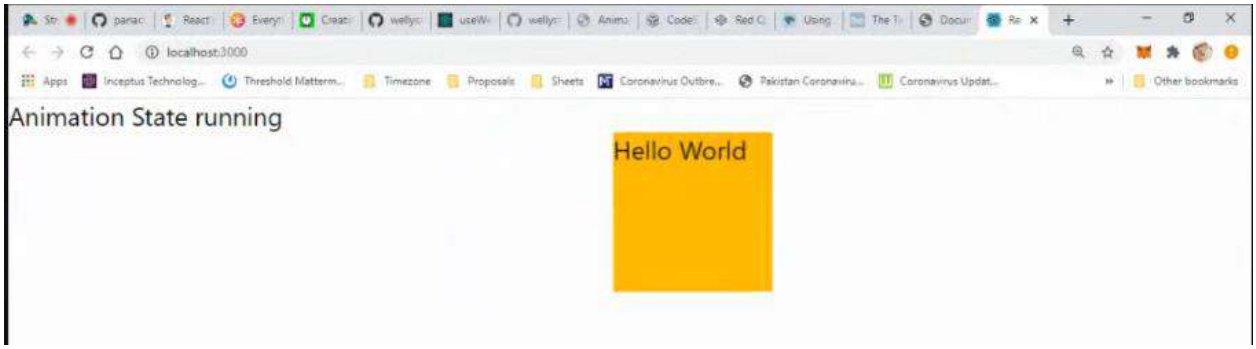
You can use Animation State as a playstate by keeping limited iterationi.e “4”

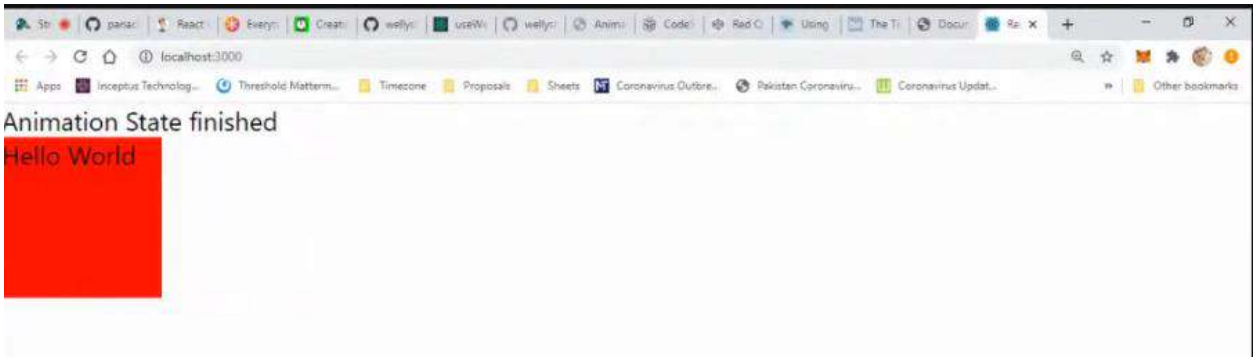
“ Animation State {playState} ”

```
function App(){
  const{ref,playState,getAnimation]=useWebAnimation
   keyframes:[
      {transform: "translate(0px)", background:"red"}
      {transform: "translate(500px)", background: "yellow"}
 ],
 timing:{
   duration:3000
   iteration:4
   direction:"alternate",
   easing"step<4,end)"

 }
});
  return(
   <div>Animation State {playState}</div>
    <div ref=(ref)style={{background:"red",width:"100",height:"100">
  Hello World
  <input ref=(inputEl) type="text"/>
  </div>
 );
}

export default App;
```

- **After 4 iterations the object will stop running and it will display**

  “Animation State finished”

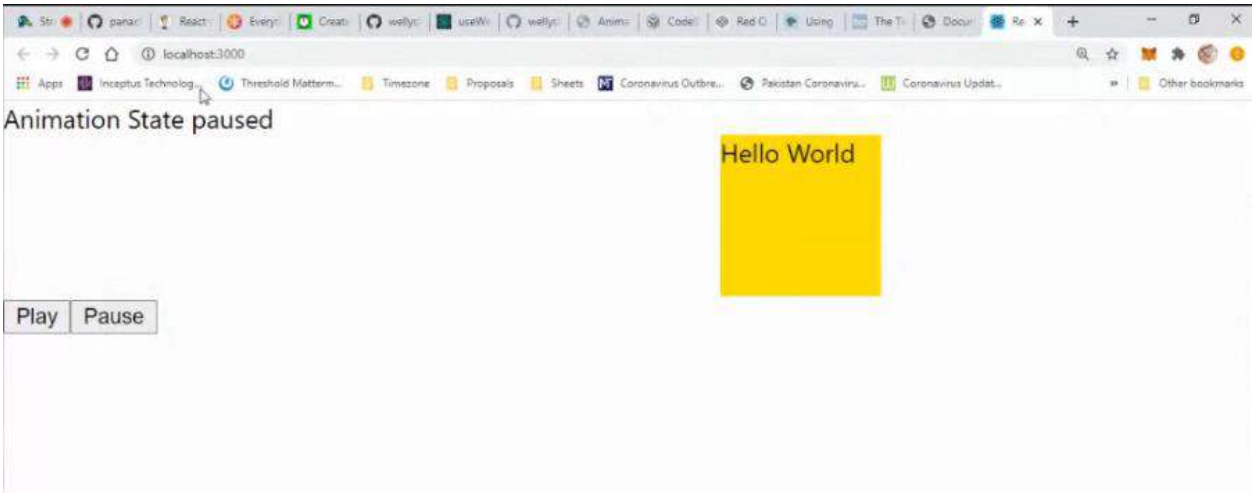


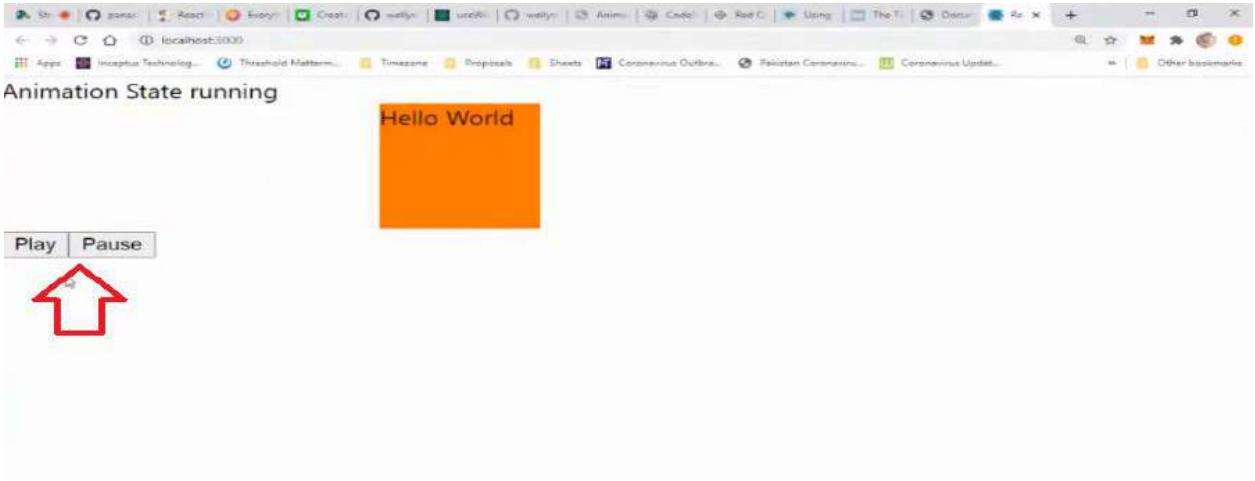


- Now use **getAnimation** to display the play and pause control over the object by usin button tag as

```
  </div>
 <button onClick={()=>{
 getAnimation().play();
}}>play</button>

 <button onClick={()=>{

 getAnimation().pause();

}}>pause</button>
```

```
import React, {useRef} from 'react';
import './App.css';
import useWebAnimationfrom "@wellsyshen/use-web-animation
function App(){
   const{ref,playState,getnimation]=useWebAnimation
    keyframes:[
       {transform: "translate(0px)", background:"red"}
       {transform: "translate(500px)", background: "yellow"}
  ],
 timing:{
    duration:3000
    iteration:infinity
    direction:"alternate",
    easing"step<4,end)"
  }
});
   return(
    <div>Animation State {playState}</div>
     <div ref=(ref)style={{background:"red",width:"100",height:"100">
   Hello World
              </div>
             <button onClick={()=>{
             getAnimation().play();
            }}>play</button>

             <button onClick={()=>{
             getAnimation().pause();
            }}>pause</button
  );
}
export default App;
```





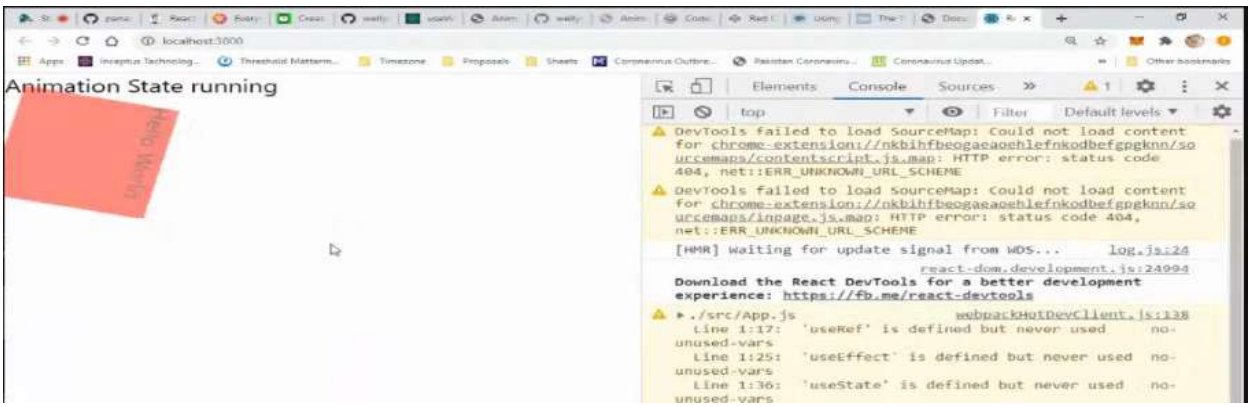
- **you can bounce , rotate ,spin the object by importing in the webAnimation hook and and creating its object as**

  import useWebAnimationfrom,{rotateOut} "@wellsyshen/use-web-animation

  const {ref,playState}=useWebAnimation({...rotateOut});

```
import React, {useRef} from 'react';
import './App.css';
import useWebAnimationfrom,{rotateOut} "@wellsyshen/use-web-animation
function App(){

const {ref,playState}=useWebAnimation({...rotateOute});

  /* const{ref,playState,getnimation]=useWebAnimation

    keyframes:[

      {transform: "translate(0px)", background:"red"}
```

```
  {transform: "translate(500px)", background: "yellow"}
  ],
 timing:{
    duration:3000
    iteration:infinity
    direction:"alternate",
    easing"step<4,end)"
  },
onUpdate:({playState, animate,animation})=>{
console.log("Play State",plyState);
console.log("animate",animate);
console.log("animation",animation);
 }
});/*
   return(
    <div>Animation State {playState}</div>
     <div ref=(ref)style={{background:"red",width:"100",height:"100">
   Hello World

   </div>
  /*<button onClick={()=>{
  getAnimation().play();
 }}>play</button>
  <button onClick={()=>{

  getAnimation().pause();
 }}>pause</button>
</div>

  );
}
export default App;
```

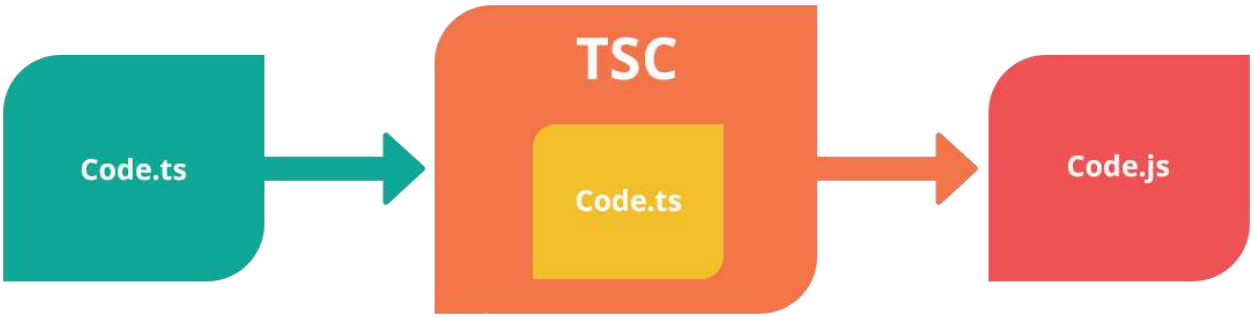


Session 13: Programming TypeScript

1.Defining TypeScript

TypeScript is an object-oriented programming language developed and maintained by the Microsoft Corporation. It is a superset of JavaScript and contains all of its elements.

TypeScript follows the OOPS concept and with the help of TSC (TypeScript Compiler), we can convert Typescript code (.ts file) to JavaScript (.js file)



- TypeScript simplifies JavaScript code, making it easier to read and debug.
- TypeScript is open-source.
- TypeScript provides highly productive development tools for JavaScript IDEs and practices, like static checking.
- TypeScript makes code easier to read and understand.
- With TypeScript, we can make a huge improvement over plain JavaScript.
- TypeScript gives us all the benefits of ES6 (ECMAScript 6), plus more productivity.
- TypeScript can help us to avoid painful bugs that developers commonly run into when writing JavaScript by type-checking the code.
- The powerful type system, including generics.
- TypeScript is nothing but JavaScript with some additional features.
- Structural, rather than nominal.
- TypeScript code can be compiled as per ES5 and ES6 standards to support the latest browser.
- Aligned with ECMAScript for compatibility.
- Starts and ends with JavaScript.
- Supports static typing.
- TypeScript will save developers time.
- TypeScript is a superset of ES3, ES5, and ES6.

Additional Features of TypeScript

- Functions with optional parameters.
- Functions with REST parameters.
- Generics support.
- Modules support.

Transpiler

A transpiler is a compiler like a program, which converts ES6 JavaScript code to ES5 JavaScript code so that it can run in browsers. When the transpiler sees an expression that uses a language feature that needs to be translated, it produces a logically equivalent expression. The produced expressions can be very similar, or very different from the source expression.

2. Transpiler vs Compiler

Transpilers, or **source-to-source compilers**, are tools that read source code written in one programming language and produce the equivalent code in another language that has a similar level of abstraction. A good example of a transpiler is the **Typescript transpiler**, which converts Typescript code to JavaScript. **Babel transpiler** can also be used for ES6 JS code to ES5 JS code.

Whereas compilers also convert the code from one language to another language but both languages are very different in abstraction level. e.g. `.java` to `.class` file compilation.

ES6 vs ES5

To understand the transpiler, you must understand first the difference between ES6 and ES5 JavaScripts. ES6 (ECMAScript 6) is a specification for the next version of JavaScript. Some of its major enhancements include modules, class declarations, lexical block scoping, iterators and generators, promises for asynchronous programming, destructuring patterns, and proper tail calls.

Features are great but most browsers do not support this specification to date. So any UI application, written with ES6, the spec will not run in most browsers. To run these applications, this ES6 source code must be converted to supported JavaScript version ES5. ES5 is supported is almost all the browsers and is the most stable version to date.

ES6 – Brings 'types' to JavaScript. Bring it closer to strongly typed languages such as Java, C#.

Most browsers do not support it to date. It must be converted to ES5 to be executed in browsers.

ES5 – Vanilla JavaScript which we have been writing all these years.

The Compiler

The way TypeScript works are unusual compared to other mainstream languages like JavaScript or Java, so we must be on the same page before we go any further.

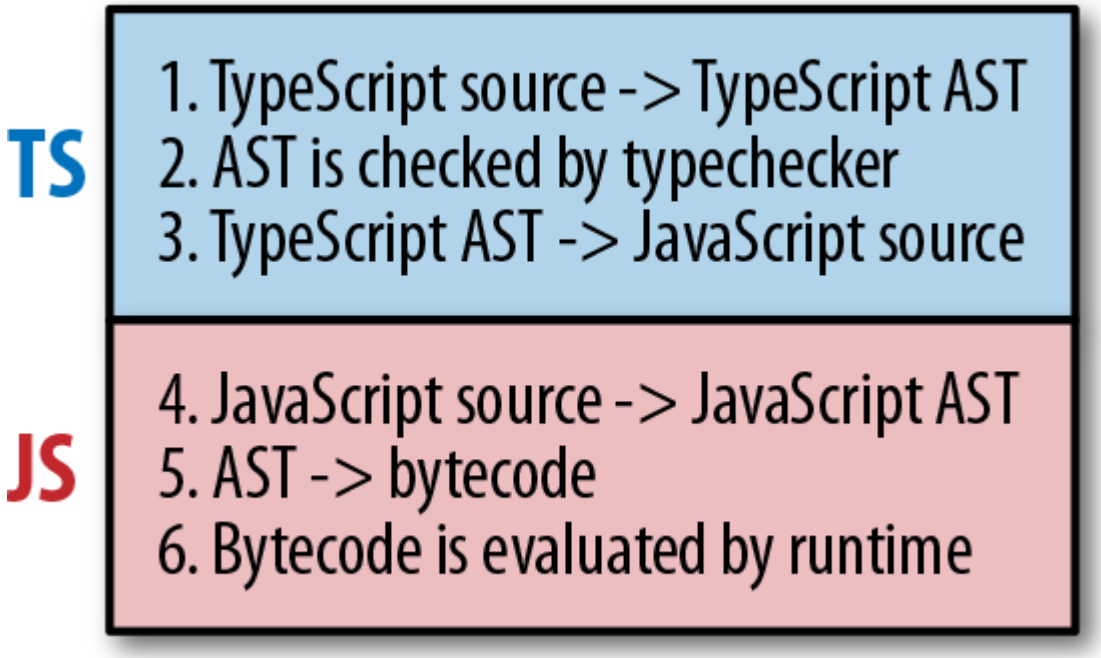
Let's start broad: programs are files that contain a bunch of text written by you, the programmer. That text is parsed by a special program called a compiler, which transforms it into an abstract syntax tree (AST), a data structure that ignores things like whitespace, comments, and where you stand on the tabs versus spaces debate. The compiler then converts that AST to a lower- level representation called bytecode. You can feed that bytecode into another program called a runtime to evaluate it and get a result. So when you run a program, what you're doing is telling the runtime to evaluate the bytecode generated by the compiler from the AST parsed from your source code. The details vary, but for most languages, this is an accurate high-level view.

Once again, the steps are:

1. The program is parsed into an AST.
2. AST is compiled to bytecode.
3. Bytecode is evaluated by the runtime.

Where TypeScript is special is that instead of compiling straight to bytecode, TypeScript compiles to… JavaScript code! You then run that JavaScript code like you normally would—in your browser, or with NodeJS, or by hand with a paper and pen (for anyone reading this after the machine uprising has begun).

Great question. I skipped over a crucial step: after the TypeScript Compiler generates an AST for your program—but before it emits code—it typechecks your code.



3. Why TypeScript Not java?

- installTyprscript beta 4.0 `npm install -g typescript`

  - create a file index.ts
  - create a file of index.js

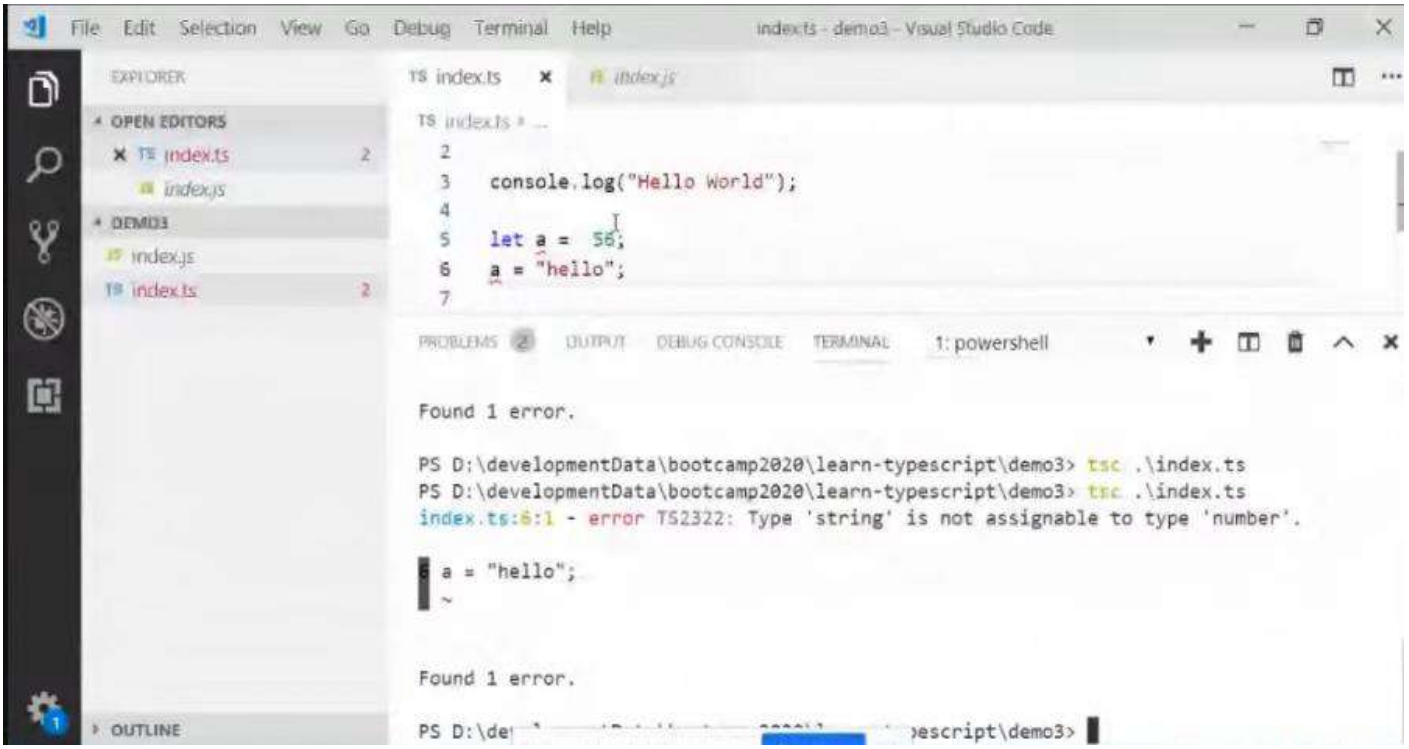
- in TypeScript there are two types of annotations:

Implicit – means that the type is inferred by TypeScript type inference system which takes responsibility away from us of writing the types:

console.log("Hello World")

let a: number = 56;

a ="45"



Explicit – generally it's manifestly adding type to our codebase. We have to exactly to know what kind of type the value is eg:

const car: string = 'Honda';

3.1. Using types to prevent programs from doing invalid things.1

Here are a few examples of things that are invalid:

- Multiplying a number and a list
- Calling a function with a list of strings when it needs a list of objects
- Calling a method on an object when that method doesn't exist on that object
- Importing a module that was recently moved

Some programming languages try to make the most of mistakes like these. They try to figure out what you meant when you did something invalid, because hey, you do what you can, right? Take JavaScript, for example:

```
3 + []              // Evaluates to the string "3"
let obj = {}
obj.foo             // Evaluates to undefined
function a(b) {
  return b/2
  }
```

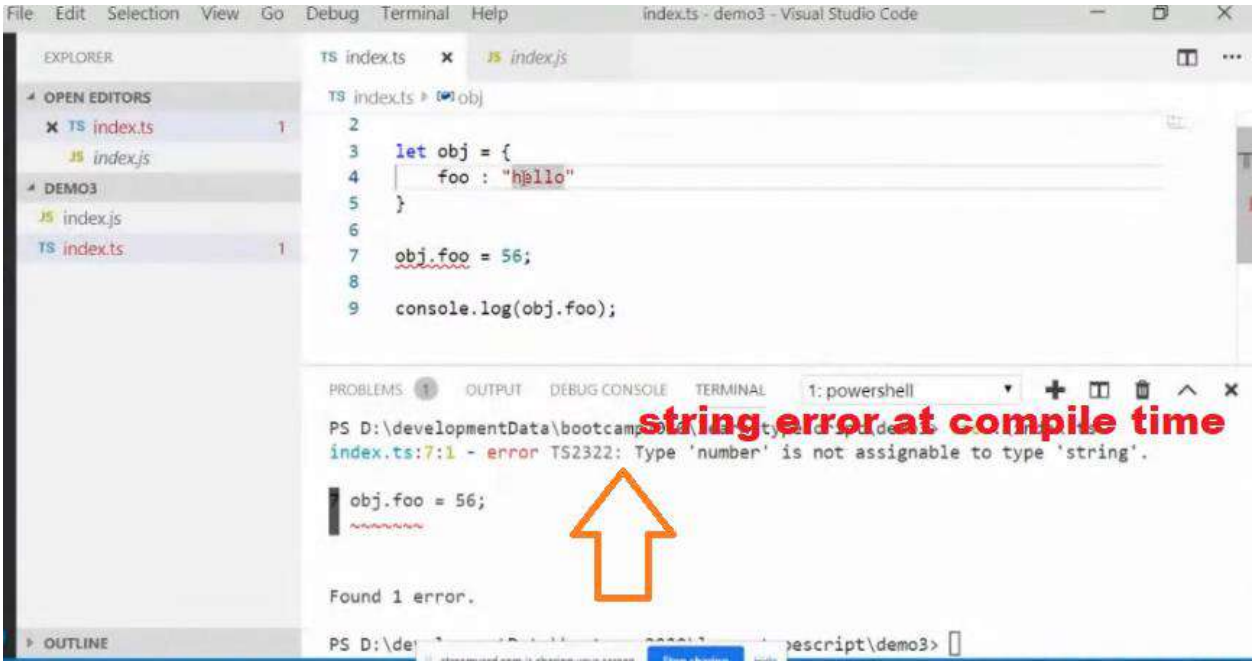
```
a("z")              // Evaluates to NaN
```

that TypeScript gives you helpful error messages is when it gives them to you: TypeScript gives you error messages in your text editor, as you type. That means you don't have to rely on unit tests or smoke tests or coworkers to catch these sorts of issues: TypeScript will catch them for you and warn you about them as you write your program. Let's see what TypeScript says about our previous example:

```
3 + []              // Error TS2365: Operator '+' cannot be applied to types '3'
                     // and 'never[]'.

let obj = {}
obj.foo             // Error TS2339: Property 'foo' does not exist on type '{}'.
function a(b: number) {
  return b / 2
}
a("z")              // Error TS2345: Argument of type '"z"' is not assignable to
                     // parameter of type 'number'.
```

Hence TypeScript Help to Collect All errors at compile time not at run time this is the super cool beauty that why we use typescript instead of javaScript

Lets us take an object of food property we can not assign it 56 because food property is a type of string that's what typescript find it as an error at its compiling time



3.2. TYPE SYSTEM

A set of rules that a type-checker uses to assign types to your program.

There are generally two kinds of type systems: type systems in which you have to tell the compiler what type everything is with explicit syntax, and type systems that infer the types of things for you automatically. Both approaches have trade-offs.1

TypeScript is inspired by both kinds of type systems: you can explicitly annotate your types, or you can let TypeScript infer most of them for you.

To explicitly signal to TypeScript what your types are, use annotations. Annotations take the form *value: type* and tell the typechecker, "Hey! You see this *value* here? Its type is *type*." Let's look

at a few examples (the comments following each line are the actual types inferred by TypeScript):

```
let a: number = 1                    // a is a number
let b: string = 'hello'              // b is a string
let c: boolean[] = [true, false] // c is an array ofbooleans
```

And if you want TypeScript to infer your types for you, just leave them off and let TypeScript get to work:

```
let a = 1                            // a is a number
let b = 'hello'                      // b is a string
let c = [true,  false]      // c is an array of booleans
```

Type system feature	JavaScript	TypeScript
When are types checked?	At runtime	**At compile time**
When are errors surfaced?	At runtime (mostly)	**At compile time (mostly)**

tsconfig.json

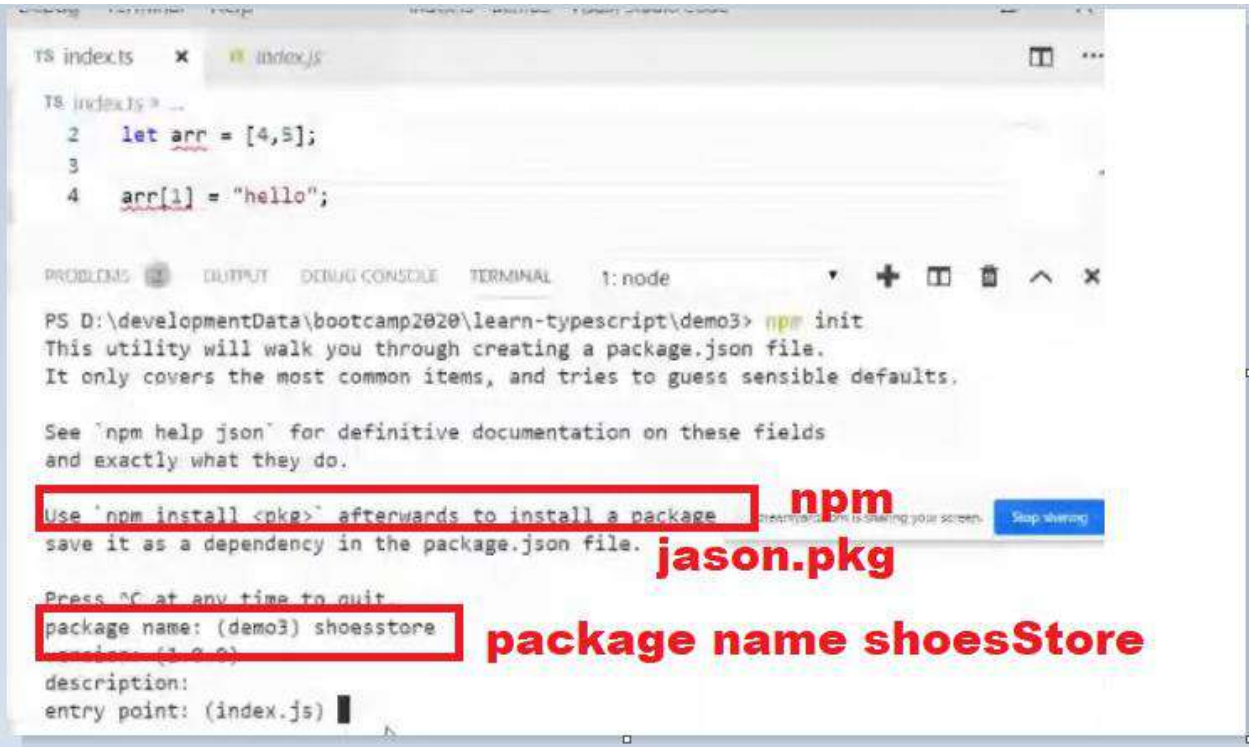
Every TypeScript project should include a file called tsconfig.json in its root directory. This tsconfig.json is where TypeScript projects define things like which files should be compiled, which directory to compile them to, and which version of JavaScript to emit.

- Create a new file called *tsconfig.json* in your root folder (touch tsconfig.json),5 then pop it open in your code editor and give it the following contents:

```
{
```

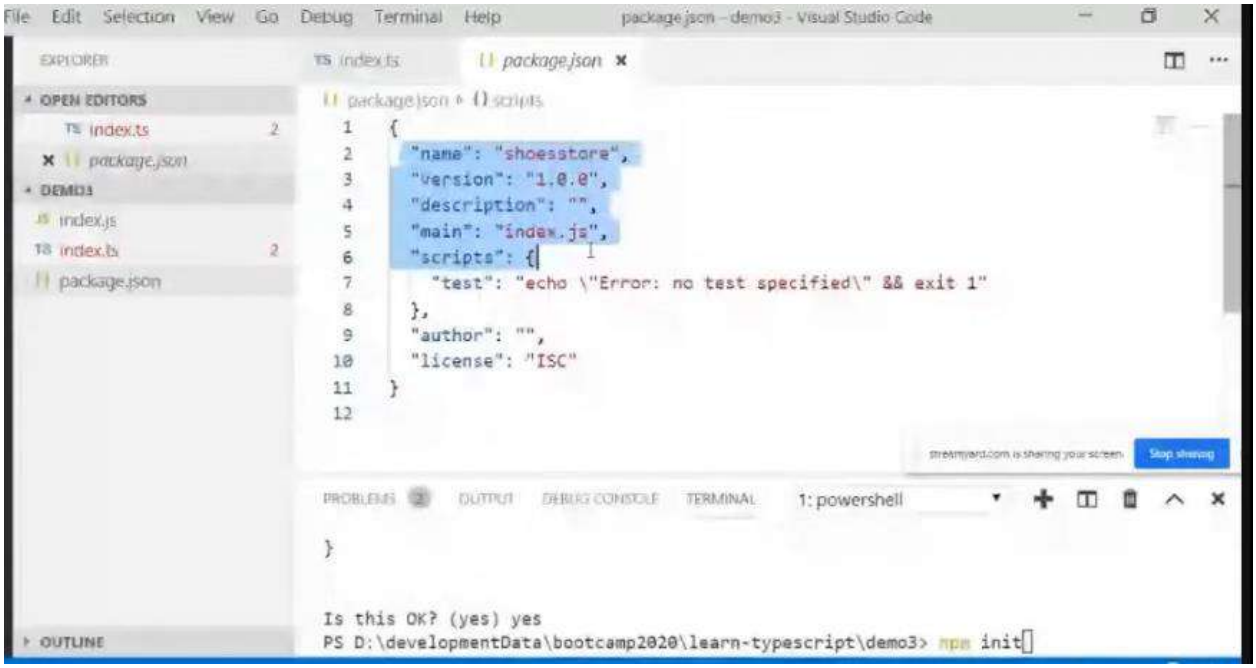
```
"compilerOptions": {
"lib": ["es2015"],
"module": "commonjs",
"outDir": "dist",
"sourceMap": true,
 "strict": true,
"target": "es2015"
},
"include": [
"src"
]
}
```

- Let's Create a project
- type npm that will install the Jason package
- enter package name

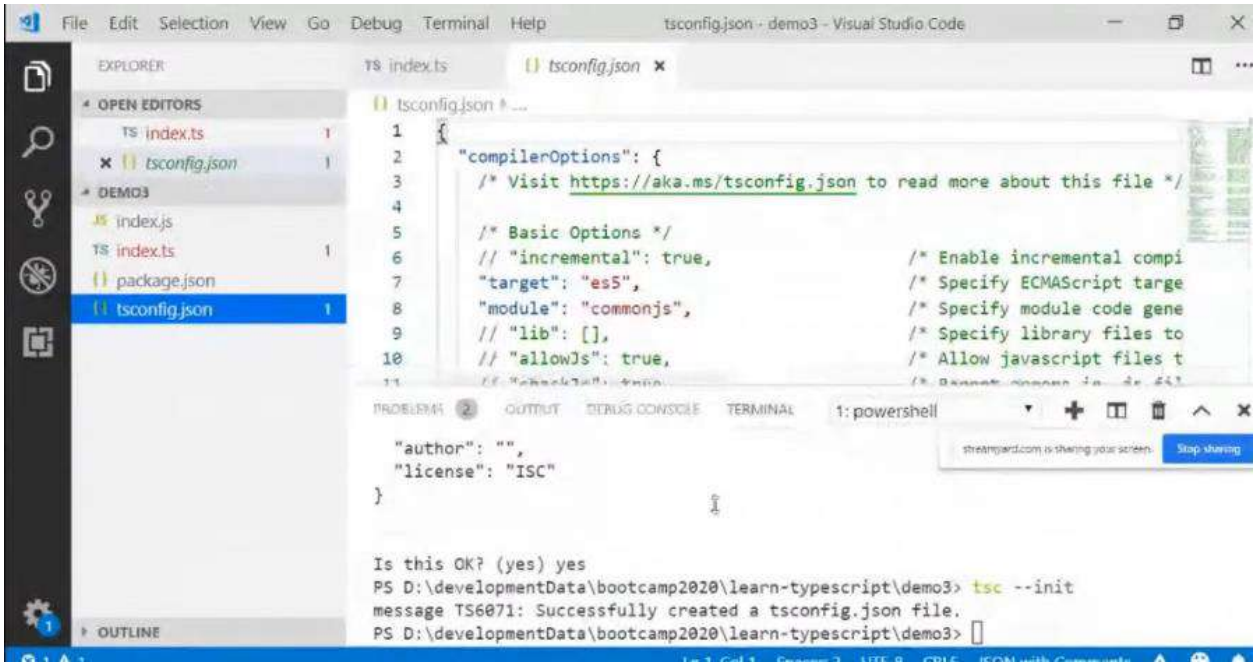


- it has all properties

```
{
  "name" :"shoesstore'"
  "version": "1.0.e",
  "description":"",
  "scripts" :{
   "test":"echo\"Error:no test specified\"&&exit 1"
   },
  "author": ""
   "license":"ISC"
 }
```



- *type txc -- init*
- *this will have number of files but we will use a few of it*



But from these all files we will only install these properties:

```
{

"compilerOptions": {
"lib": ["es2015"],
"module": "commonjs",
"outDir": "dist",
"sourceMap": true,
 "strict": true,
"target": "es2015"
},
"include": [
"src"
]
}
```

- remove all other js files
- make a folder of src ll typescript files will be available in the “src” folder

write this code

```
"use strict"

var arr=[4,5];

arr[1]=78;

console.log(arr[1]);
```

- you no need to enter ts,packjason, and tsconfig to be created it has all in it as shown
- so what you need is just to install a source code in the folder for your typescript and generate

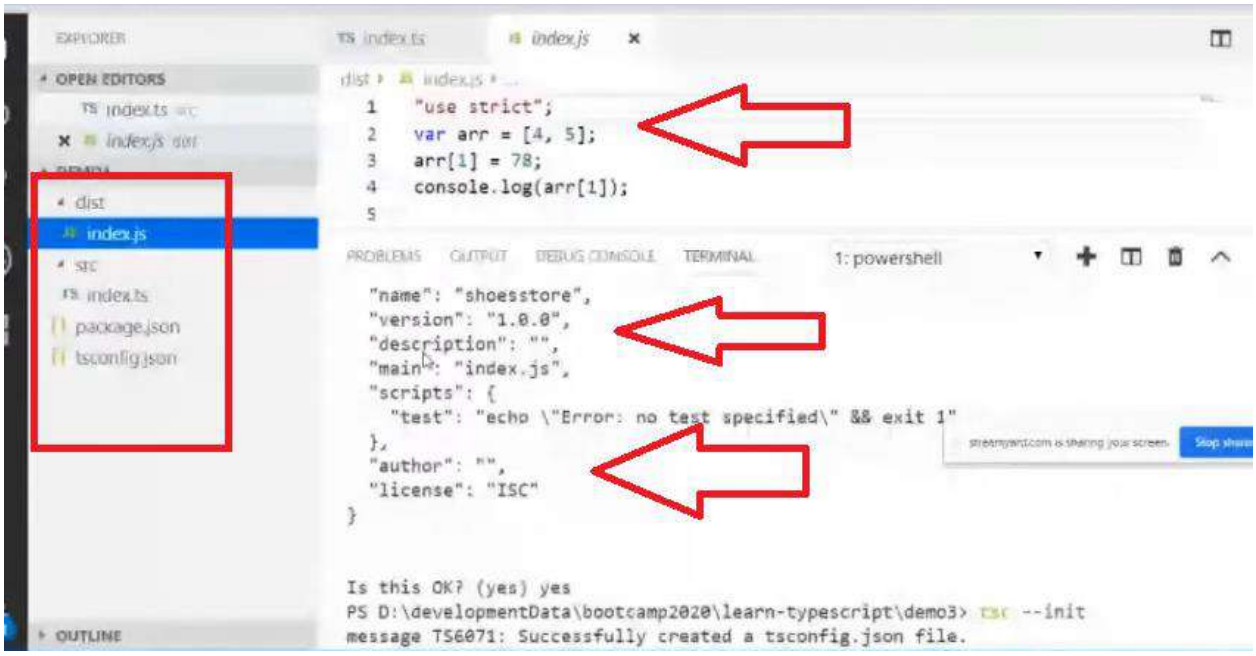
output in the particular folder that it

- *Pop open src/index.ts in your code editor, and enter the following TypeScript code:*

```
console.log('Hello TypeScript!')
```

- *Then, compile and run your TypeScript code:*

```
# Compile your TypeScript with TSC

./node_modules/.bin/tsc


# Run your code with NodeJS

node ./dist/index.js
```



If you've followed all the steps here, your code should run and you should see a single log in your console:

Exercises

Now that your environment is set up, open up src/index.ts in your code editor. Enter the following code:

```
let a = 1 + 2
let b = a + 3
let c = {
    apple: a,
    banana: b
}
let d = c.apple * 4
```

Now hover over a, b, c, and d, and notice how TypeScript infers the types of all your variables for you: ais a number, bis a number, cis an object with a specific shape, and d is also a number

```
6   let a = 1 + 2
7   let b = a + 3
8   let c = {
9     apple: a,
10    banana: b
11  }   let d: number
12  let d = c.apple * 4
```

Play around with your code a bit. See if you can:

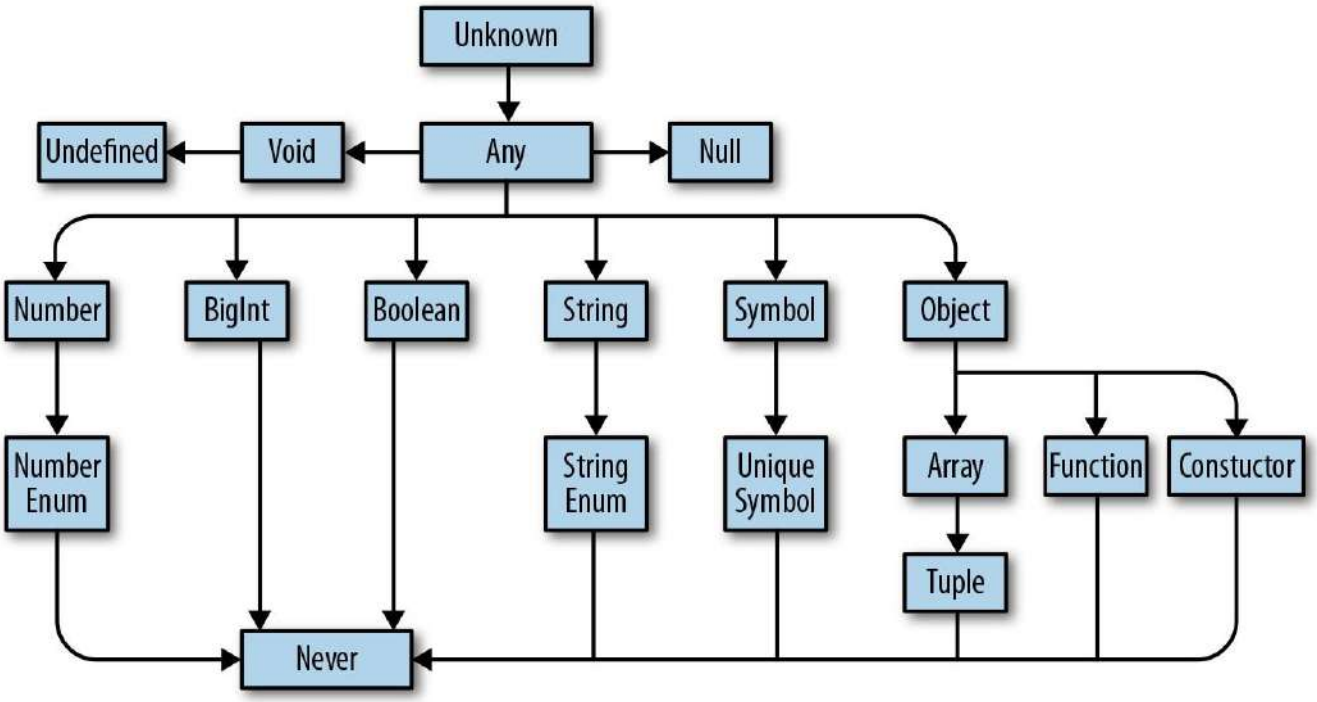
- Get TypeScript to show a red squiggly when you do something invalid (we call this "throwing a TypeError").
- Read the TypeError, and try to understand what it means.
- Fix the TypeErrorand see the red squiggly disappear.

4. TYPE

A set of values and the things you can do with them.

- The booleantype is the set of all booleans (there are just two: trueand false) and the operations you can perform on them (like ||, &&, and !).
- The numbertype is the set of all numbers and the operations you can perform on them (like +, -, *, /, %, ||, &&, and ?), including the methods you can call on them
- like .toFixed, .toPrecision, .toString, and so on.
- The stringtype is the set of all strings and the operations you can perform on them
- (like +, ||, and &&), including the methods, you can call on them like .concat and .toUpperCase.

TypeScript's type of hierarchy

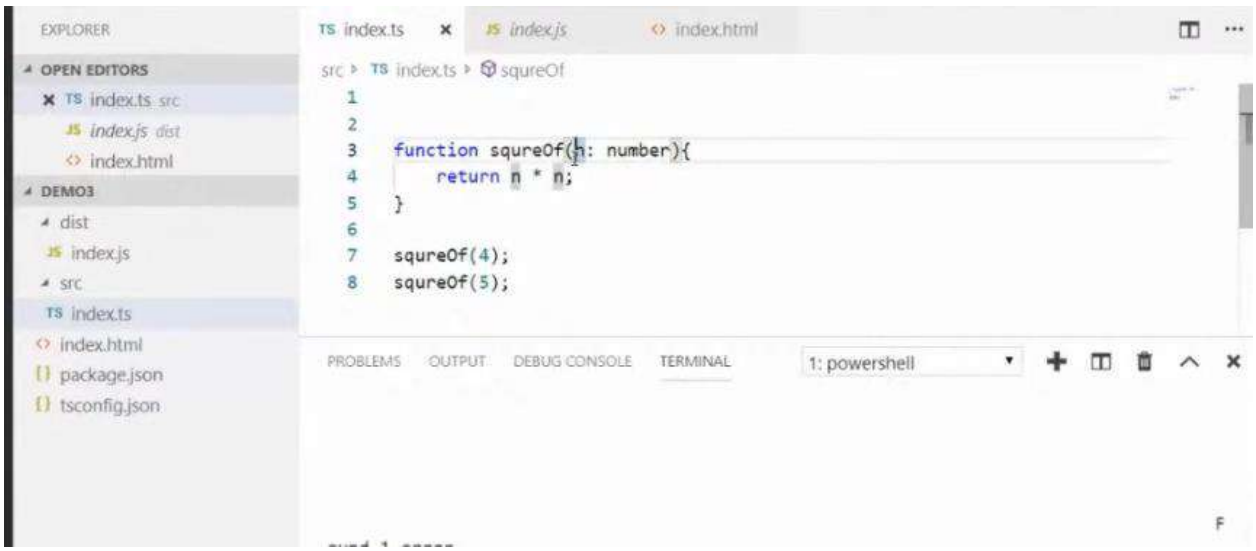


Say you have a function that takes some value and returns that value multiplied by itself:

```
function squareOf(n) {
  return n * n
}
squareOf(2)      // evaluates to 4
squareOf('z')    // evaluates to NaN
```

This function will only work for numbers—if you pass anything besides a number to squareOf, the result will be invalid. So what we do is explicitly annotate the parameter's type:

```
function squareOf(n: number) {
  return n * n
}
squareOf(2)      // evaluates to 4
squareOf('z')    // Error TS2345: Argument of type '"z"' is not assignable to
                 // parameter of type 'number'.
```



5. STRUCTURAL TYPING

A style of programming where you just care that an object has certain properties, and not what its name is (nominal typing). Also called *duck typing* in some languages (or, not judging a book by its cover).

There are a few ways to use types to describe objects in TypeScript. The first is to declare a value as an object:

```
let a: object = {
```

```
  b: 'x'
}
```

What happens when you access b?

```
a.b    // Error TS2339: Property 'b' does not exist on type 'object'.
```

What if we leave off the explicit annotation, and let TypeScript do its thing?

```
let a = {
b: 'x'
}        // {b: string}
a.b      // string

let b = {
c: {
d: 'f'
}
}        // {c: {d: string}}
```

TYPE INFERENCE WHEN DECLARING OBJECTS WITH CONST

What would have happened if we'd used constto declare the object instead?

```
const a: {b: number}= {
  b: 12
```

You might be surprised that TypeScript inferred `b`as a `number`, and not as the literal `12`. After all, we learned that when declaring `number`s or `string`s, our choice of `const` or `let` affects how TypeScript infers our types.

Unlike the primitive types we've looked at so far—`boolean`, `number`, `bigint`, `string`, and `symbol`—declaring an object with `const` won't hint to TypeScript to infer its type more narrowly. That's because JavaScript objects are mutable, and for all TypeScript knows you might update their fields after you create them.

We explore this idea more deeply—including how to opt into narrower inference—in "Type Widening".

Object literal syntax says, "Here is a thing that has this shape." The thing might be an object literal,
or it might be a class:

```
let c: {
  firstName: string
  lastName: string
} = {
```

```
    firstName: 'john',
    lastName:
    'barrowman'

  }


    class
  Person {

constructor
          (

      public firstName: string,        // public is shorthand for

                                       // this.firstName = firstName

      public lastName: string

    ) {}

  }

  c = new Person('matt', 'smith') //OK
```

{firstName: string, lastName: string} describes the *shape* of an object, and both the object literal and the class instance from the last example satisfy that shape, so TypeScript lets us assign a Personto c.

Let's explore what happens when we add extra properties, or leave out required ones:

```
  let a: {b: number}

  a = {}    // Error TS2741: Property 'b' is missing in type '{}'

            // but required in type '{b: number}'.


  a = {

    b: 1,

    c: 2    // Error TS2322: Type '{b: number; c: number}' is not assignable

  }         // to type '{b: number}'. Object literal may only specify known

            // properties, and 'c' does not exist in type '{b: number}'.
```

Optional (?) isn't the only modifier you can use when declaring object types. You can also mark fields as read-only (that is, you can declare that a field can't be modified after it's assigned an initial value—kind of like constfor object properties) with the readonlymodifier:

```
let user: {
  readonly firstName: string
} = {
  firstName: 'abby'
}

user.firstName // string
user.firstName =
  'abbey with an e' // Error TS2540: Cannot assign to 'firstName' because it
                    // is a read-only property.
```

Object literal notation has one special case: empty object types ({}). Every type—except nulland undefined—is assignable to an empty object type, which can make it tricky to use. Try to avoid empty object types when possible:

```
let danger:
{} danger={}
danger={x:
1}danger=[]
danger=2
```

As a final note on objects, it's worth mentioning one last way of typing something as an object: Object. This is pretty much the same as using {}, and is best avoided. 3

To summarize, there are four ways to declare objects in TypeScript:

1. Object literal notation (like {a: string}), also called a *shape*. Use this when you know which *fields your object could have, or when all of your object's values will have the same type.*
2. Empty object literal notation ({}). Try to avoid this.
3. The objecttype. Use this when you just want an object, and don't care about which fields it *has.*
4. The Objecttype. Try to avoid this.

Session 13(A): Programming TypeScript

1. Creating React Quiz App(TypeScript)

- Open up a terminal window
- use npx create-react-app quiz --template types crit
- it will take a few minutes to build a new react app

```
Adil:GitHub adil$ npx create-react-app quiz --template types
cript
```

```
  yarn build
    Bundles the app into static files for production.

  yarn test
    Starts the test runner.

  yarn eject
    Removes this tool and copies build dependencies, configu
ration files
    and scripts into the app directory. If you do this, you
can't go back!

We suggest that you begin by typing:

  cd quiz
  yarn start

Happy hacking!
```

- install style components these are like material UI
- add react-dom

```
> core-js-pure@3.6.5 postinstall /Users/adil/Documents/GitHu
b/quiz/node_modules/core-js-pure
> node -e "try{require('./postinstall')}catch(e){}"

npm notice created a lockfile as package-lock.json. You shou
ld commit this file.
npm WARN notsup Unsupported engine for watchpack-chokidar2@2
.0.0: wanted: {"node":"<8.10.0"} (current: {"node":"12.17.0"
,"npm":"6.14.5"})
npm WARN notsup Not compatible with your version of node/npm
: watchpack-chokidar2@2.0.0

+ @types/styled-components@5.1.1
+ styled-components@5.1.1
added 506 packages from 265 contributors, removed 151 packag
es, updated 1402 packages and audited 1982 packages in 407.5
7s
```

- Now in the index.tsx files create a component folder and create a component of "QuestionCard.tsx

"QuestionCard.tsx

```
import React from 'react'
export const QuestionCard =()=>{
  return(
   <div>
     Question Card
   </div>
 )
}
```

index.tsx

- Create a component index.tsx and in it follow these codings;

  - create a heading tag for "Quiz"
  - declare const startQuiz=async()
  - declare const nextQuestion=async()
  - make an onClick button for start the quiz and inside it write a paragraph of begin quiz
  - add score
  - make a place holder for loading and question card and next Question

index.tsx

```
import React from'react';
function App(){
const startQuiz= async()
const nextQuestion=async()

  return (
   <div className ="App">
    <h1>Quiz</h1>
   <button className='start' onClick={startQuiz}>
        Begin Quiz
   </button>
   <p className='score'>
     Score
    </p>
  <p>
  Loading
  </p>
   <QuestionCard
<button ClassName='next' onClick ={nextQuestion}>
  Next question
</button>
   </div>
  );
}
export default App;
```

- go to the browser to look final output as

  we have

  - quiz heading
  - Begin Quiz ClickOn Button
  - Loading
  - QuestionCard

- In "QuestionCard.tsx to get properties of question
- Add type Props:for properties and inside them add another properties as :

```
type Props={
    question:string;
    answer:string[];
    callback: any;
    userName:String;
    questionNum: number;
    totalQuestions:number;
```

-
- After that in the export const QuestionCard function Add React.FC<Props>=({}) and add props init as :

```
export const QuestionCard: React.FC<Props>=(question, Answer,
callback,userAnswer,questionNum,totalQuestions)=>{
```

- in div section dispaly question over total question with the paragraph details of the current question as

```
<div>
  <p>
    Question :{questionNum}/{totalQuestion}
  </p>
 <p dangerous lysetInnerHTML={_html:question}}/>
```

- define properties of all question by using useState to describe question state and then add these properties into the QuestioCard
- declare the function of total questions as

```
const TOTAL_QUESTIONS =10;
```

- use the value of these question in the questioncardas as well as fetch answer in the question and also let know if the user has selected or not slected any question
- callback function is called if the called value is correct

```
<QuestionCard
questionNum={number+1}//question .1
totalQuestions={TOTAL-QUESTIONS}
question={question[number].question}
answer={question[number].answer}
userAnswer={userAnswers? userAnswer[number] ;undefined
callback={checkAnswer}
```

- for that create another component for checking answer that is onthe mouseEvent that will be get from a click on html button

```
const checkAnswer=(e:React.MouseEvent<HTMLButtonElement>)=>();
```

now to this API create anothr component API.tsx

APP.tsx

```
import Reac,{useState}from'react';
const TOTAL_QUESTIONS =10;
function App(){
  const[loadind,setLoading]=useStte(false);
  const[question,setQuestions]=useState([]);
  const[number,setNumbers]=useState(0);
  const[userAnswers,setUserAnswer]=useState([]);
  const[score,setScore]=useState(0);
  const[gameover,setGameOver]=useState(true);


const startQuiz=async()=>{};
const nextQuestion=async()=>{};
const checkAnswer=(e:React.MouseEvent<HTMLButtonElement>)=>();

  return (
   <div className ="App">
    <h1>Quiz</h1>
   <button className='start' onClick={startQuiz}>
        Begin Quiz
   </button>
   <p className='score'>
     Score
    </p>
  <p>
  Loading
  </p>
   <QuestionCard
    questionNum={number+1}//question .1
    totalQuestions={TOTAL-QUESTIONS}
    question={question[number].question}
    answer={question[number].answer}
    userAnswer={userAnswers? userAnswer[number] ;undefined
    callback={checkAnswer}
 />
<button ClassName='next' onClick ={nextQuestion}>
  Next question
</button>
   </div>
  );
}
export default App;
```

- create a component folder and create a component of "QuestionCard.tsx

to get properties of question

- Add type Props:for properties and inside them add another properties as :

```
type Props={
    question:string;
    answer:string[];
    callback: any;
    userName:String;
    questionNum: number;
    totalQuestions:nnumber;
```

- After that in the export const QuestionCard function Add React.FC<Props>=({}) and add props init as :

```
export const QuestionCard: React.FC<Props>=(question, Answer,
callback,userAnswer,questionNum,totalQuestions)=>{
```

- in div section dispaly question over total question with the paragraph details of the current question as

```
<div>
  <p>
    Question :{questionNum}/{totalQuestion}
  </p>
 <p dangerous lysetInnerHTML={_html:question}}/>
```

- for answer map throght the array of questiom

for that in div section map the answer as

```
<div>
  {answer.map(answer =>(
  </div>
```

- to make button for this map we use span as

```
<button disabled={userAnswer} onClick={callback}>
    <span dangerouslySetInnerHTML={{___html:answer}}/>
```

QuestionCard.tsx

```
import React from 'react'
type Props={
    question:string;
    answer:string[];
    callback: any;
    userName:any;
    questionNum: number;
    totalQuestions:nnumber;

 }

export const QuestionCard React.FC<Props>=(question, Answer,
callback,userAnswer,questionNum,totalQuestions)=>{
  return(
   <div>
   <p>
     Question :{questionNum}/{totalQuestion}
   </p>
  <p dangerous lysetInnerHTML={_html:question}}/>
  <div>

   {answer.map(answer =>(

  <div>

   <button disabled={userAnswer} onClick={callback}>

    <span dangerouslySetInnerHTML={{___html:answer}}/>

   </div>

 )
```

API.tsx

- call API to call the data
- create API.tsx file

  export "export const fetchQuestions=async() and enter the parameter in it as

  - amount: number;
  - difficulty

- for difficulty export "export enum Difficulty" this will restrict us to a certain situation that is only available in API

  - EASY
  - MEDIUM
  - HARD

- for putting these enum built endpoint and provide the URL of API as

  const endpoint=`https://opentdb.com/api.php?`

- now to call this we use jason data as

  const data=await fetch(endpoint)).json();

- to store data in it we will call data in console log

  API in APP.tsx

- call fetch function into the App.tsx as

  import {fetchQuestions, difficulty} from './API';

- and log this data as well as fetch it

  console.log(fetchQuestion(TOTAL QUESTION,Difficulty,Easy));

- in Api export question type as

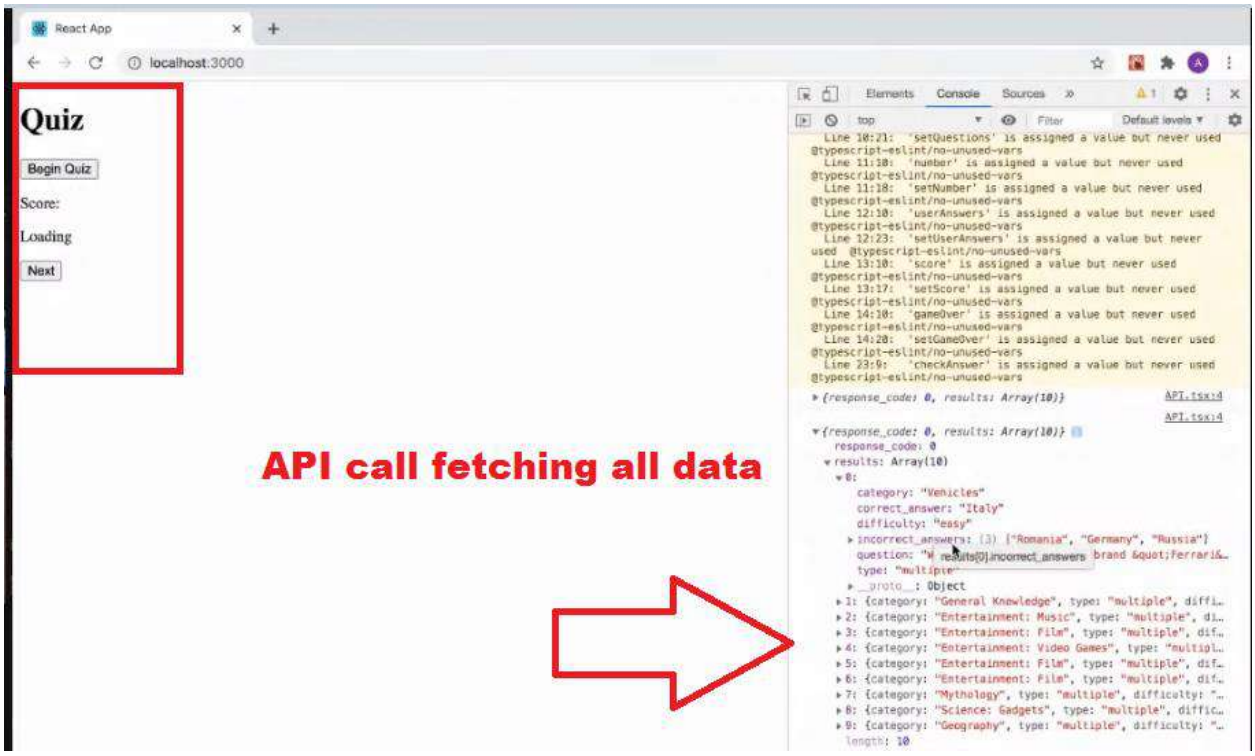
```
export type Question{
category:string;
correct-Answer:string;
incorrect-answer:string[];
difficulty:string;
question:string;
type:string;
```

API.tsx

```
export const fetchQuestions=async(amount:number,difficulty:Difficulty)=>
const
endpoint=`https://opentdb.com/api.php?amount=${amount}&difficulty=${d
ifficulty}&type=multiple;`
const data=await fetch(endpoint)).json();
console.log(data);

}

export enum Difficulty{
EASY ="easy"
MEDIUM="medium"
HARD ="hard"
}
        export type Question{
        category:string;
        correct_answer:string;
        incorrect_answer:string[];
        difficulty:string;
        question:string;
        type:string;
}
Export type QuestionState = Question & { answers: string[ ] };
```

APP .tsx

```
import Reac,{useState}from'react';
const TOTAL_QUESTIONS =10;
function App(){
  const[loadind,setLoading]=useStte(false);
  const[question,setQuestions]=useState([]);
  const[number,setNumbers]=useState(0);
  const[userAnswers,setUserAnswer]=useState([]);
  const[score,setScore]=useState(0);
  const[gameOver,setGameOver]=useState(true);

  console.log(fetchQuestion(TOTAL_QUESTION, Difficulty.Easy));

const startQuiz=async()=>{};
const nextQuestion=async()=>{};
const checkAnswer=(e:React.MouseEvent<HTMLButtonElement>)=>();

  return (
   <div className ="App">
    <h1>Quiz</h1>
   <button className='start' onClick={startQuiz}>
        Begin Quiz
   </button>
   <p className='score'>
     Score
    </p>
  <p>
  Loading
  </p>
   <QuestionCard
    questionNum={number+1}//question .1
    totalQuestions={TOTAL-QUESTIONS}
    question={question[number].question}
    answer={question[number].answer}
    userAnswer={userAnswers? userAnswer[number] ;undefined
    callback={checkAnswer}
 />
<button ClassName='next' onClick ={nextQuestion}>
  Next question
</button>
   </div>
  );
}
export default App;
```


React App
localhost:3000
Quiz
Begin Quiz
Score:
Loading
Next
API call fetching all data
Elements
Console
Sources
API.tsx:4
API.tsx:4

API.tsx:4
API.tsx:4
response_code: 0
results: Array(10)
category: "Vehicles"
correct_answer: "Italy"
difficulty: "easy"
incorrect_answers: (3) ["Romania", "Germany", "Russia"]
results[0].incorrect_answers
length: 10

- In API.tsx create a new object that will comine two pf question answer as in one

  export type QuestionState =Question & {answer:string[]}

- create a retundata map that will display the question answer both as

```
return data.results.map((question:Question))=.(
 {
   .....question,
answers :[…question.Incorrect -Answer, question.correct -Answers]
```

}

API.tsx

```
export const fetchQuestions=async(amount:number,difficulty:Difficulty)=>
const
endpoint=`https://opentdb.com/api.php?amount=${amount}&difficulty=${difficulty}&type=multiple;`
const data=await fetch(endpoint)).json();
        return data.results.map((question:Question))=.(
          {
            ...question,
                answers shuffleArray:([…question.Incorrect -Answer, question.correct -
        Answers]

          }
console.log(data);

export enum Difficulty{
EASY ="easy"
MEDIUM="medium"
HARD ="hard"
}
        export type Question{
        category:string;
        correct-Answer:string;
        incorrect-answer:string[];
        difficulty:string;
        question:string;
        type:string;
}
export type QuestionState =Question & {answer:string[]};
```

- add answer ,shuffel the answer for that create another function
- create a file for utilities
- inside this file create function to shuffle the conetnt of the array
- get existing array and sort it in random way

```
export const shuffleArray =array:ant[]=.

[...array].sort(()=.Math.random()-0.5)
```

- import shuffle array from utilities in API to shuffle the answers

```
import { shuffleArray } from './utilities'
```

utilities .txs

```
export const shuffleArray =array:ant[]=.

[...array].sort(()=.Math.random()-0.5)
```

- import {QuestionCard} from './components/QuestionCard';
  import {fetchQuestions,Difficulty,QustionState} from './API;

  - provide type of anwer object as

```
type AnswerObject={

question:string;

answer:string;

correct:boolean;

correct Answer:string;

}
```

APP .tsx

```
import React,{useState}from'react';
import {QuestionCard} from './components/QuestionCard';
import {fetchQuestions,Difficulty,QustiOnState} from './API;
      const TOTAL_QUESTIONS =10;
  type AnswerObject={
  question:string;
   answer:string;
   correct:boolean;
   correct Answer:string;
}
```

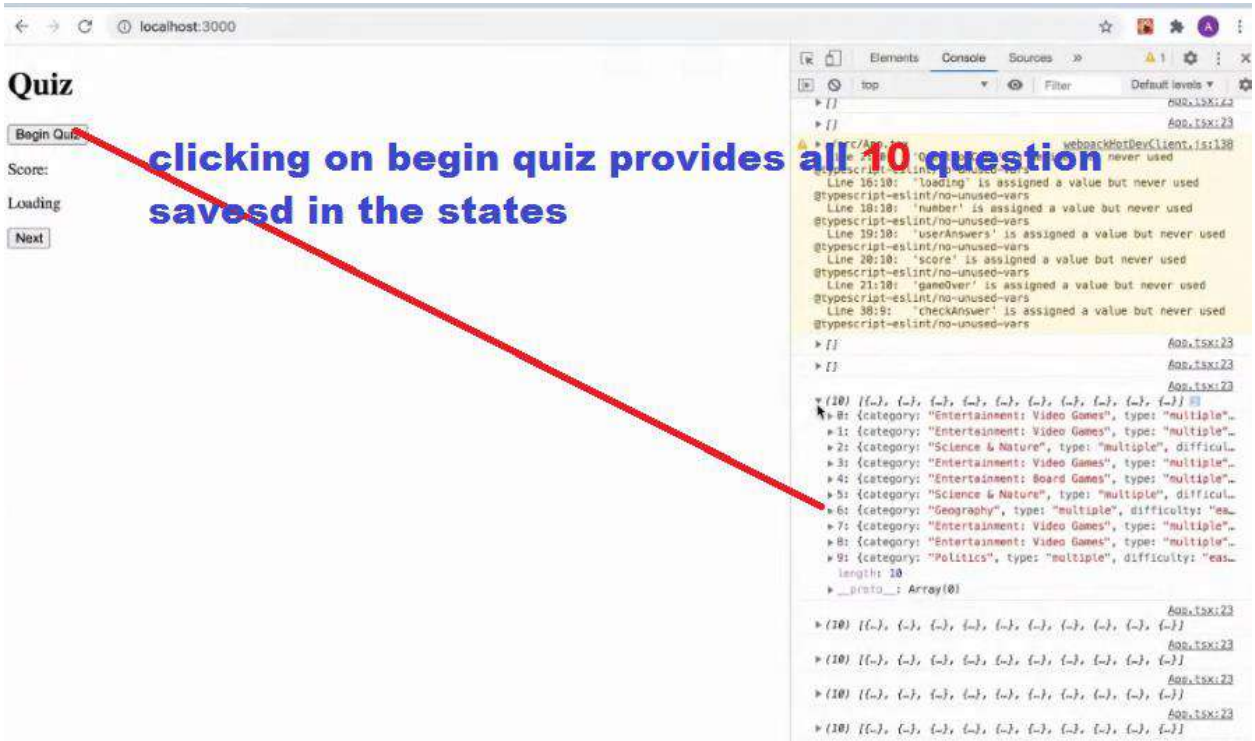
- Set begin quiz with this coding inside App.tsx
- Console.log to fetch data as well
- By clicking on begin quiz will provide all data of 10 question that are being saved in the states

```
13
14  function App() {
15
16    const [loading, setLoading] = useState(false);
17    const [questions, setQuestions] = useState<QuestionState[]>([]);
18    const [number, setNumber] = useState(0);
19    const [userAnswers, setUserAnswers] = useState<AnswerObject[]>([]);
20    const [score, setScore] = useState(0);
21    const [gameOver, setGameOver] = useState(true);
22
23    console.log(questions);
24
25
26    const startQuiz = async() => {
27      setLoading(true);
28      setGameOver(false);
29      const newQuestions = await fetchQuestions(TOTAL_QUESTIONS, Difficulty.EAS\
30      setQuestions(newQuestions);
31      setScore(0);
32      setUserAnswers([]);
33      setNumber(0);
```

console log to fetch data

start quiz begin activity

set all terms



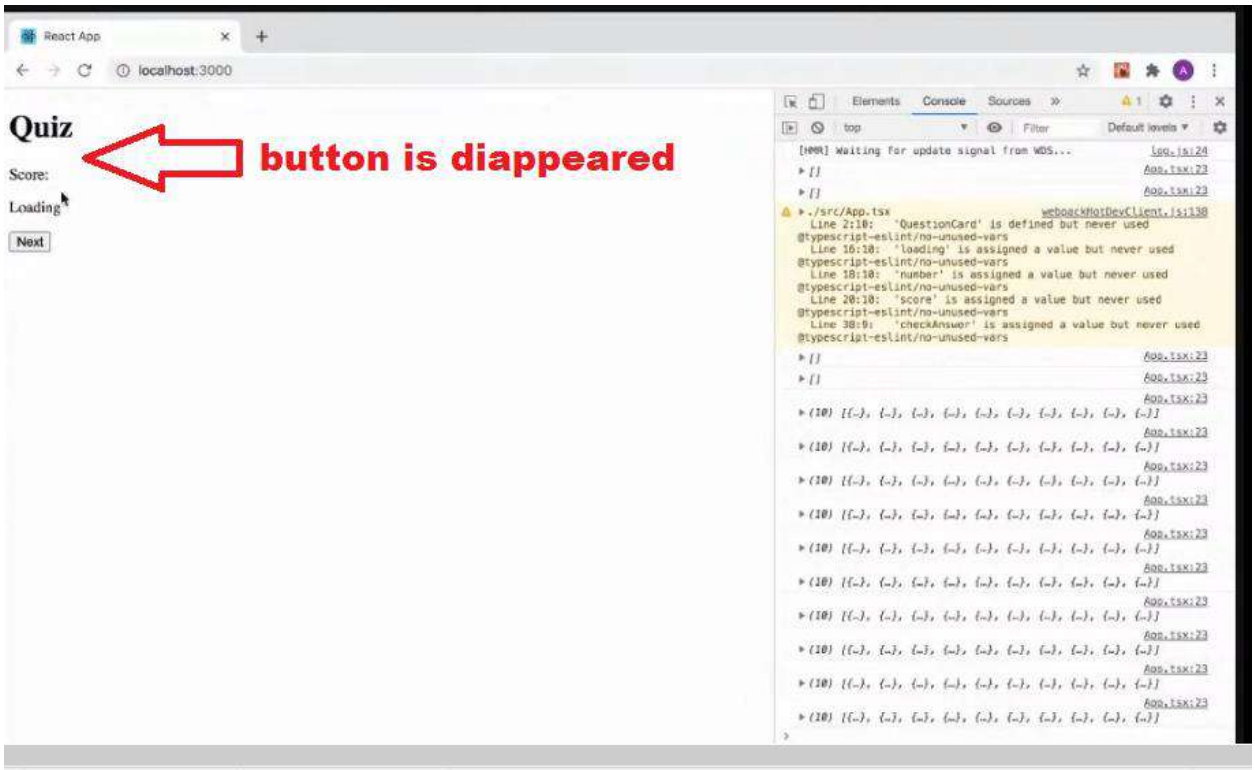
set conditional displaying

- button to show the quiz is completed will show the game over

- so check if the game is over or the user answer is equal to 10 questions then the button will be shown otherwise nothing

```
<div className="App">
  <h1>Quiz</h1>
  {gameOver || userAnswers.length === TOTAL_QUESTIONS ? (
  <button className='start' onClick={startQuiz}>
    Begin Quiz
  </button> ) : null
```

```
return (
  <div className="App">
    <h1>Quiz</h1>
    {gameOver || userAnswers.length === TOTAL_QUESTIONS ? (
    <button className='start' onClick={startQuiz}>
      Begin Quiz
    </button> ) : null
    <p className='score'>
      Score:
    </p>
    <p>
      Loading
    </p>
    {/*
    <QuestionCard
```



score

- if the quiz is done the paragraph will not be shown and vice versa

```
{!gameover?()
<p className-'score'>
  Score:
  </p>)):null}
```

Loading

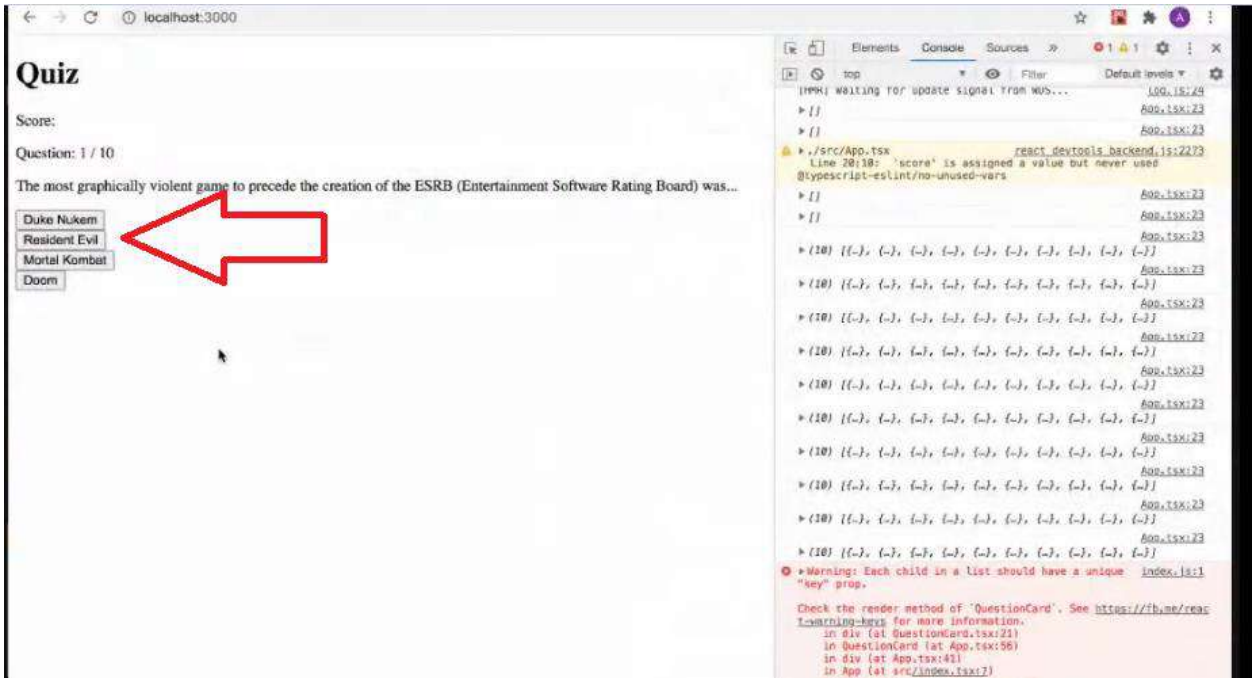
if the data is loadind then load ing print will be shown otherwise not

```
{loading?(
<p>
loading
</p>):null;
```

QuestionCard

if the not loading if the game is not over than show this questioncard otherwise not

```
{!loading && !gameOver ? (
<QuestionCard
  questionNum={number + 1}
  totalQuestions={TOTAL_QUESTIONS}
  question={questions[number].question}
  answers={questions[number].answers}
  userAnswer={userAnswers ? userAnswers[number] : undefined }
  callback={checkAnswer}
/> ) : null }
```



This is how you can check out the **set conditional displaying of your data** in the quiz

Session 13(B): Programming TypeScript

Quiz React App

- add functonality to check the answer
- create a coditional statement wether the quiz is over or not
- create a const to store the answer we have selected from an event from the current value

```
        if(!gameOver){

    const answer= e.currentTarget.Value:

    }

  };
```

- **now update questionCard.tsx and add value is equal to the answer in the button section**

```
{answers.map(answer => (
    <div>
        <button disabled={userAnswer} value={answer} onClick={callback}>
```

- create another const of correct answer to check as to wether the question answerr by the user is true or false if it is true then setScore by prev+1

```
const correct = question[number].correct_answer ===answer;

if(correct) setScore(prev=> prev+1)'
```

- create another object and in it state the user answer in it

```
const answerObject ={
  question:question [number].qusetion
  answer,
  correct,
  correctAnswer:qusetion[number].correct_answer
```

- call setuserAnswer function get previous data and get new data
- this will update the state and keep the history about the user answers

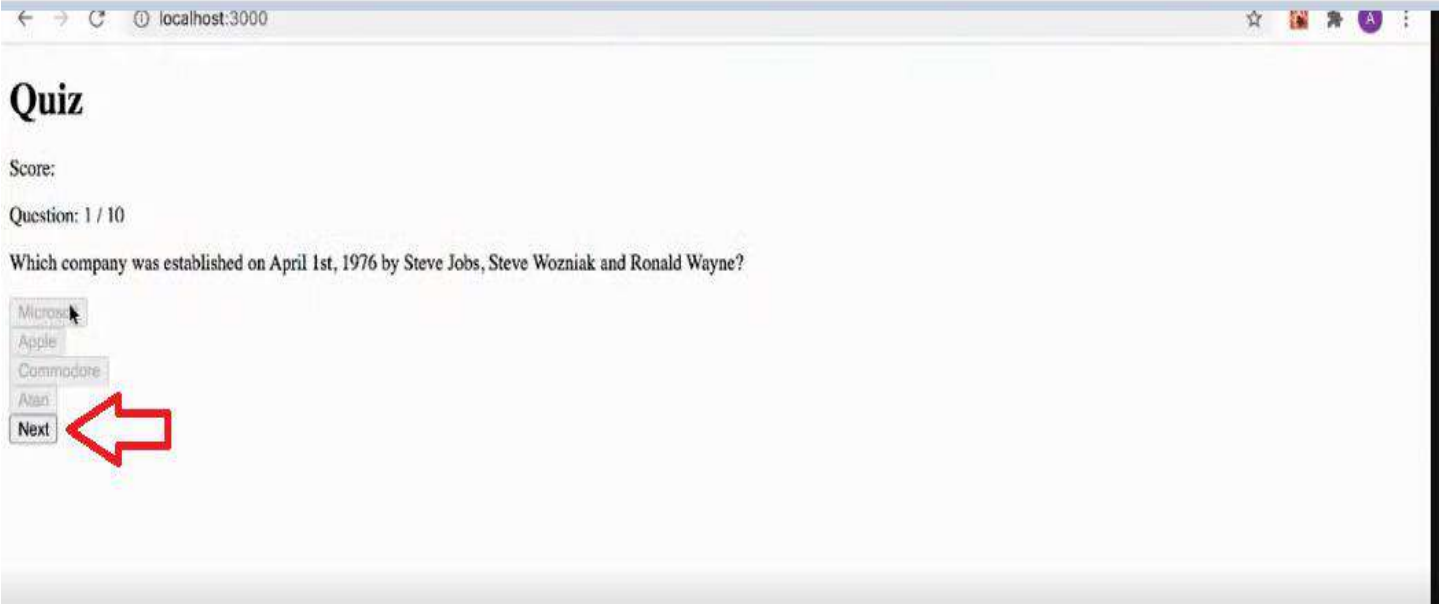
```
setUserAnswer(prev=>[...prev, anserObject])
```

- look at the app and click any option; it would show us the button of next as shown

APP .tsx

```
import React, {useState} from'react' ;
import { QuestionCard } from './components/QuestionCard' ;
import {fetchQuestions, Difficulty, QuestiOnState} from './API ;
          const TOTAL_QUESTIONS =10;
   type AnswerObject={
   question: string;
    answer: string[ ];
    correct: boolean;
    correct Answer: string;
}

function App() {
  const [loading, setLoading] = useState(false) ;
  const [questions, setQuestions] = useState<QuestiOnState[ ]>([ ]);
  const [number, setNumbers] = useState(0);
  const [userAnswers, setUserAnswer] = useState<AnswerObject[ ]>([ ]);
  const [score, setScore] = useState(0);
  const [gameover, setGameOver ]= useState(true);

   console.log(fetchQuestion(TOTAL QUESTION,Difficulty,Easy));

const startQuiz = async( ) => {};

const nextQuestion = async ( ) => {};
const checkAnswer = (e:React.MouseEvent<HTMLButtonElement>) => {} ;
          if(!gameOver){
            const answer= e.currentTarget.Value:
          const correct = question[number].correct_answer ===answer;
          if(correct) setScore(prev=> prev+1)
          const answerObject = {
            question:questions[number].qusetion,
            answer,
             correct,
            correctAnswer: questions[number].correct_answer
            }
                setUserAnswer(prev => [...prev, anwserObject])
          }
          };

  return (
   <div className ="App">
    <h1>Quiz</h1>
   <button className='start' onClick={startQuiz}>
        Begin Quiz
   </button>
   <p className='score'>
     Score
    </p>
  <p>
```

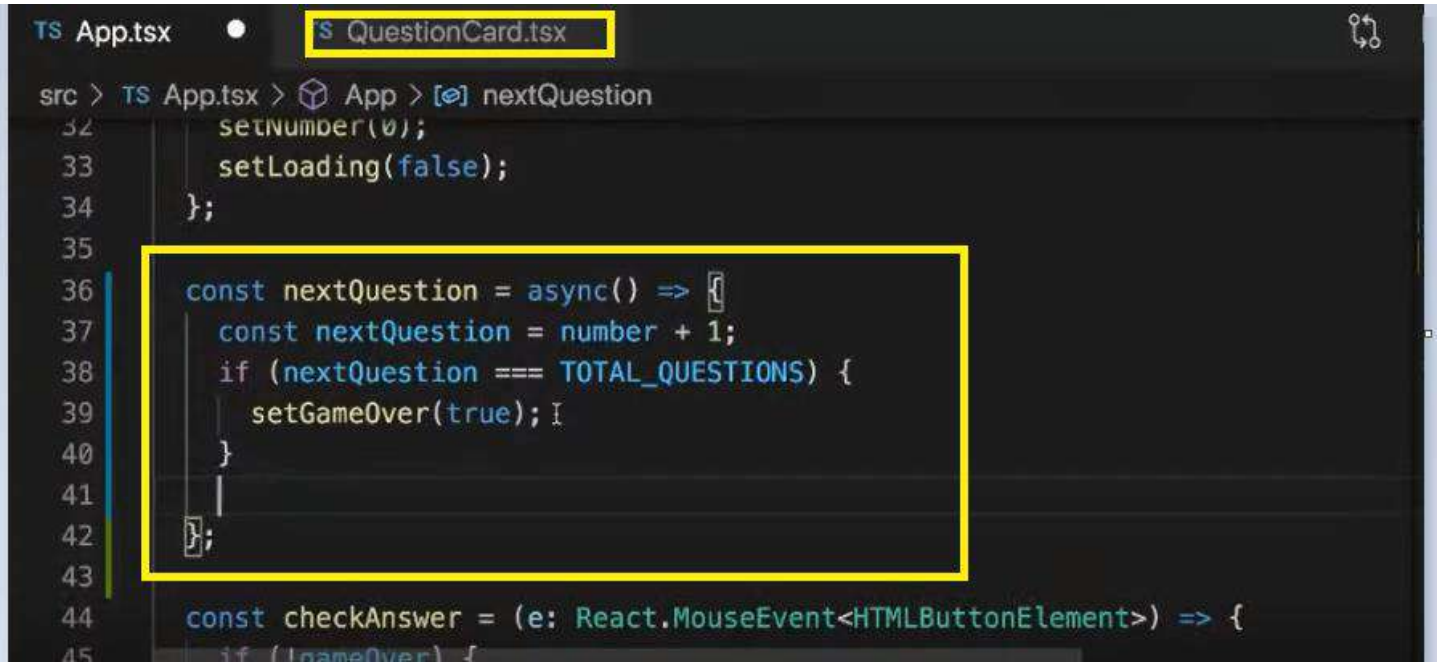
```
  Loading
  </p>
   <QuestionCard


  questionNum={number+1}//question .1
    totalQuestions={TOTAL-QUESTIONS}
    question={question[number].question}
    answer={questions[number].answer}
    userAnswer={userAnswers ? userAnswers[number] : undefined}
    callback={checkAnswer}
 />
<button ClassName='next' onClick ={nextQuestion}>
  Next question
</button>
   </div>
  );
}
export default App;
```



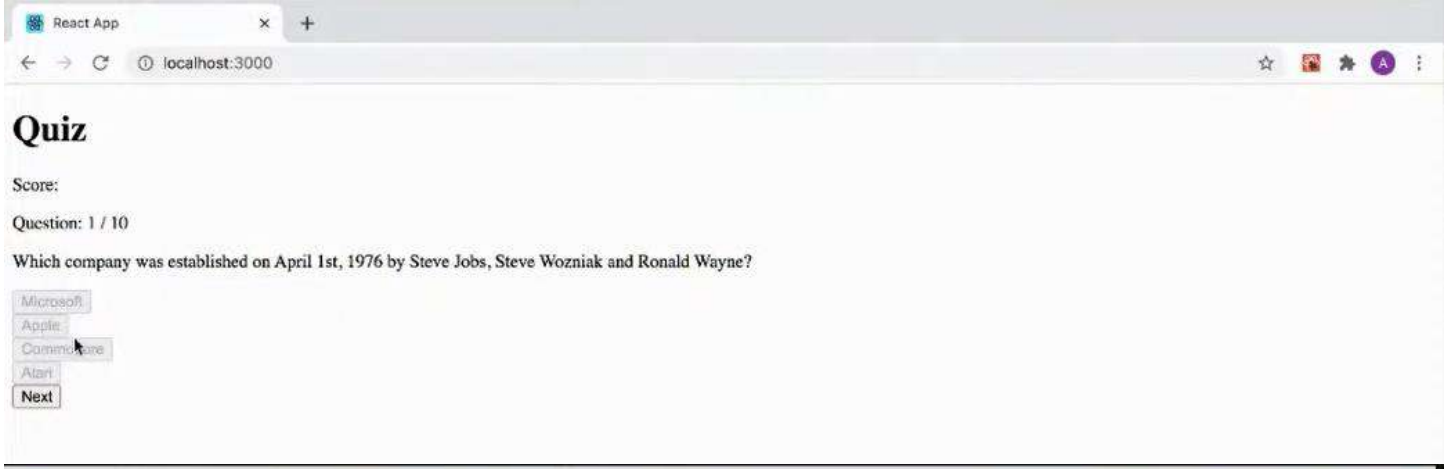
- update the number as the value of the next question in QuestionCard.tsx
- apply logic to get know that the user has not been ended and continue to next and if it has completed the question than it should display the game over

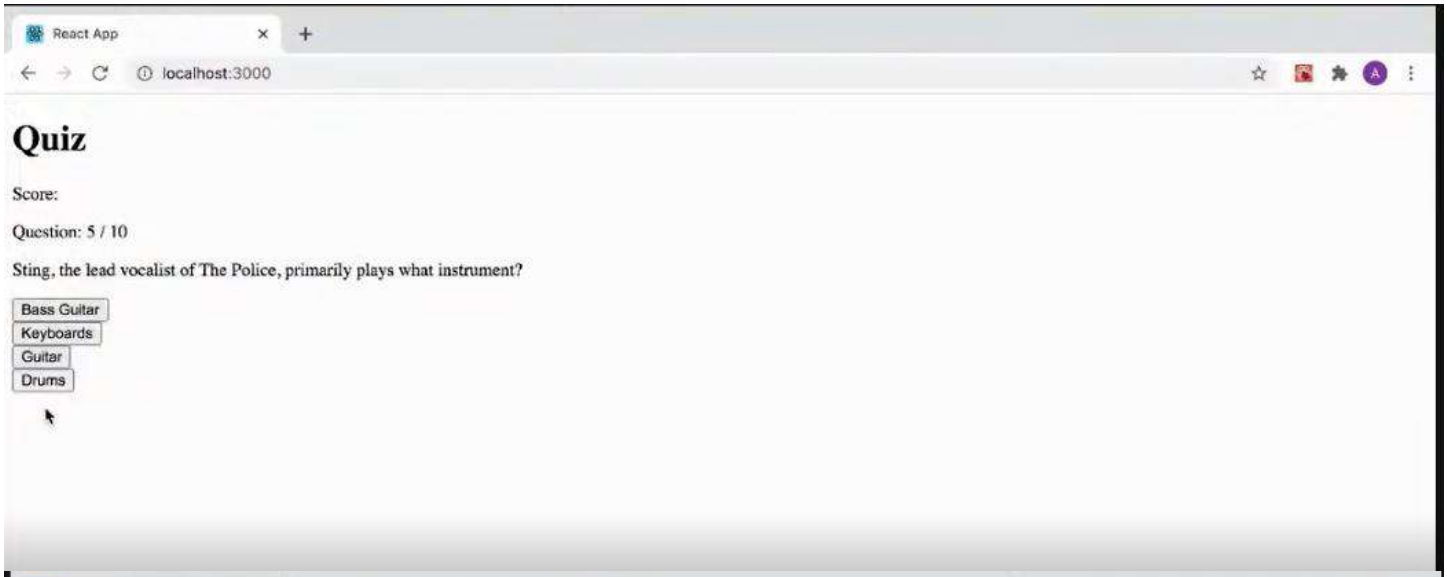
as

```
const nextQusetion= number + 1;
if (nextQuestion === TOTAL_QUESTIONS) {
  setGameOver(true);
}
else {
   setNumber(nextQuestion);
}
```



Look at the display it will keep going to the next question as we proceed by selecting a choice question until it reaches to total question and it shows begin quiz again as we approach total questions:

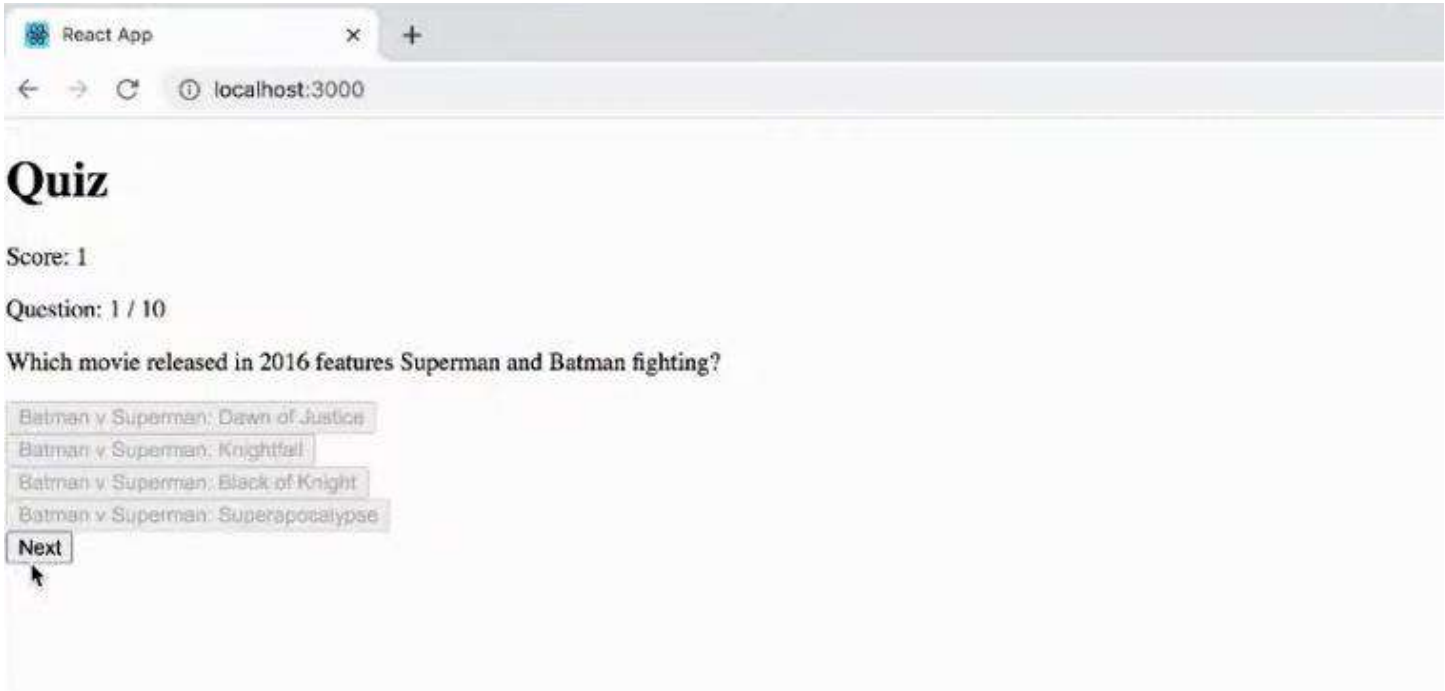

React App
localhost:3000
Quiz
Score:
Question: 1 / 10
Which company was established on April 1st, 1976 by Steve Jobs, Steve Wozniak and Ronald Wayne?
Microsoft
Apple
Commodore
Atari
Next


React App
localhost:3000
Quiz
Score:
Question: 5 / 10
Sting, the lead vocalist of The Police, primarily plays what instrument?
Bass Guitar
Keyboards
Guitar
Drums

React App
localhost:3000
Quiz
Score:
Question: 7 / 10
Which show is known for the songs "You are a Pirate", "Cooking by the Book" and "We Are Number One"?
Sofia the First
LazyTown
Tom and Jerry
DuckTales
Next

Update score section as well as:

```
<p className='score'>
  Score: {score}
</p> ) : null }
```



- now apply a CSS style to App.tsx for that purpose create a new file src as App.styles.ts
- create a global style from style component

App.styles.ts

```
import Styled, {createGlobalStyle} frrom 'styled-components';

export consr GlobalStyle= createGlobalStyle'
html {
    height:100%;
   }

  body {
          background-color:navy;
          margin:0;
          padding;0 20px;
          display flex;
          justify-content:center
     }
    *{
       box-sizing:border-box;
       font-family: 'catamaran', sans-serif;
    }


 export const Wrapper=styled.div`
  display flex;
  flex-direction: column;
  align-items: center;
>p {
    color:#fff;
  }
 .score {
  color:#fff:
  font-size:2rem;
  margin:0;
}
 h1 {
   font-family: Fascinate Inline;
   background-image: linear-gradient(180deg,#fff, #87fiff);
   font-weight: 400;
   background size: 100%;
   background-clip: text;
  -webkit-background-clip:  text;
  -webkit-text-fill-color: transparent;
  -moz-background-clip: text;
  -moz-text-fill-color: transparent;
   filter: drop-shadow(2px 2px 300085a3);
   font-size: 70px;
   text-align: center;
```

```
margin:20px;
 }
 .start, .next {
   cursor: pointer;
   background: linear-gradient(180deg,#ffffff, 3ffcc91);
   border: 2px solid 3d38558;
   box-shadow:0px 5px 10px rgba(0,0, 0, 0.25);
   border-radius: 10px;
   height: 40px;
   margin: 20px 0;
   padding:0 40px:
 }

.start{
   max-width:200px;
};
```

- now import this css style to our App.ts as

  import{GloblStyle} from '/App.style.ts';

- now in App.tsx apply this globalStyle as a whole as

  <>

  <GlobalStyle/>

- Add wrapper as well and close the tag

```
};

return (
  <>
    <GlobalStyle />
    <Wrapper>
      <h1>Quiz</h1>
      {gameOver || userAnswers.length === TOTAL_QUESTIONS ? (
      <button className='start' onClick={startQuiz}>
        Begin Quiz
      </button> ) : null }
      {!gameOver ? (
      <p class=Name='score'>
```

APP .tsx

```
import {fetchQuestions, Difficulty, QuestiOnState} from './API ;
const TOTAL_QUESTIONS =10;
   type AnswerObject={
   question: string;
    answer: string[ ];
    correct: boolean;
    correct Answer: string;
}

function App() {
  const [loading, setLoading] = useState(false) ;
  const [questions, setQuestions] = useState<QuestiOnState[ ]>([ ]);
  const [number, setNumbers] = useState(0);
  const [userAnswers, setUserAnswer] = useState<AnswerObject[ ]>([ ]);
  const [score, setScore] = useState(0);
  const [gameover, setGameOver ]= useState(true);

   console.log(fetchQuestion(TOTAL QUESTION,Difficulty,Easy));

const startQuiz = async( ) => {};

const nextQuestion = async ( ) => {};
const checkAnswer = (e:React.MouseEvent<HTMLButtonElement>) => {} ;
if(!gameOver){
  const answer= e.currentTarget.Value:
const correct = question[number].correct_answer ===answer;
if(correct) setScore(prev=> prev+1)
const answerObject = {
  question:questions[number].qusetion,
  answer,
   correct,
  correctAnswer: questions[number].correct_answer
  }
                setUserAnswer(prev => [...prev, anwserObject])
}
};

  Return(

   <>
            <GlobalStyle/>
          <wrapper>

<div className ="App">
 <h1>Quiz</h1>
<button className='start' onClick={startQuiz}>
    Begin Quiz
</button>
<p className='score'>
```

```
    </p>
  <p>
  Loading



  </p>

   <QuestionCard


  questionNum={number+1}//question .1
    totalQuestions={TOTAL-QUESTIONS}
    question={question[number].question}
    answer={questions[number].answer}
    userAnswer={userAnswers ? userAnswers[number] : undefined}
    callback={checkAnswer}
 />
<button ClassName='next' onClick ={nextQuestion}>
  Next question
</button>
   </div>

  );
}
export default App;
```

- apply a CSS style to our questionCard component; create another src fie of QusetioonCard.style.ts it contains

CSS style for

- Wrapper

- W rapper button

QusetioonCard.style.ts

```
import styled from 'styled-components';

export const Wrapper=styled.div`
  max-width: 1100px;
 background: 3ebfeff;
  border-radius: 10px;
  border: 2px solid #0085a3;
  padding: 20px;
  box-shadow: 0px 5px 10px rgba(0, 0, 0, 0.25);
  text-align:center;
p{
 font-size:1rem;
}
;
;
   type ButtomWrapperProps={
     correct:boolean;
     userClicked:boolean;
   };

export const ButtonWrapper= styled.div<ButtonWrapperProps>`
  transition: all 0.3s ease;
  :hover{
    opacity: 0.8;
  }
button {
    cursor:pointer;
    user-select:none;
   font-size:0.8rem;
   width: 100%;
   height: 40px;
   margin: 5px 0;
   background:${({correct,userClicked}) =>
       correct
           ? 'linear-gradient(90deg, 356FFA4, 359BC86)'
           : !correct && userClicked
           ?' linear-gradient(90deg, #56cff, #beafb4)'};
           : 'linear-gradient(90deg,#56ccff, #6eafb4)'};

  border: 3px solid 3ffffff;
  box-shadow:1px 2px 0px rgba(0,0,0.1);
  border-radius:10px;
  color :#fff;
  text-shadow:0px 1px rgba(0, 0, 0.25);
  }

;
```

- Now import QusetionCard.style.ts to questionCard.ts component as

  Import {Wrapper, ButtonWrapper} from ‘./QuestionCard.styles’

```
TS QuestionCard.tsx ●   TS QuestionCard.styles.ts
src > components > TS QuestionCard.tsx > ...
1  import React from 'react';
2  import { Wrapper, ButtonWrapper } from ./QuestionCard.styles'
```

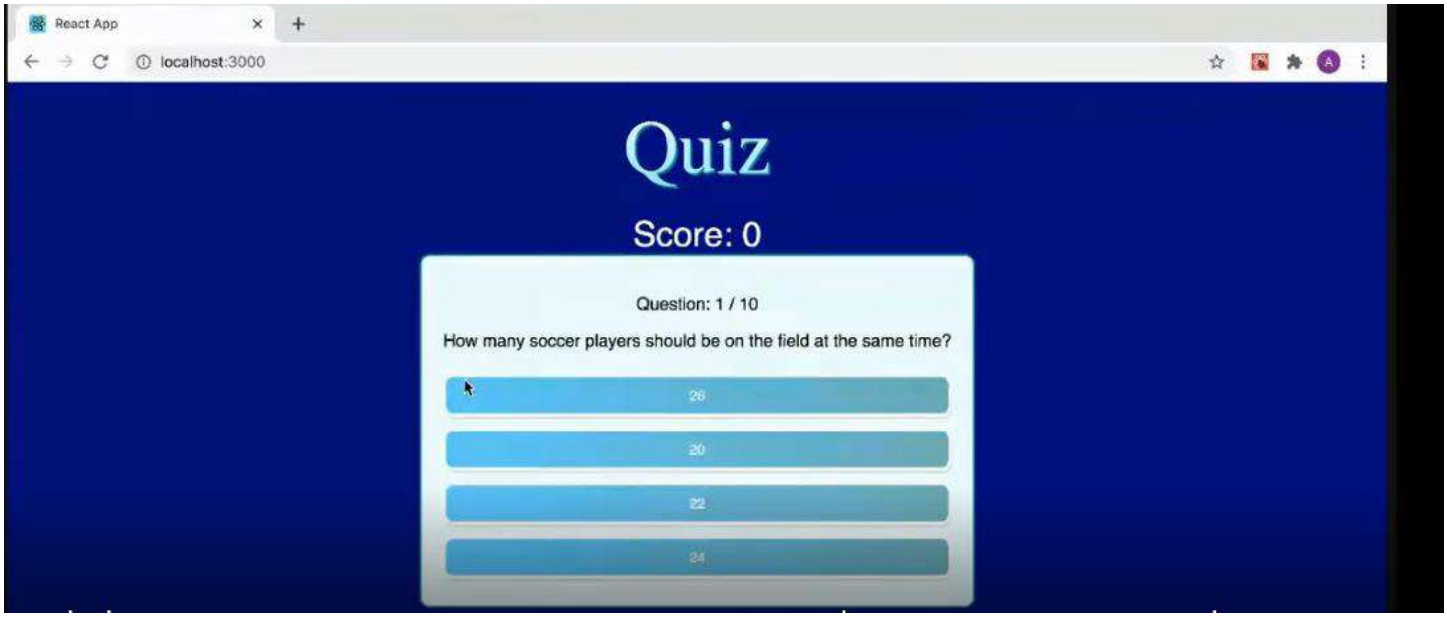
- Apply Wrapper and buttonWrapper for user OnClicked or correct answer choice as shown in QuestionCard.ts as shown in place of div:

```
TS QuestionCard.tsx X   TS QuestionCard.styles.ts
src > components > TS QuestionCard.tsx > QuestionCard
12
13  export const QuestionCard: React.FC<Props> = ({ question, answers, callback,
14      return (
15          <Wrapper>
```

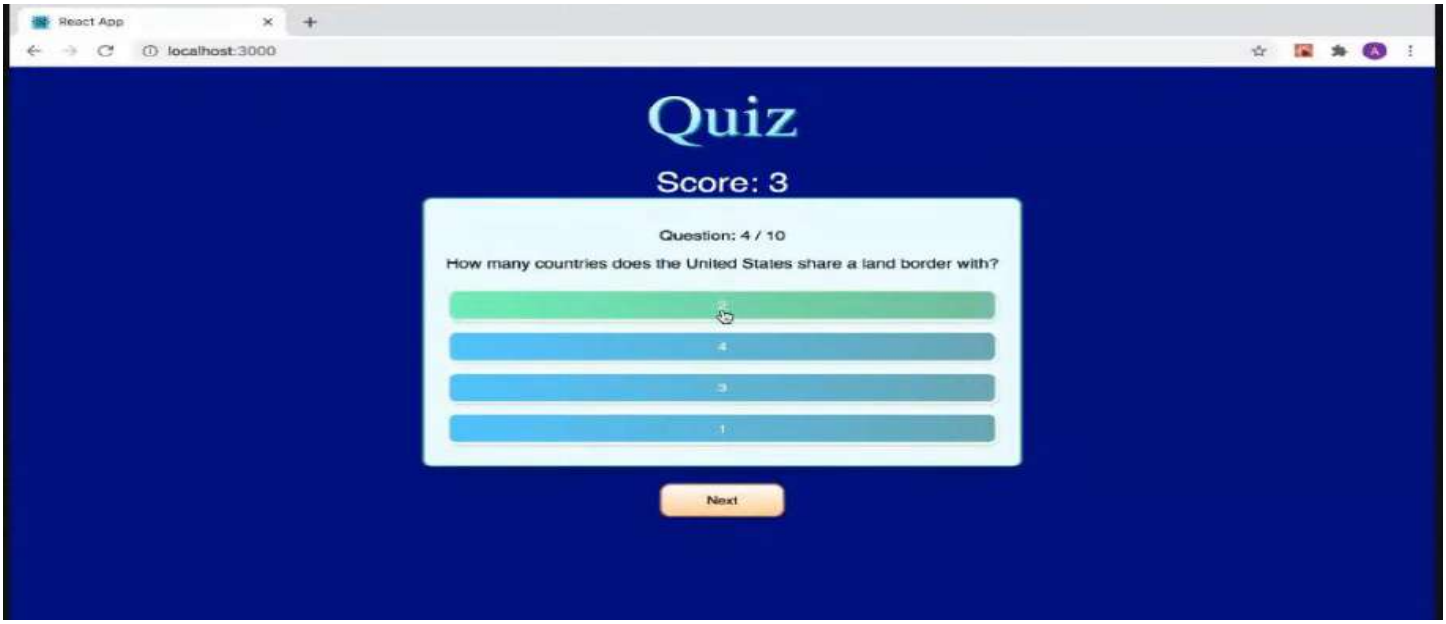
```
<Wrapper>
    <p>
        Question: {questionNum} / {totalQuestions}
    </p>
    <p dangerouslySetInnerHTML={{ __html: question}} />
    <div>
        {answers.map(answer => (
            <ButtonWrapper
                correct = {userAnswer?.correctAnswer === answer}
                userClicked = { userAnswer?.answer === answer }
            >
                <button disabled={userAnswer} value={answer} onClick=
                    <span dangerouslySetInnerHTML={{ __html: answer }}
                </button>
            </ButtonWrapper>
        ))}
    </div>
</Wrapper>
```

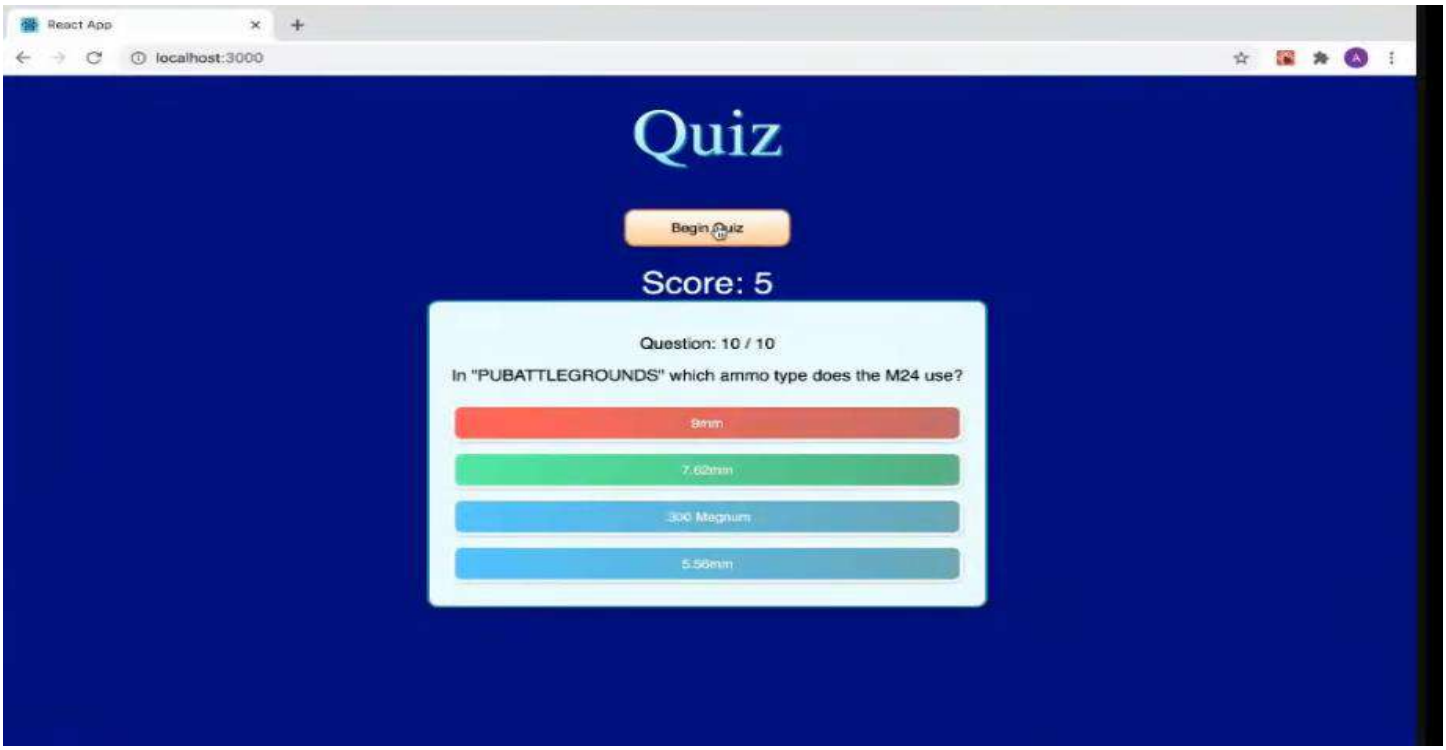
Activate Windows

- Look at the App Css has been Applied to all Question boxes



- Keep going by clicking to next button


React App
localhost:3000
Quiz
Score: 3
Question: 4 / 10
How many countries does the United States share a land border with?
4
3
1
Next


React App
localhost:3000
Quiz
Begin Quiz
Score: 5
Question: 10 / 10
In "PUBATTLEGROUNDS" which ammo type does the M24 use?
9mm
7.62mm
.300 Magnum
5.56mm

Session 14: Progressive Web Apps (PWA) and Deploying

1. **Progressive Web Application**

A progressive web application (PWA) is a type of application software delivered through the web, built using common web technologies including HTML, CSS, and javascript It is intended to work on any platform that uses a Standard-complaint-browser, including both desktop and mobile devices.

Features include

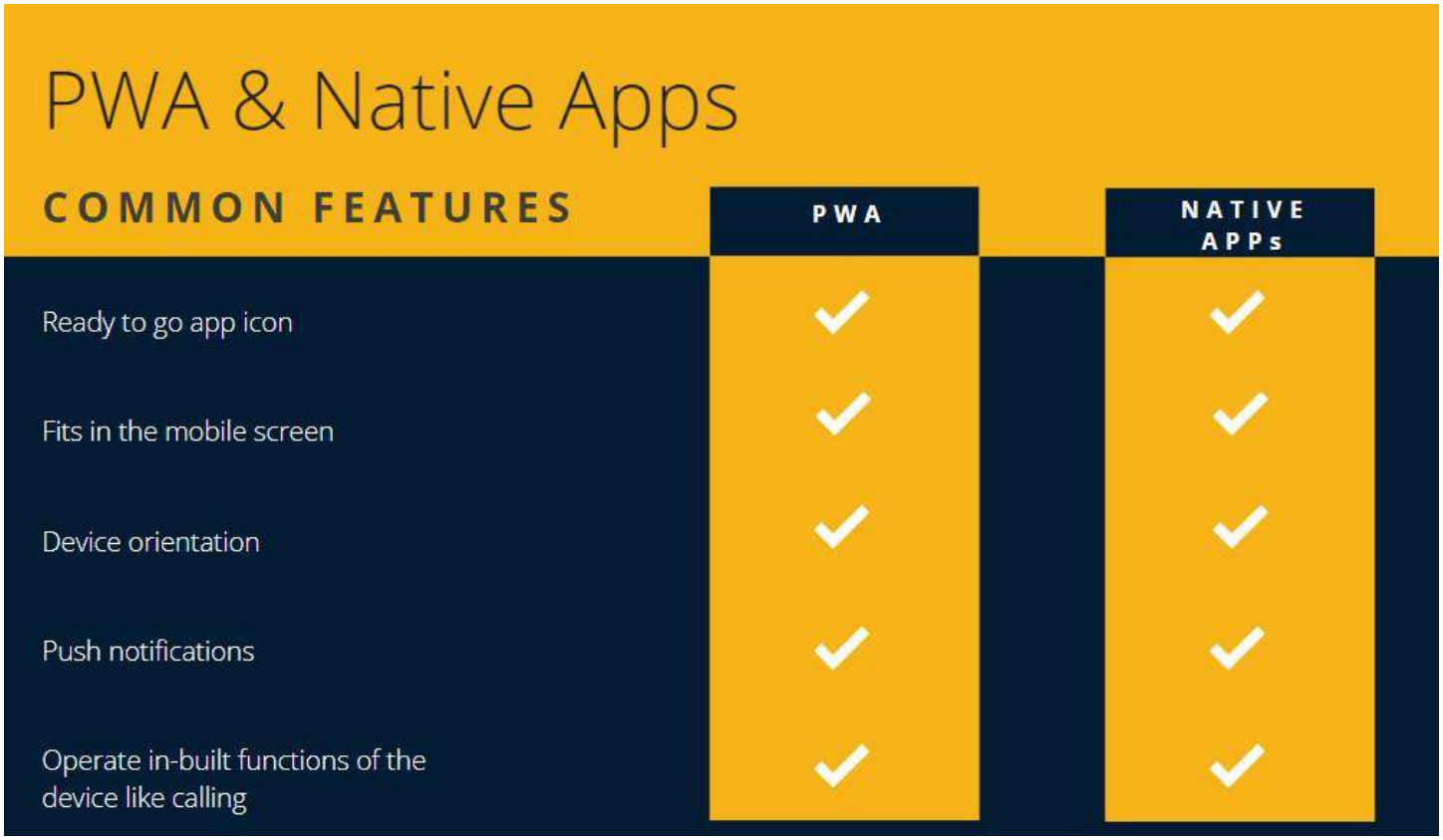
- Work offline
- High performance
- Background processing in service workers in a separate thread
- Access to the phone's sensors
- Support for push notifications
- An icon on the phone‘s home screen

A Progressive Web App is a website that looks and feels like a native mobile application without actually downloading it. The website offers the desired app-like functions and features directly on the website for mobile users. The PWAs are designed to resolve the limitations offered by the native apps while keeping the best of the customer experience in the line.

- the Progressive Web Apps are the best amalgamation of responsive websites and the native websites making it the better of the alternative. While Responsive sites are the original website with a flexible user interface, they lack the exclusive functionalities offered by the native apps.

- On the other hand, native apps are often too expensive and seek help from expert mobile app developers. PWAs catch up on the best parts of both worlds.

- In the following comparison chart, we bring out the common points that both Native apps and the PWA apps share:



Benefits of Progressive Web Apps

Progressive Web Apps bring out the best of Web browsing and the native apps to offer an excellent customer experience. The web app version goes past the limitations of the native apps to offer a smoother and faster experience. Above all, it doesn't ask for needless app permissions or harming your privacy. Here we enlist a few advantages, in brief, PWA carries over the Native apps:

"Progressive" in PWA

PWAs are designed to offer native-app like experience without the need for downloading them. The services are rendered directly via web browsers like Chrome, Firefox, and Safari (on iOS). The installable home button redirects the user to the web browser

Tools to develop a PWA by own
With PWAs catching up, a good number of developmental tools have popped up that offer a smooth and clutter-free User interface for PWAs. As the leading PWA development agency, The Brihaspati Infotech has implemented PWAs for several projects on various platforms. We have worked on popular frameworks and tools dedicated to PWA projects.

Here we enlist the essential parts for a successful PWA development:

Frameworks:

The first part of PWA's development is to choose a suitable framework. JavaScript leads the way with some of the best libraries for this highly responsive web app development.

- React
- Vue
- Angular
- Containers for your coding:

PWAs demand to repeat codes at multiple stages. Solutions to contain such codes and managing them is equally necessary. We have some popular containers as a solution:

- Redux
- Docker
- Workbox
- Module bundler:

Running large .js files without a proper bundler often undergoes major issues. Module bundlers help manage the large javascript files and streamline the entire flow. The tools we prefer are:

- Webpack
- Parcel.JS

Inspecting the PWA:

Executing the codes and implementing them all with success is a pen and paper thing until and unless they are tested on practical cases. These tools audit the PWA features and functionalities at the core.

We have a few tools that excel at testing the PWA performances in various aspects:

- Devtools by Chrome
- Lighthouse

The workflow for developing PWA

Having said that there are plenty of options available for PWA development, we will explain to you the development cycle with the help of one technology that prevails in all aspects-React. The codes mentioned below might differ from the framework chosen. However, the basic outline and the reason for adding modules remain the same.

Step1: Creating App shell

The Application shell plays a crucial role in presenting the Progressive features of the app. The user finds it at first. Thus, it needs to be fast and shall render the pages immediately.
The developer can create a PWA surrounding from the first call by applying the following codes:

To fetch details about the gathered information, the developers need to implement React router. Where they can utilize technologies like Redux.

Step 2: Implement a Service Worker

A Service worker plays as a middle man between the application and the network. The Service worker stores the cache from the network and renders them even during the offline access. The user can thus decide the sufficiency of the available data by the app or the need for turning the connectivity on.

The part also finds itself useful at rendering the PWA functionalities like Push notifications and prompting to perfection.

Step 3: Adding Push Notifications

Push Notifications completely take the web apps a step closer to the native apps. Developers can

add them directly with push APIs. The developers can call push manager commands from within the service worker file to initiate this task.

Step4: Web App Manifest

The PWAs need to be installable. The inclusion of a Manifest JSON file at the root directory can help you at this. The file plays a key role in the kind of interface you render to your users-icons, the background color, theme, and much more.

Step 5: Analyse the Application performance

With the app being ready to launch, the developer will need to analyze the performance of the application with the help of audit tools like Lighthouse. The Tools help you analyze the speed and compatibility on the devices.

2. Register A Service Worker

To install a service worker you need to kick start the process by registering it in your page. This tells the browser where your service worker JavaScript file lives.

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', function() {
    navigator.serviceWorker.register('/sw.js').then(function(registration) {
      // Registration was successful
      console.log('ServiceWorker registration successful with scope: ', registration.scope);
    }, function(err) {
      // registration failed :(
      console.log('ServiceWorker registration failed: ', err);
    });
  });
}
```

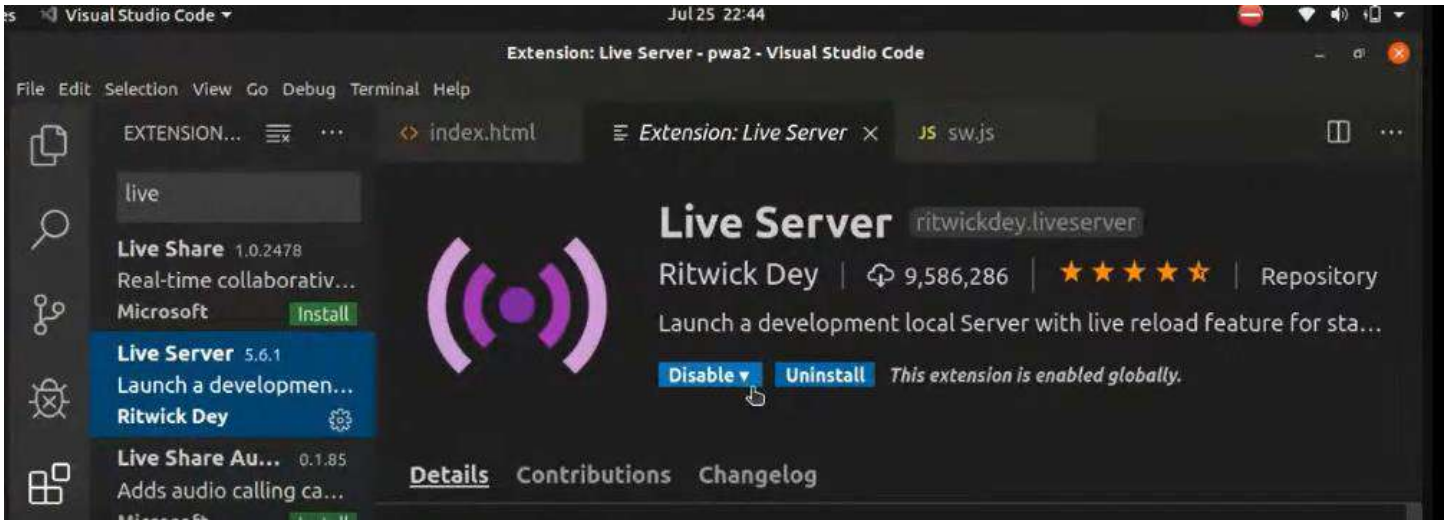
How does it work?

```
if ('serviceWorker' in navigator) {
```

- this will indicate whether the service worker is supported to our browser or not if it is in your browser then this piece of code will work and will install sw.js

```
navigator.serviceWorker.register('/sw.js').then(function(registration) {
```

- and in this file of sw.js, we will do multiple things like
  - push notification
  - offline support etc
- **what we need to do just push this code in simple html structure and add Extention of live server and install it, it will show a port below you can go online to see if this service is working or not**

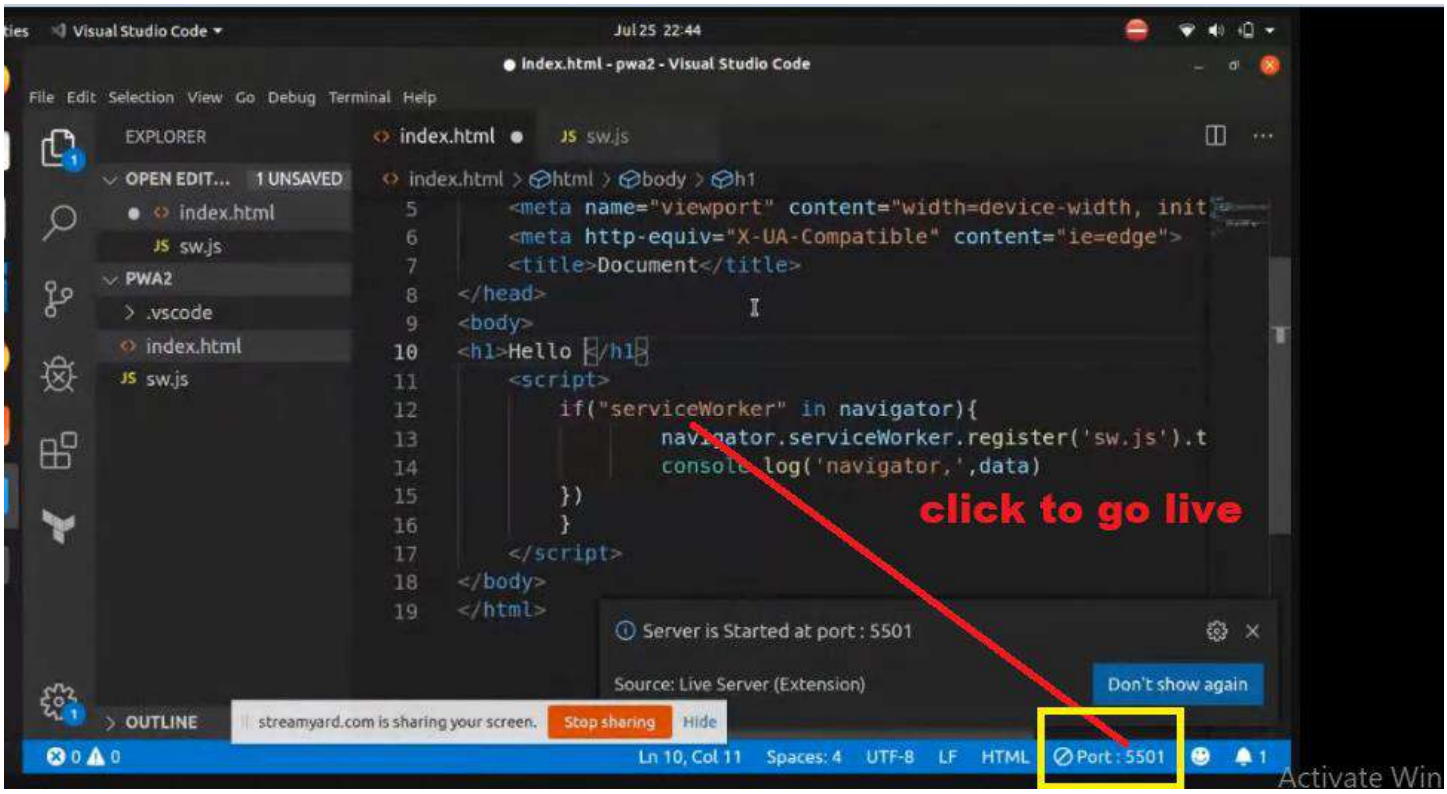


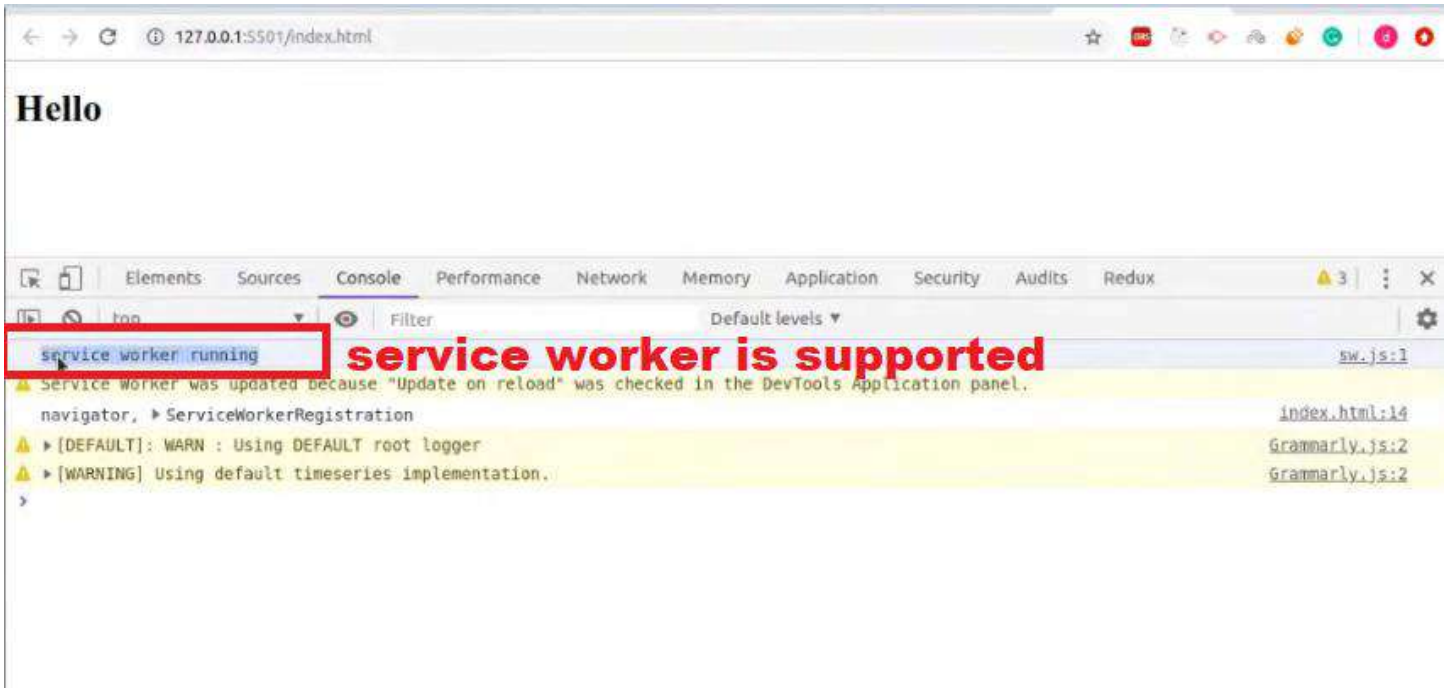
Index.html

```
!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UFT-8">
  <meta name="viewport" content
="width=device-width,init
  <meta http-equiv="X-UA-compatible" content
="ie=edge">
  <title>Document</title>
</head>
<body>
   <h1> Hello</h1>
   <script>
   if ('serviceWorker' in navigator) {

navigator.serviceWorker.register('/sw.js').then(fun
ction(registration) {

      console.log('navigator, ',data
  });
}

   </script>
</body>
```

sw.js

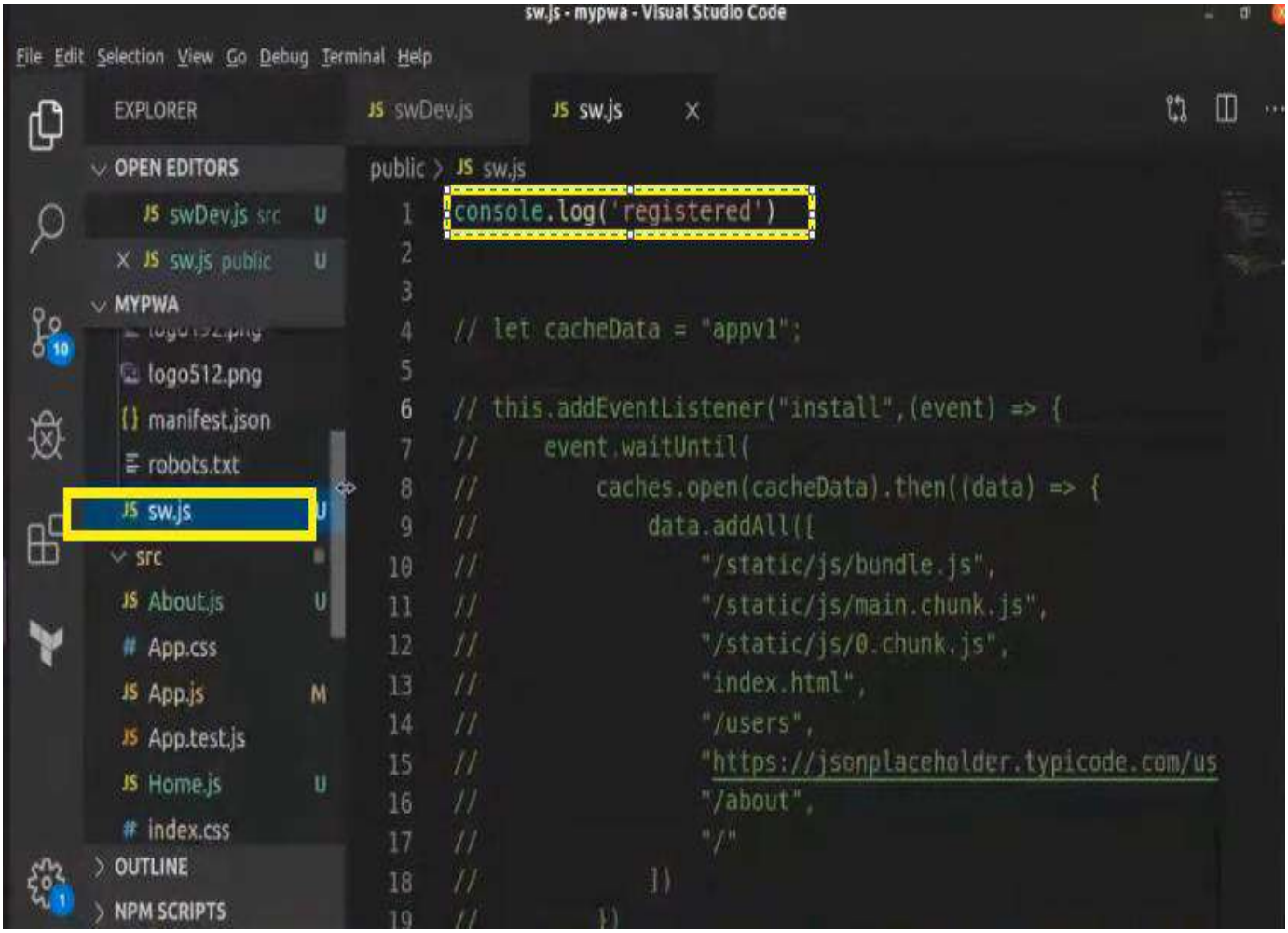
```
console.log ("service worker running")
```


Visual Studio Code
Jul 25 22:44
index.html - pwa2 - Visual Studio Code
File Edit Selection View Go Debug Terminal Help
EXPLORER
index.html
sw.js
OPEN EDIT... 1 UNSAVED
index.html
sw.js
PWA2
.vscode
index.html
sw.js
OUTLINE
<meta name="viewport" content="width=device-width, init
<meta http-equiv="X-UA-Compatible" content="ie=edge">
<title>Document</title>
</head>
<body>
<h1>Hello</h1>
<script>
if("serviceWorker" in navigator){
navigator.serviceWorker.register('sw.js').t
console.log('navigator,',data)
})
}
</script>
</body>
</html>
click to go live
Server is Started at port : 5501
Source: Live Server (Extension)
Don't show again
streamyard.com is sharing your screen.
Stop sharing
Hide
Ln 10, Col 11
Spaces: 4
UTF-8
LF
HTML
Port : 5501


127.0.0.1:5501/index.html
Hello
Elements
Sources
Console
Performance
Network
Memory
Application
Security
Audits
Redux
top
Filter
Default levels
service worker running
service worker is supported
sw.js:1
Service Worker was updated because "Update on reload" was checked in the DevTools Application panel.
navigator, ▸ ServiceWorkerRegistration
index.html:14
▸ [DEFAULT]: WARN : Using DEFAULT root logger
Grammarly.js:2
▸ [WARNING] Using default timeseries implementation.
Grammarly.js:2

3. Build A React Application

- create a simple sw.js file that registers our service
- in it, you can do whatever you want
  - run
  - all features that PWF provide us

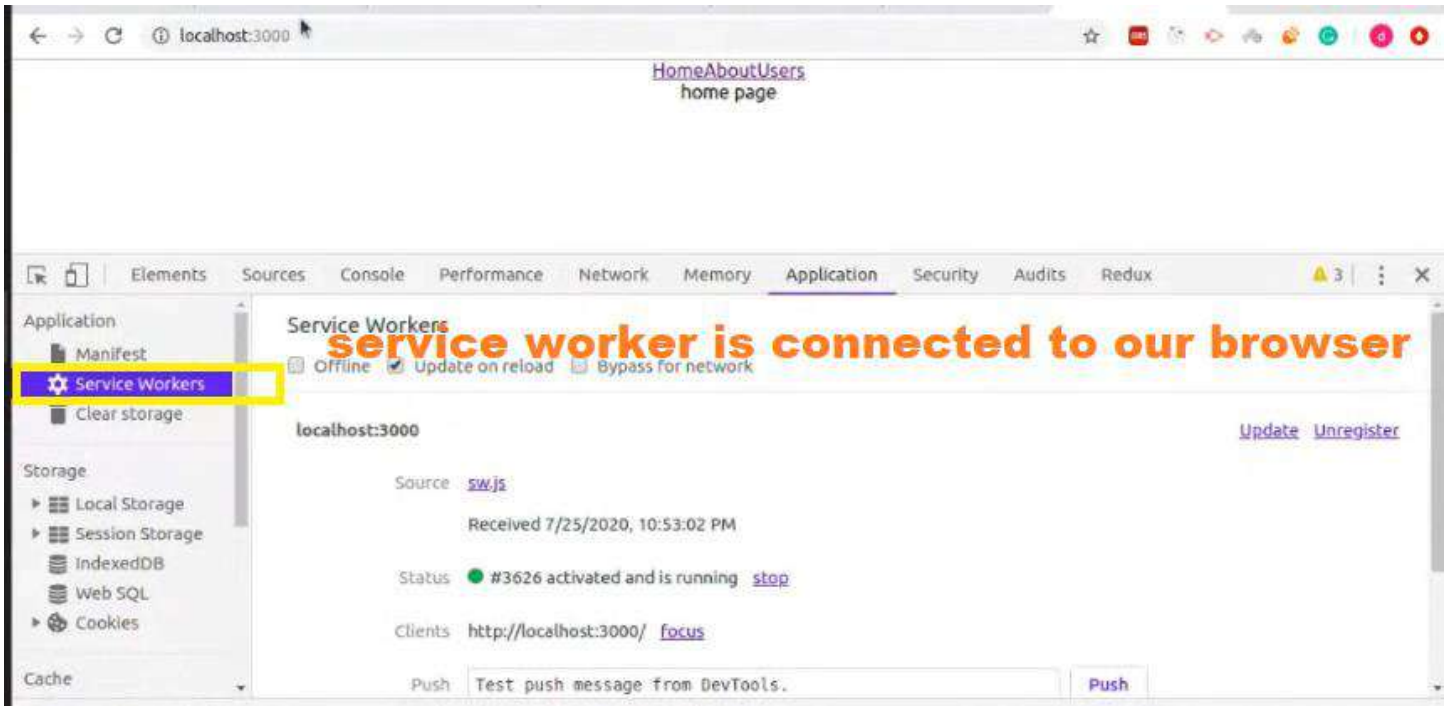


```
// this.addEventListener("fetch",(event) => {
//     if(!navigator.onLine){
//         event.respondWith(
//             caches.match(event.request).then((result)=>
//                 console.log('results',result);
//                 if(result){
//                     return result;
//                 }
//             })
//         )
//     }
// })
```

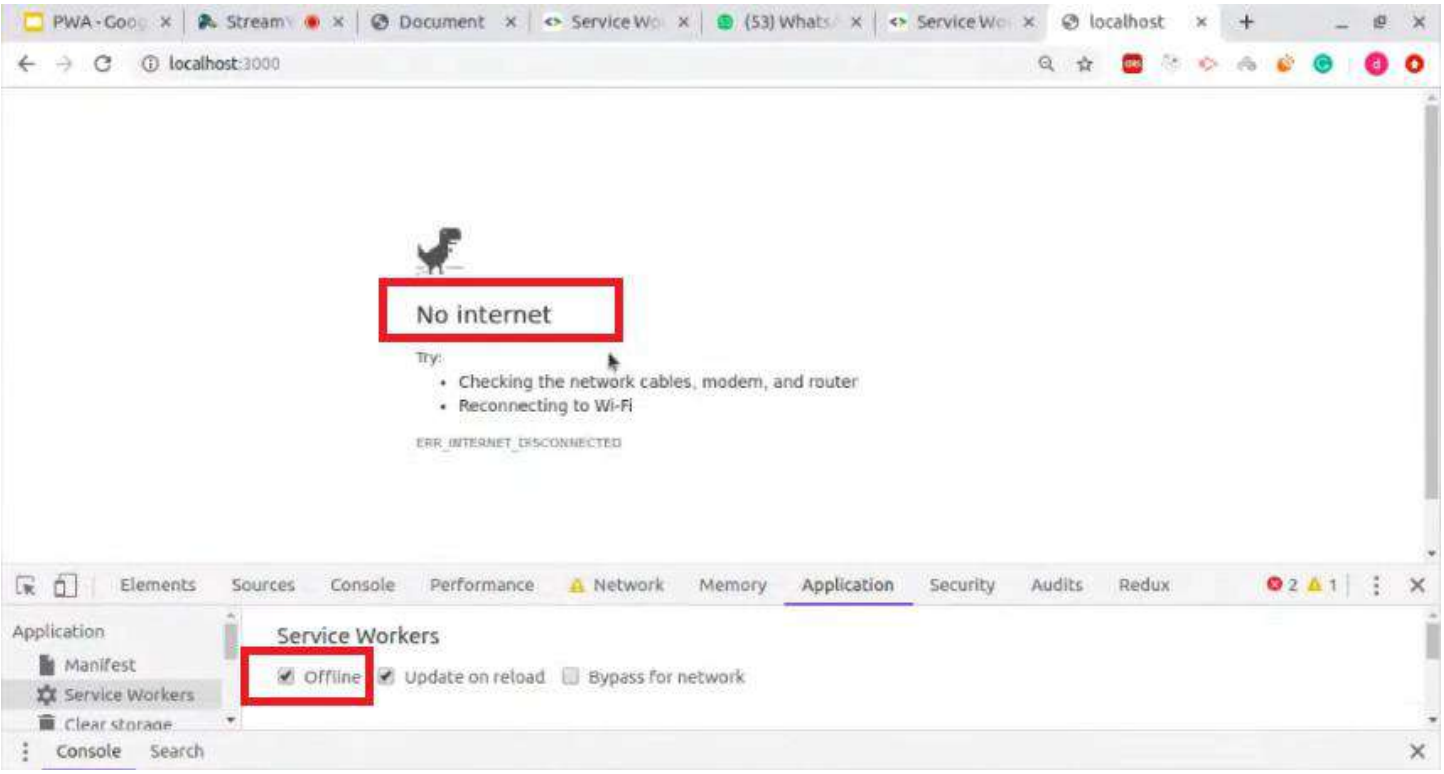
- Now create swDev.js that will provide navigator.serviceWorker.register(swDev)
- Also, provide path ${process.env.PUBLIC_URL}/sw.js
- Add cache if there is an error

```
export default function swDev(){
    let swDev = `${process.env.PUBLIC_URL}/sw.js`
    navigator.serviceWorker.register(swDev).then((result) =
        console.log('result',result);
    })
    .catch((err) => {
        console.log('err',err);
    })
}
```

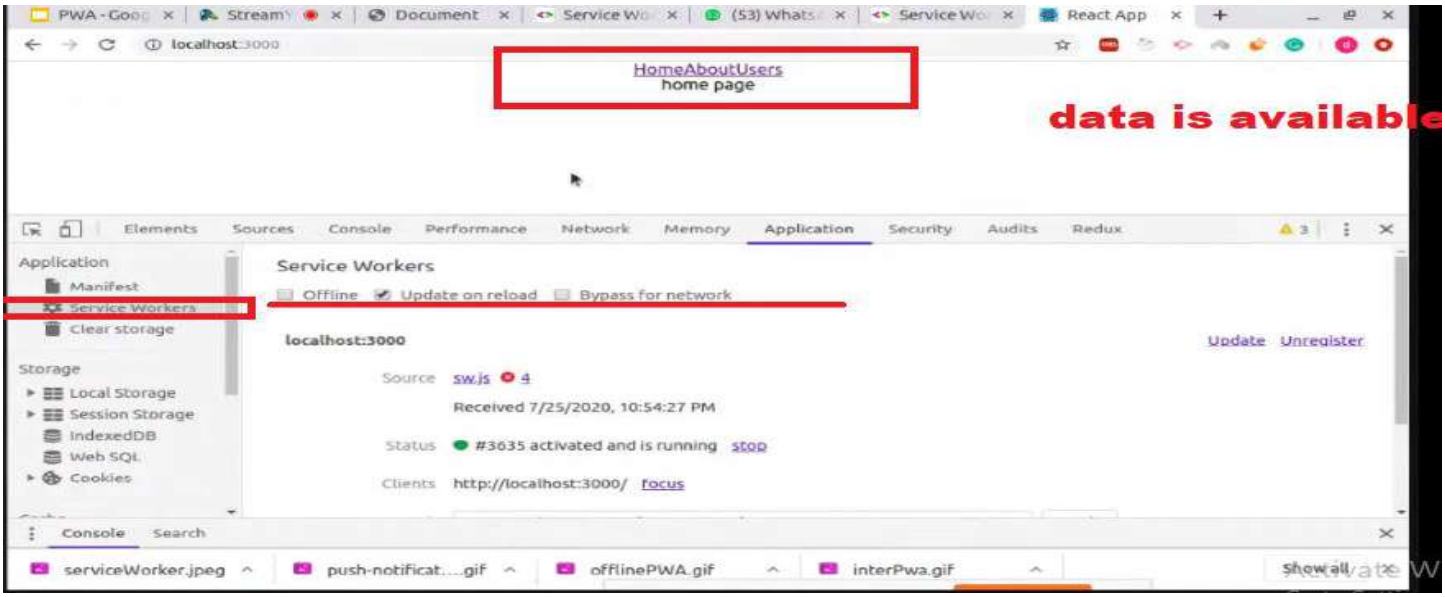
Now go to the browser and check as to whether the service worker is supported by our browser or not



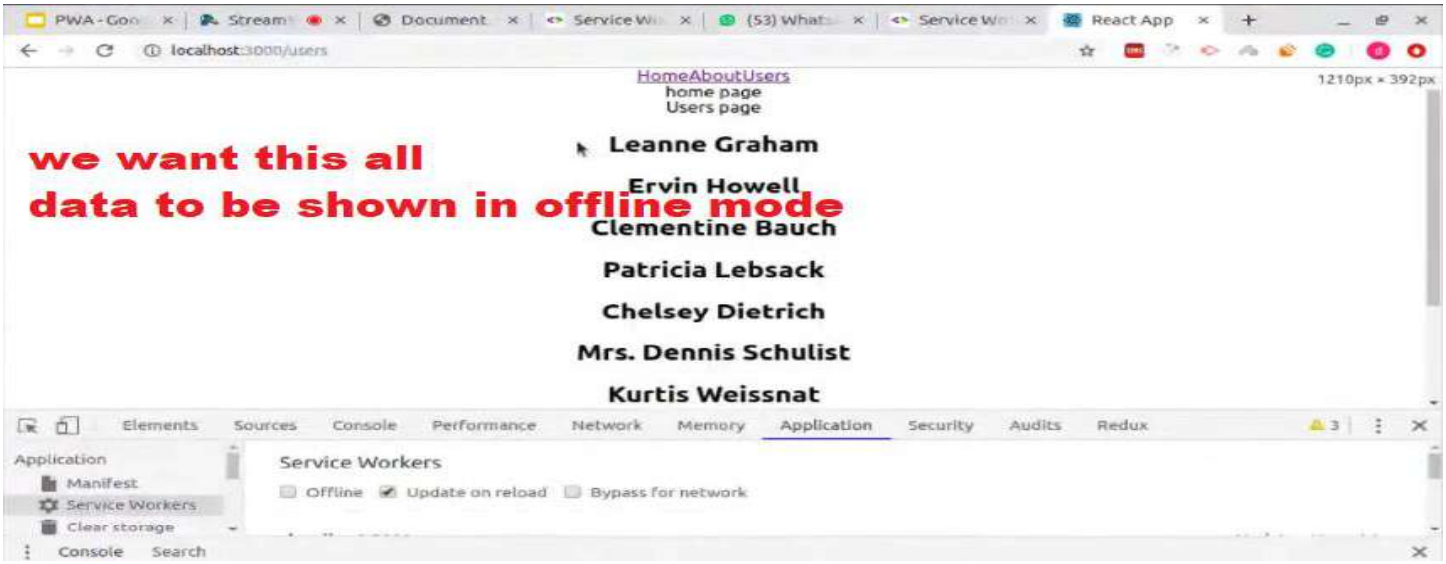
- Now click to “offline” to disconnect the serviceworker from the browser and we will see offline activity by refreshing the page as no data is available

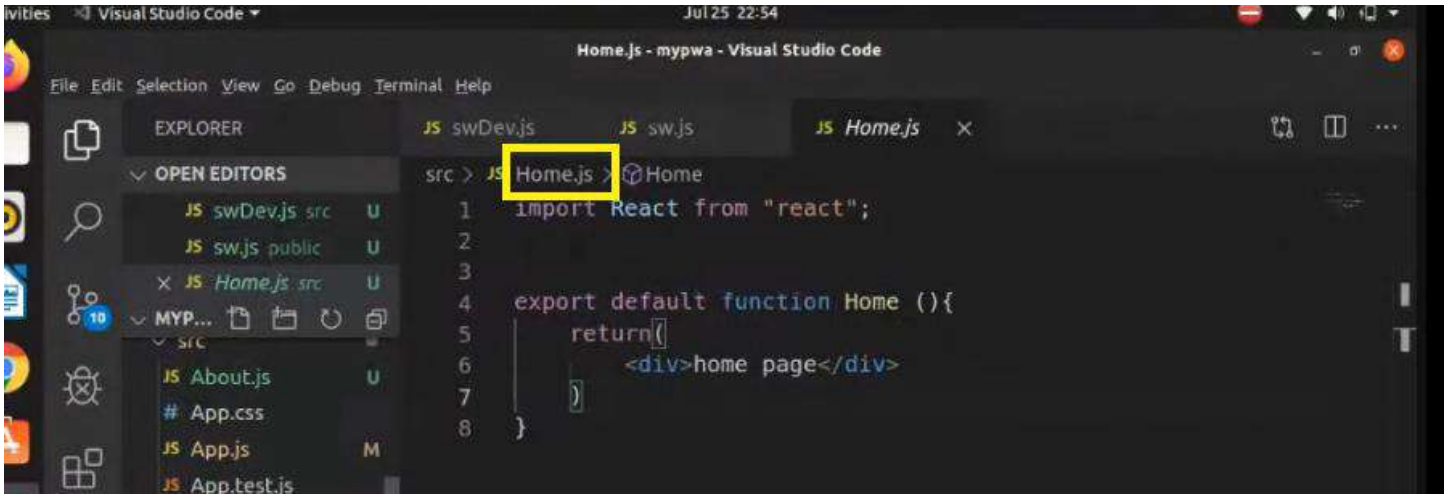


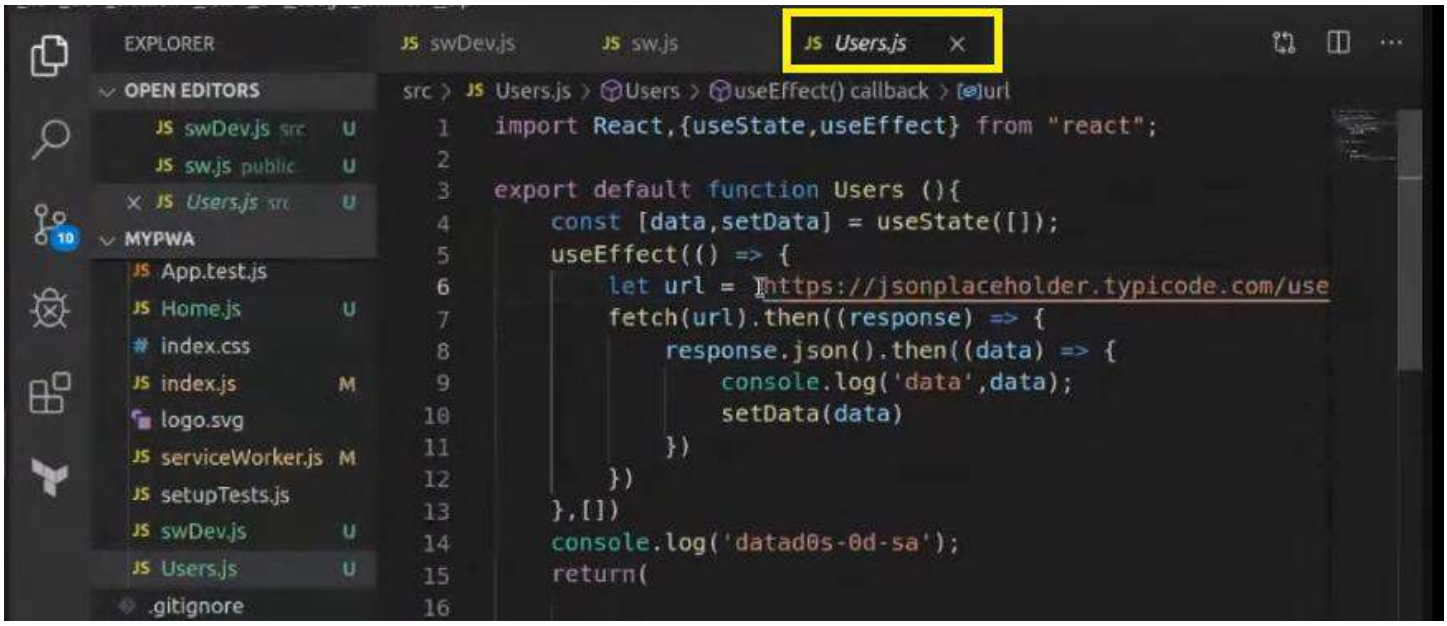
- Again now check out offline mode to go online and refresh the page then you will see the data Is available

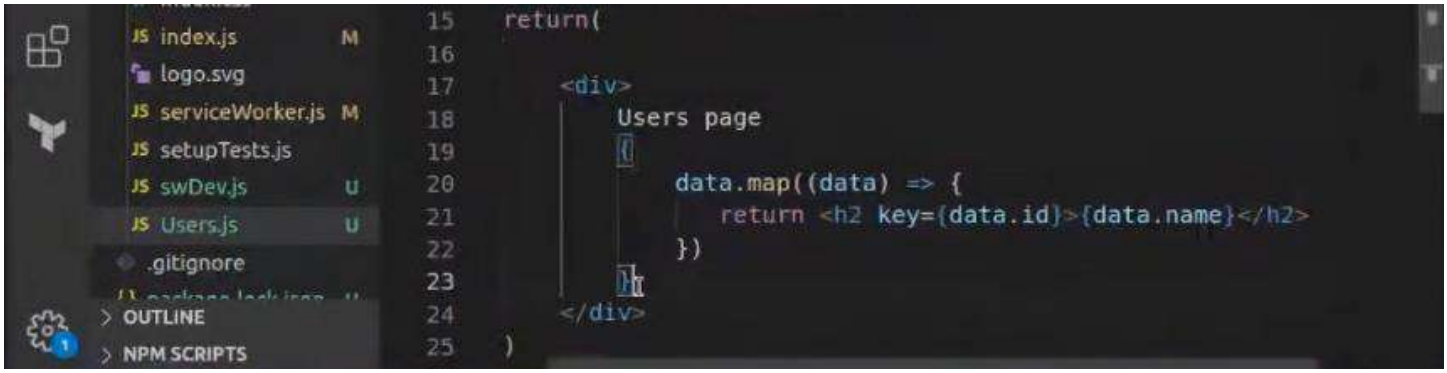


- Now our task is to achieve all data in offline mode

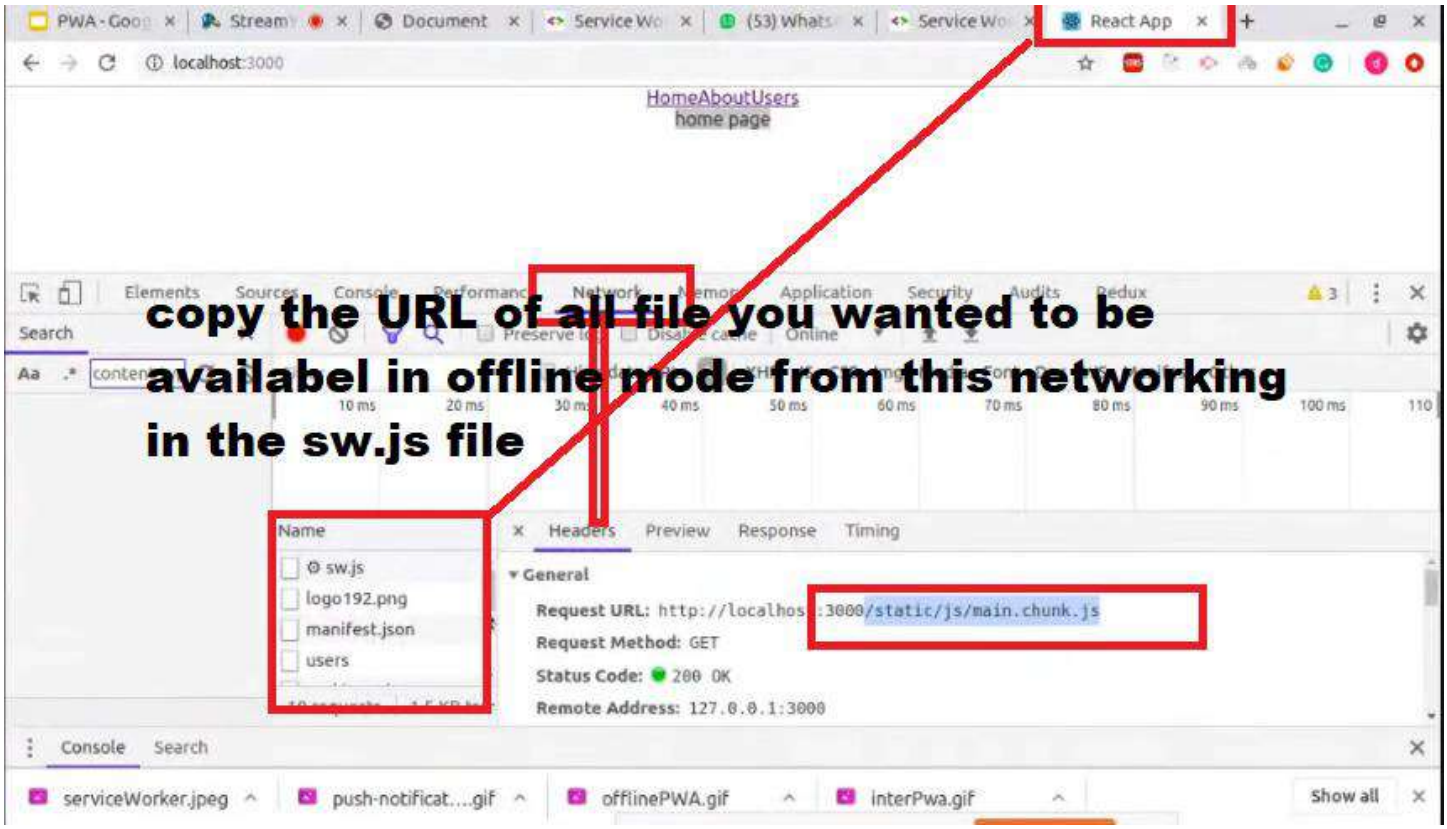


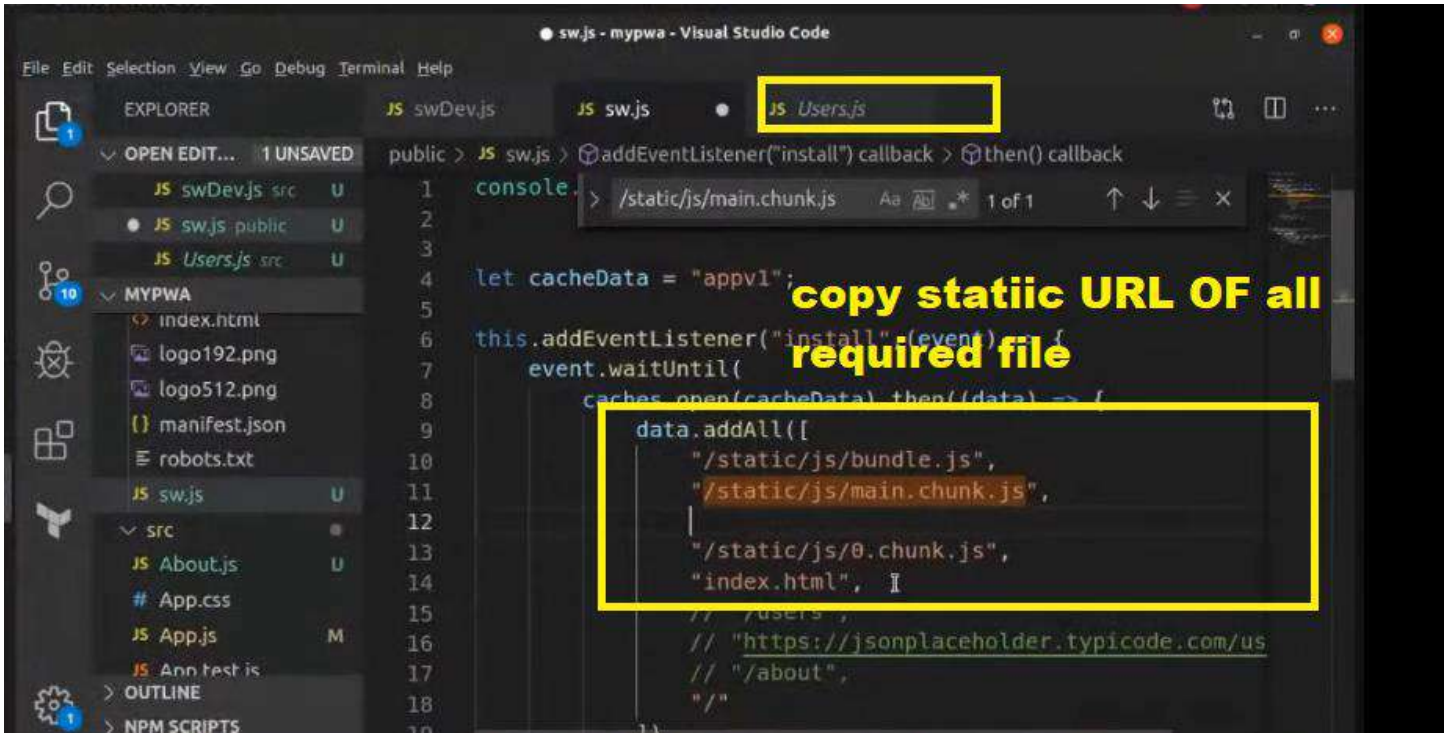




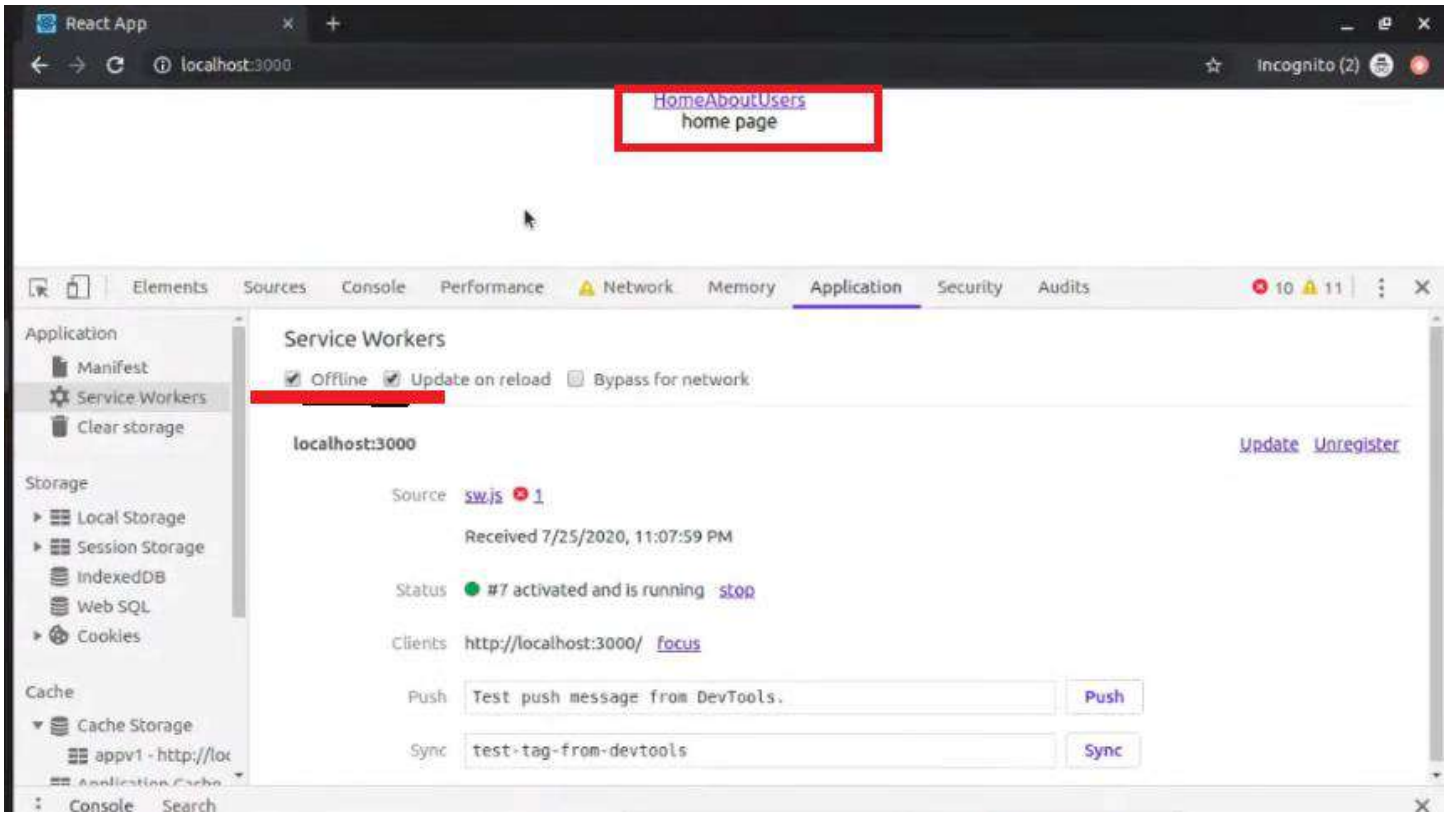


- Copy the URL of all file like home.js, App.js,nodes.js, etc from the networking as shown and paste it in the sw.js file of data as shown


copy the URL of all file you wanted to be
availabel in offline mode from this networking
in the sw.js file
Request URL: http://localhost:3000/static/js/main.chunk.js
Request Method: GET
Status Code: 200 OK
Remote Address: 127.0.0.1:3000


copy statiic URL OF all
required file
"/static/js/bundle.js",
"/static/js/main.chunk.js",
"/static/js/0.chunk.js",
"index.html",

- So look at the display in offline mode it is working ;

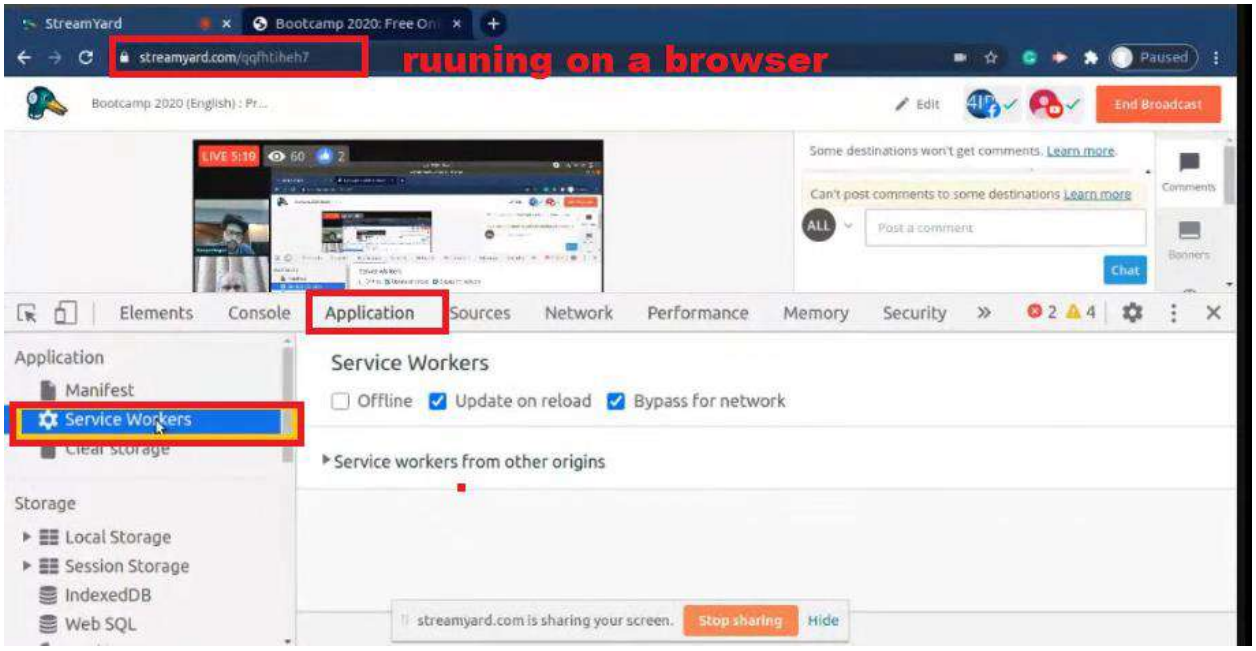


Now you can cache about data, user data, and Jason data to look it in offline mode just by simply following the above rule

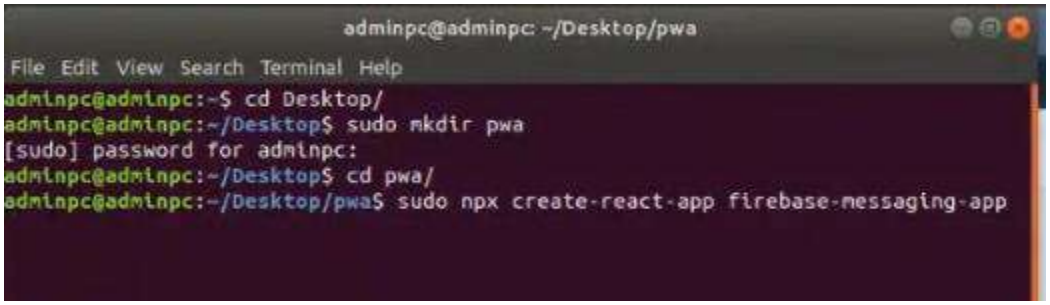
Session 14(A): Progressive Web Apps (PWA) and Deploying

Push Notification Using PWA With Service Worker

- it is not a programming language
- it is a configuration of web Application to give multiple features
- it will work in the run state in any of the browsers and the main thing is that it includes service work where we can configure everything like:
  - cache the application
  - use push notification



- open terminal
- create a react project with folder name on the desktop
- go to the folder as shown as an example

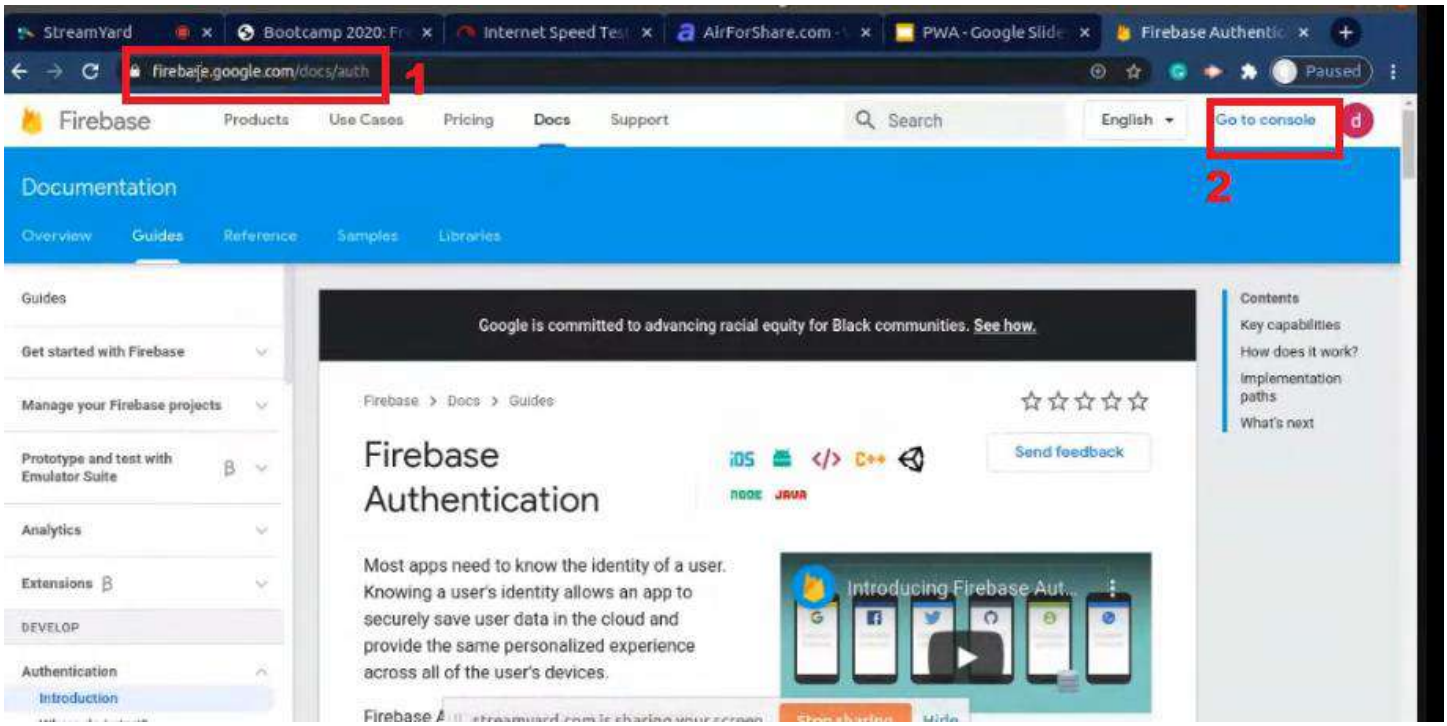


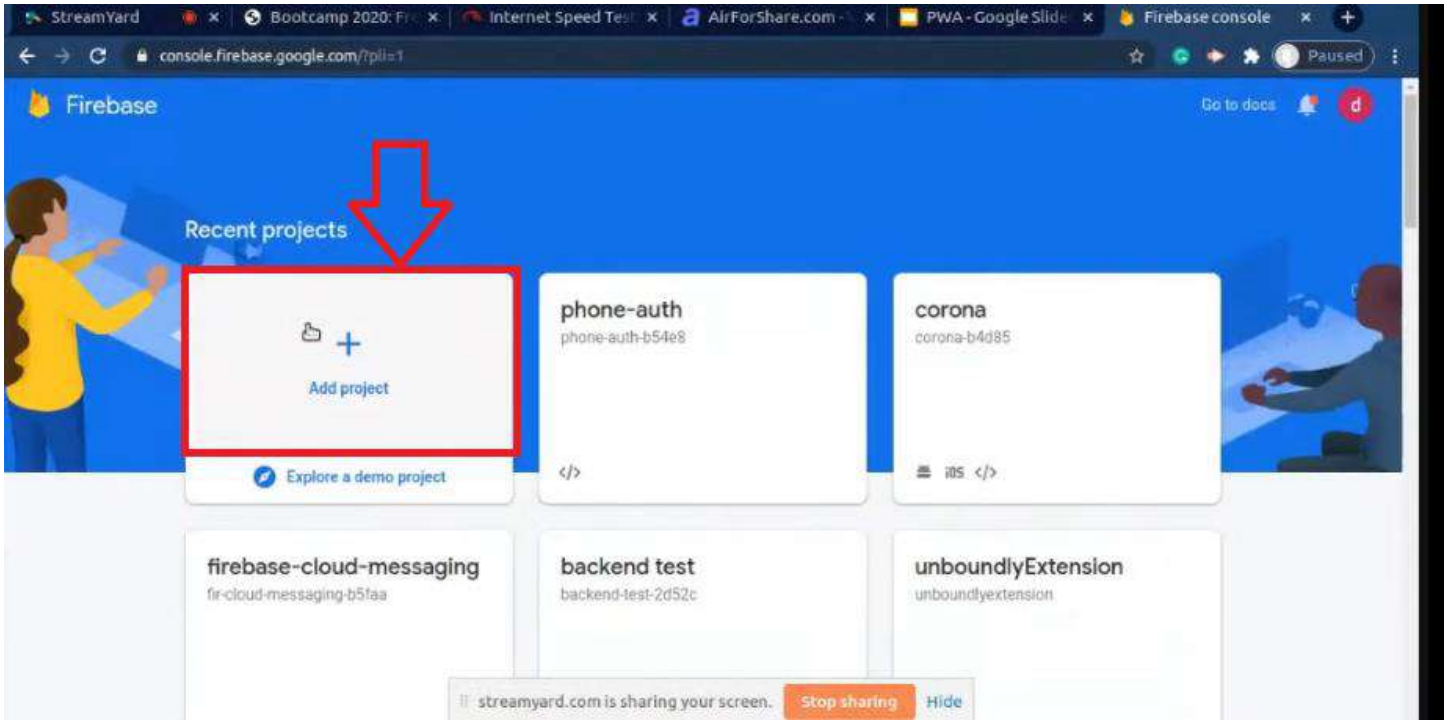
- It will take a few minutes to create the project

```
adminpc@adminpc: ~/Desktop/pwa/firebase-messaging-app
File  Edit  View  Search  Terminal  Help
  yarn start
    Starts the development server.

  yarn build
    Bundles the app into static files for production.

  yarn test
    Starts the test runner.

  yarn eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd firebase-messaging-app
  yarn start

Happy hacking!
adminpc@adminpc:~/Desktop/pwa$ ls
firebase-messaging-app
adminpc@adminpc:~/Desktop/pwa$ cd firebase-messaging-app/
adminpc@adminpc:~/Desktop/pwa/firebase-messaging-app$ code .
```

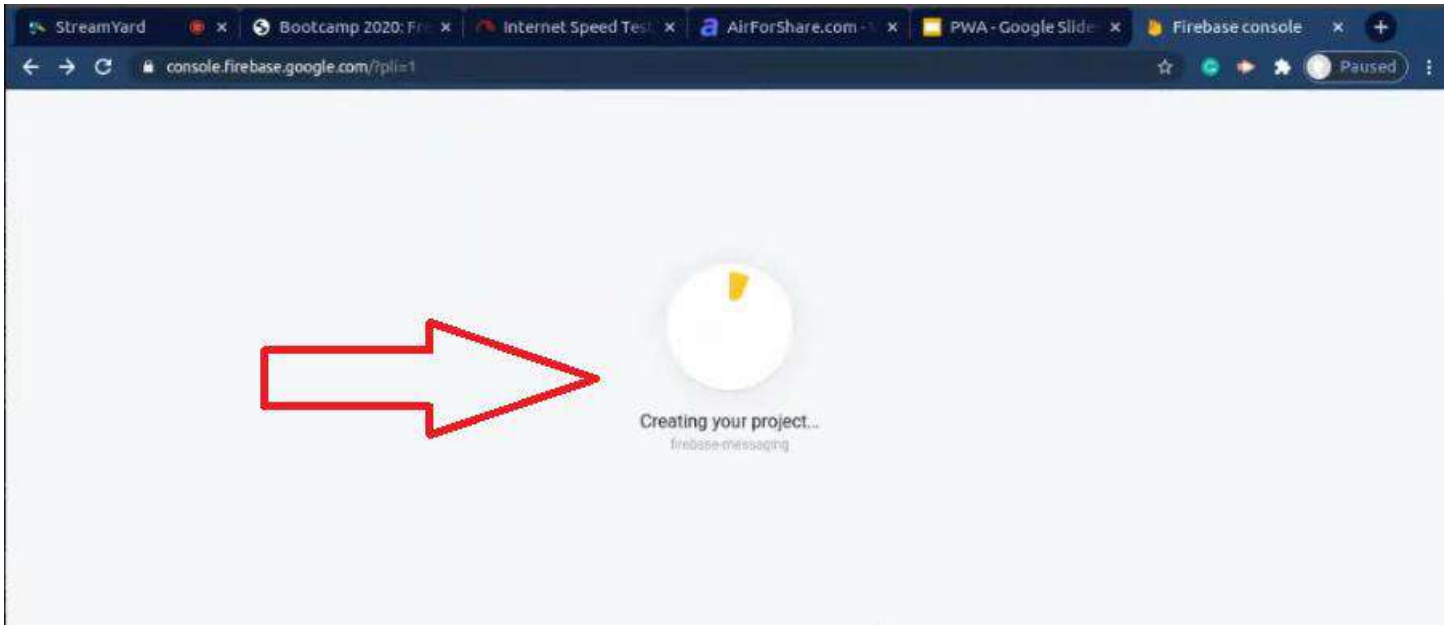
Note from the server we will send the notification and on the client site we receive it, so the easiest way to implement push notification is firebase that is we are using a firebase there. Basically it is a service of cloud messaging by google

- since we are working on firebase so login for firebase and go to console log
- Click to add project

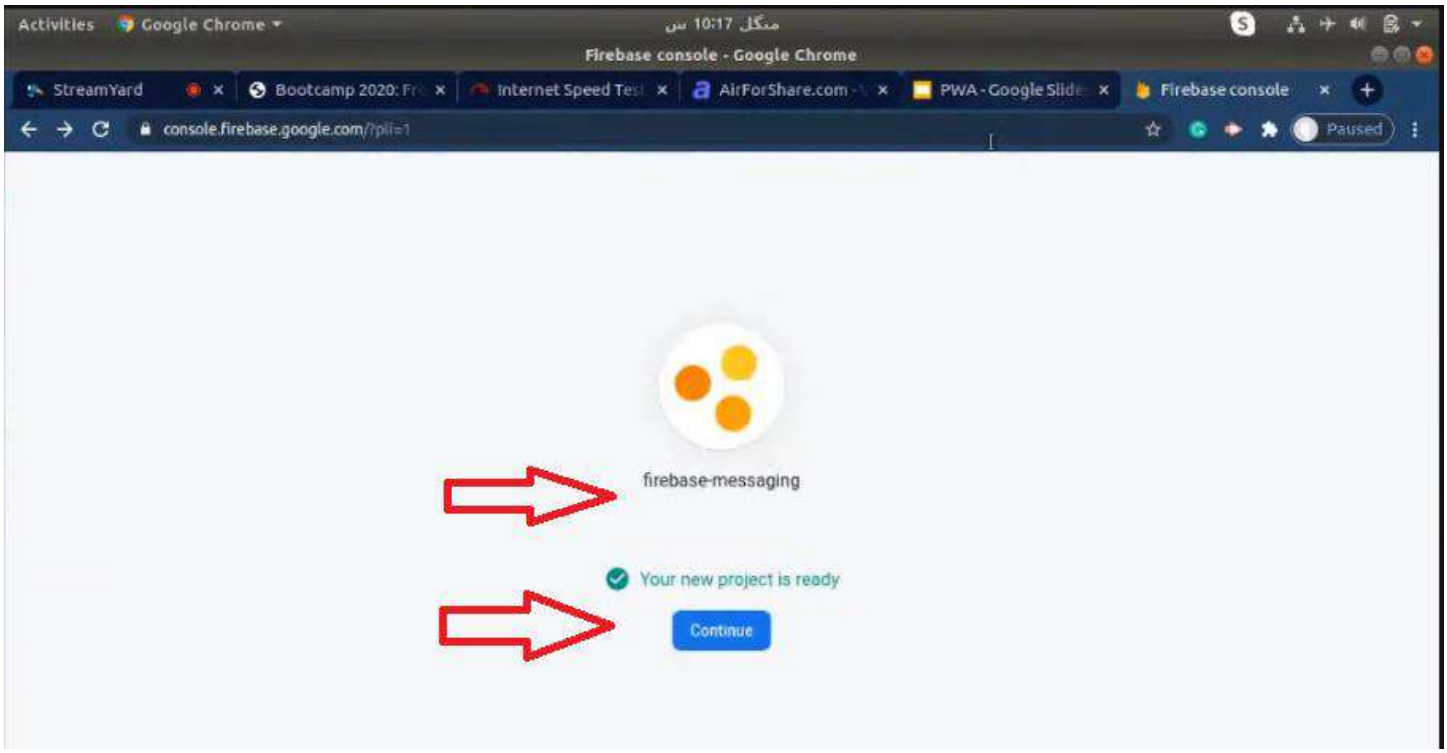




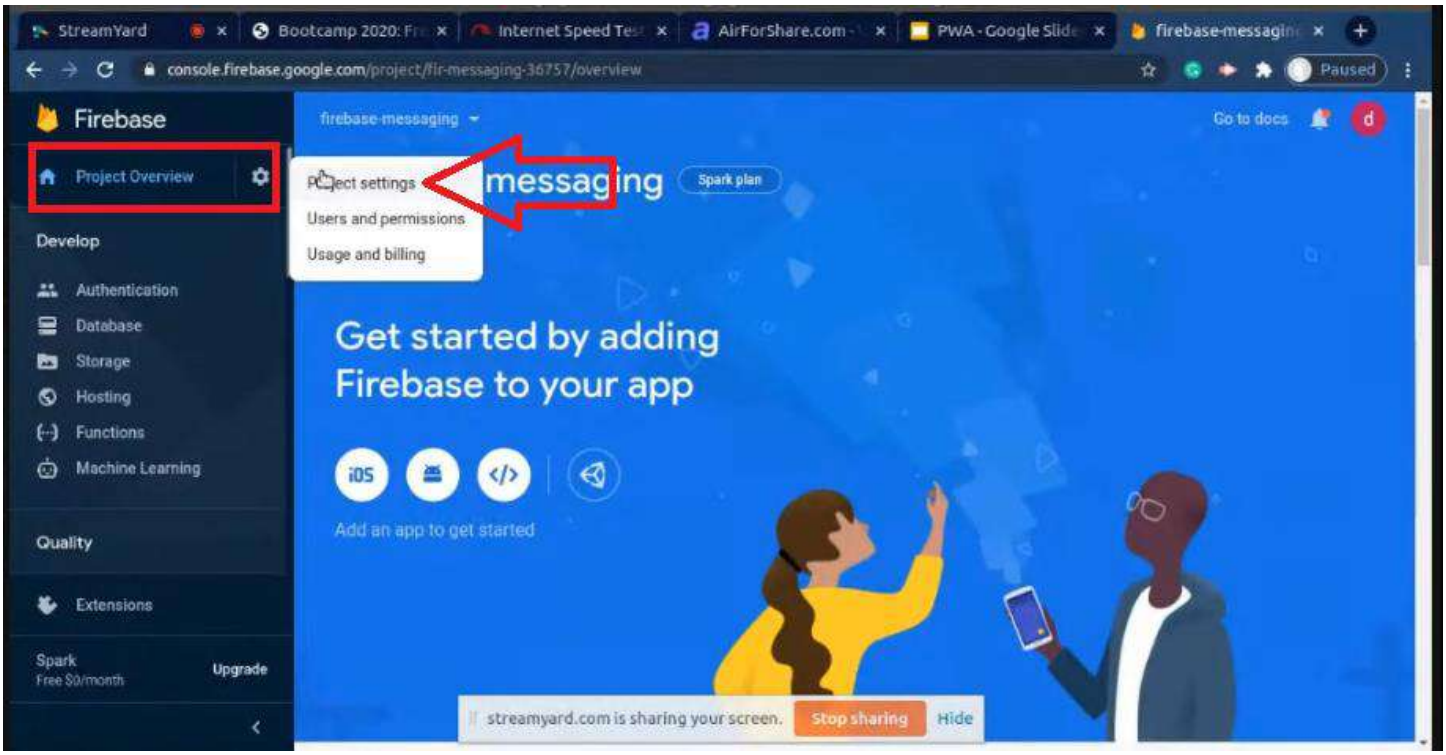
- Click on add project and create a project by writing the project name as “firebase _ messaging”, and click to continue then click to create a project on the next pop up



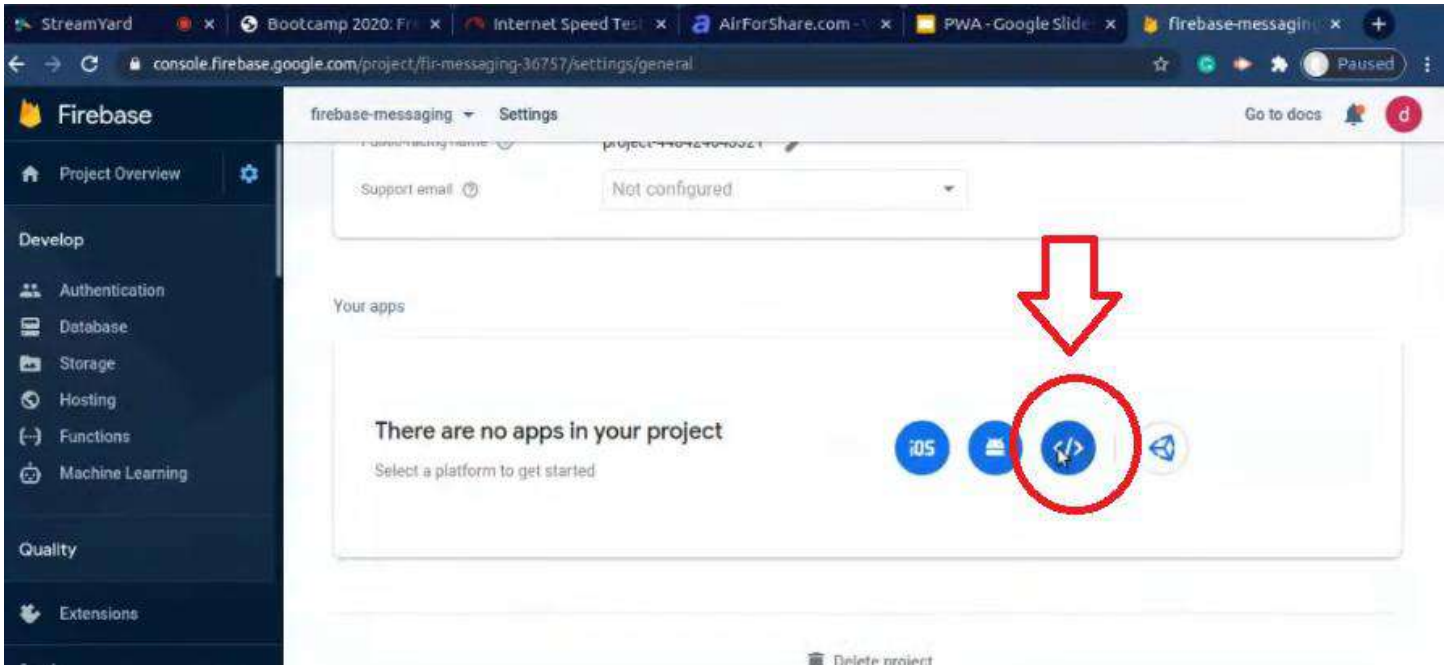
- The new project is ready

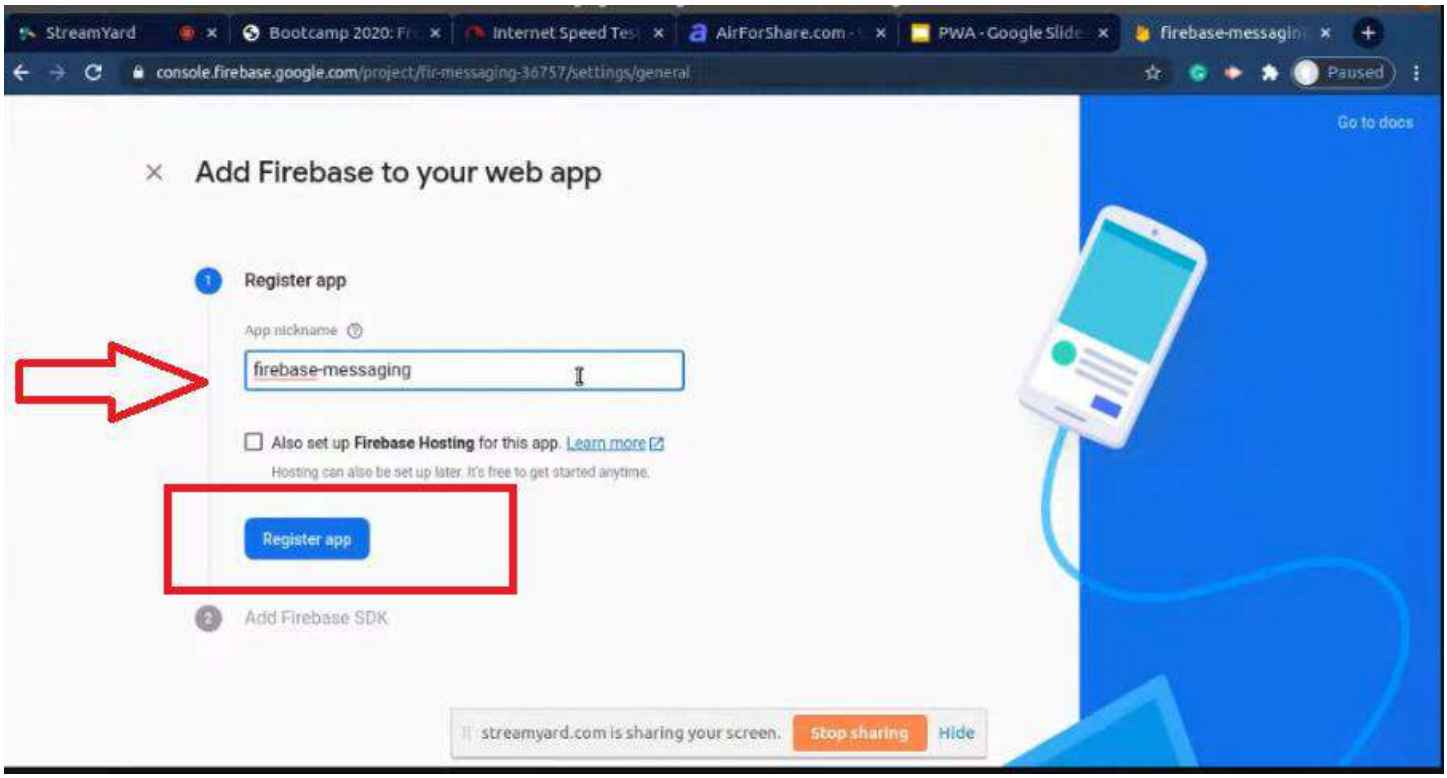


- Click to continue
- Go to project overview >project setting

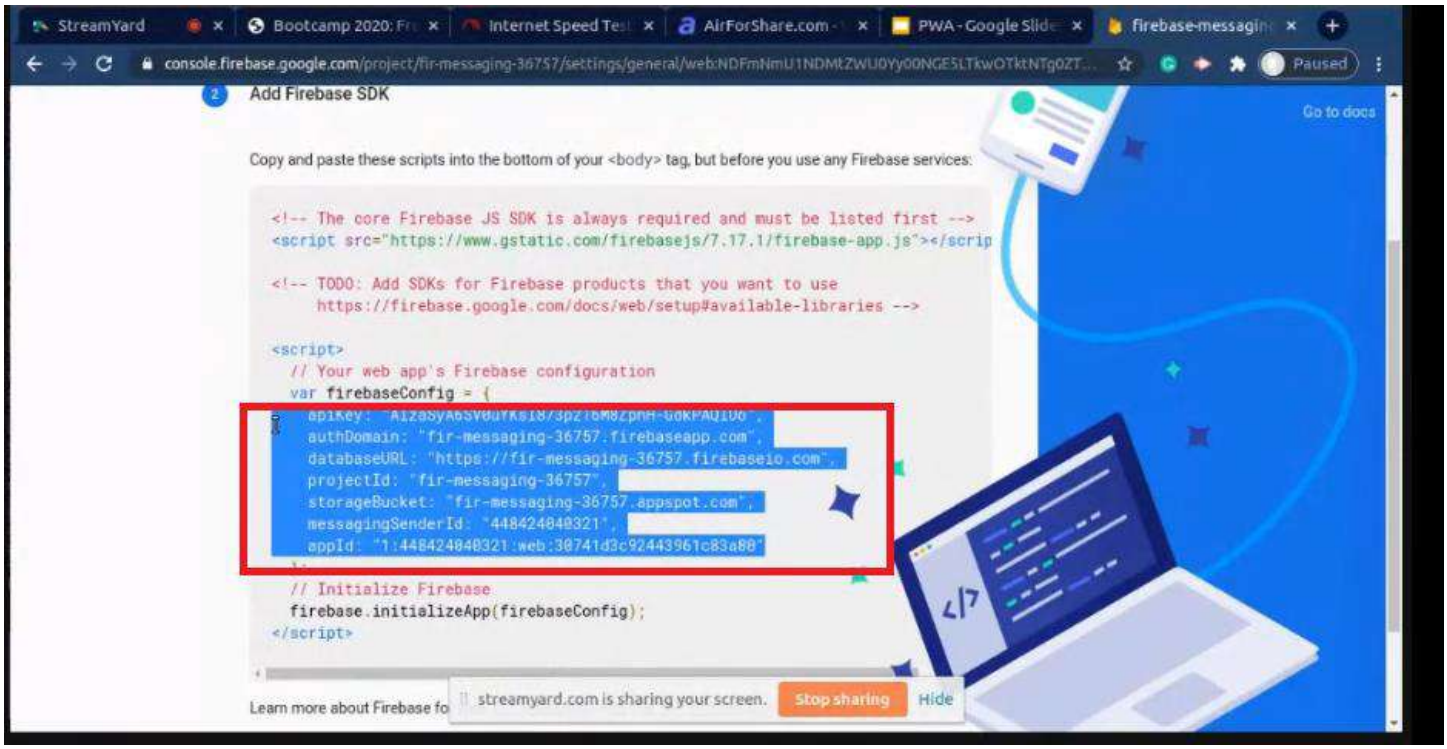


- Click on this icon
- Register your app

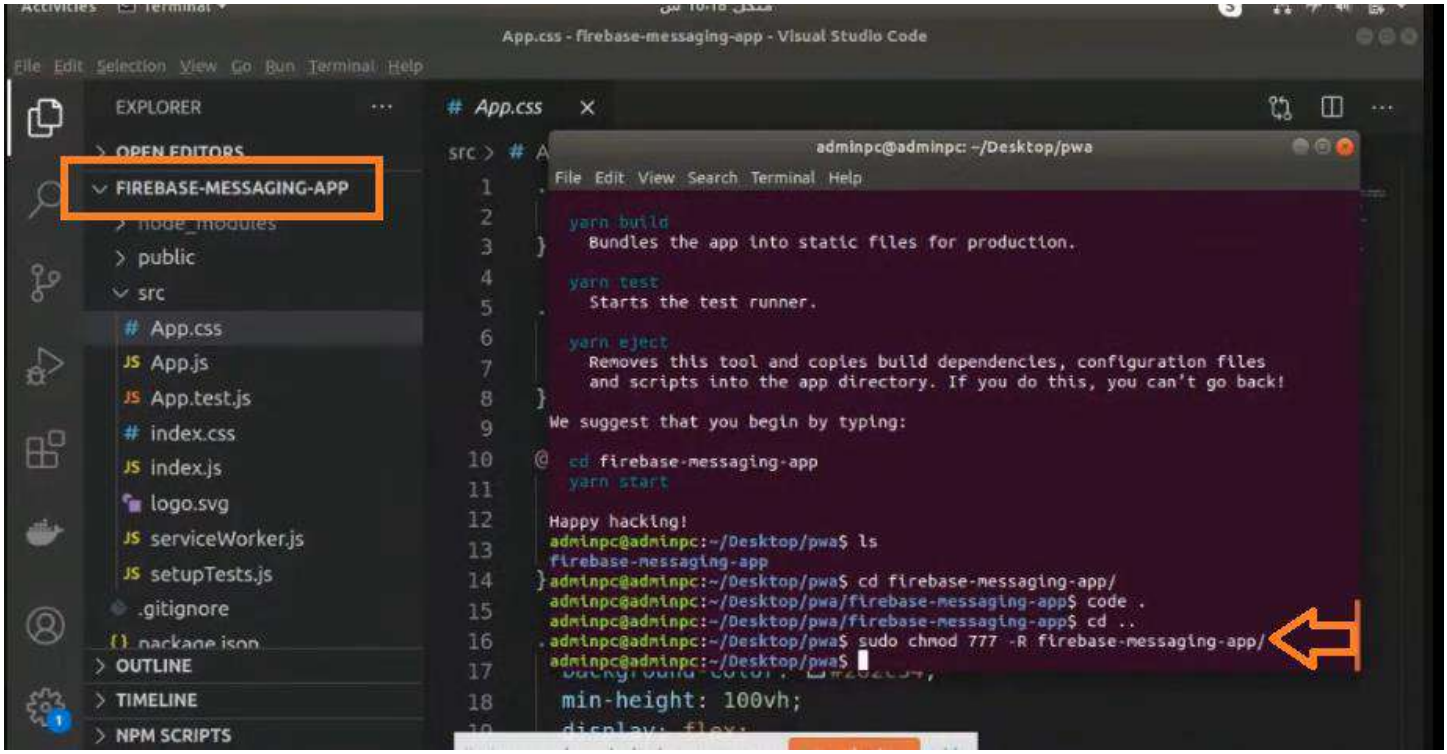




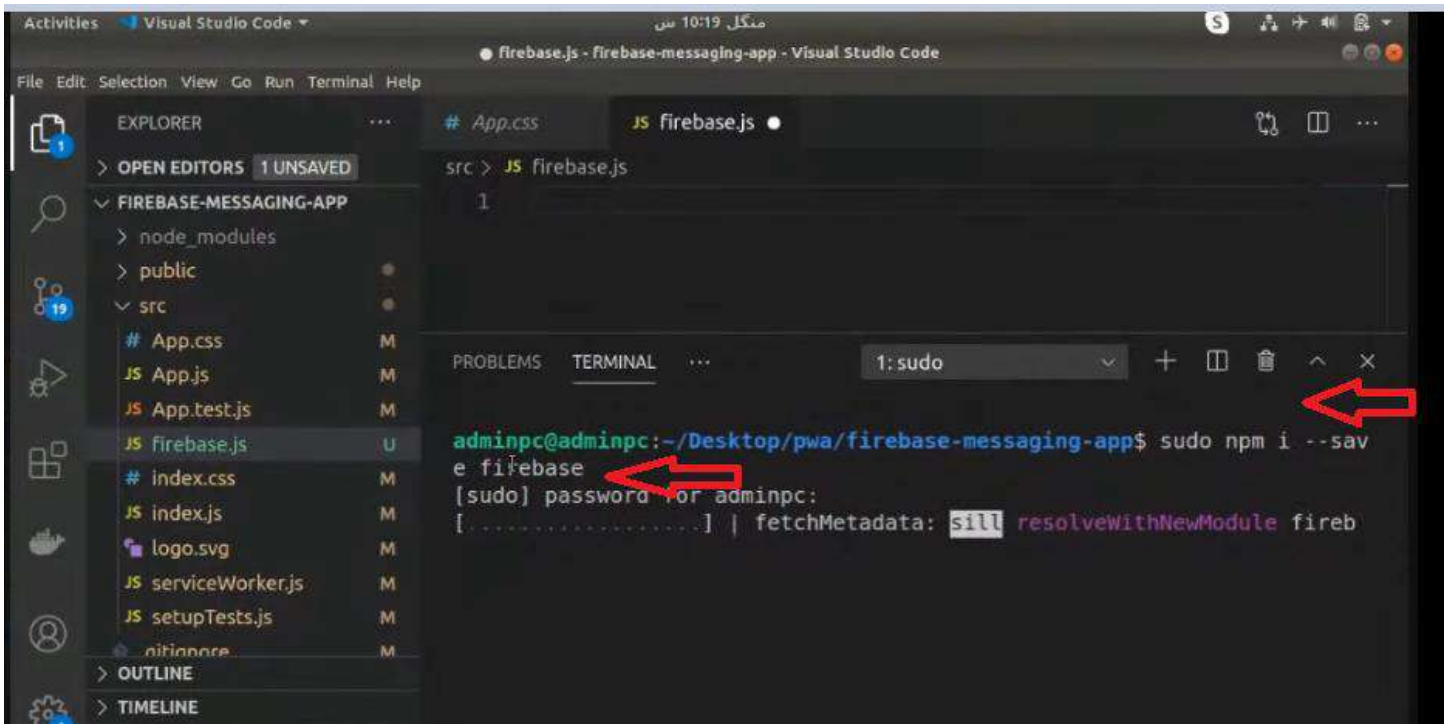
➢ When we register it we will get the configuration notification and just because of this configuration we are using this firebase



➢ go to the project and create a file firebasebase.js and provide permission to firebase



➢ run this **command sudo npm I--save firebase** base to install firebase library and install in it



- import firebase
- config it as you have created your project from console log
- then initialize this app and pass config in it
- export firebase

firebasebase.js

```
import firebase from 'firebase';

const config ={

  apiKey: "AIzaSyA6SV0uYKsl873p2T6M8ZpnH-GdkPAQlUo",
  authDomain: "fir-messaging-36757.firebaseapp.com",
  databaseURL: "https://fir-messaging-36757.firebaseio.cc
  projectId: "fir-messaging-36757",
  storageBucket: "fir-messaging-36757.appspot.com",
  messagingSenderId: "448424040321",
  appId: "1:448424040321:web:30741d3c92443961c83a80"

}

firebase.initiaizeApp(config)

export default firebase;
```

- go to the App.js
- import firebase in it as shown
- create a const messaging variable and call it from firebase

- after that just get permission that will return the promise
- the token will let us receive the notification

App.js

```
import React from 'react';
import './App.css';
import firebase from "./firebase";
function App() {
const messaging=firebase.messaging( );
messaging.requestPermission( ).then ( ) => {
   return message. getToken( )
   }).then(token) => {
    console.log('token,'token);
  })

<div className="App">
<Hello>

</div>
);
```

- create another file of firebase.messaging-sw.js
- add script from firebase sdk cdn as

  - **<script src="/__/firebase/7.22.0/firebase-app.js"></script>('https://www.gsatatic.com/firebasejs/7.17.1/firebase.app.js')**
  - **<script src="/__/firebase/7.22.0/firebase-app.js"></script>('https://www.gsatatic.com/firebasejs/7.17.1/firebase.messaging.js')**

This script contains all data coding of firebase app and messaging respectively so we are not in need to write all coding

- Initialize firebase app config it as you have created your project from console log
- After that call firebase.messaging

firebase.messaging-sw.js

import Script from firebase

<script src="/__/firebase/7.22.0/firebase-app.js"></script>('https://www.gsatatic.com/firebasejs/7.17.1/firebase.app.js')

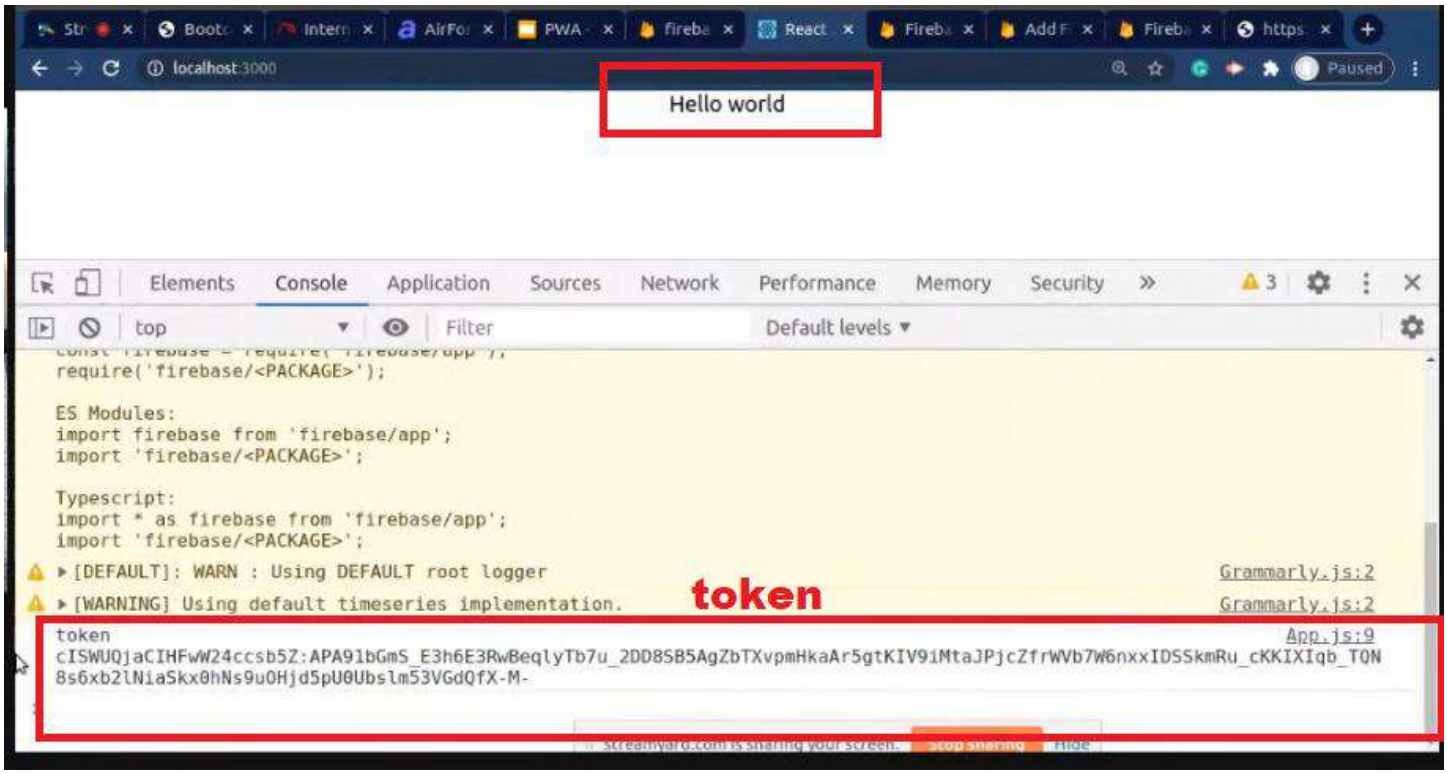
<script src="/__/firebase/7.22.0/firebase-app.js"></script>('https://www.gsatatic.com/firebasejs/7.17.1/firebase.messaging.js')

firebase.initializeApp({

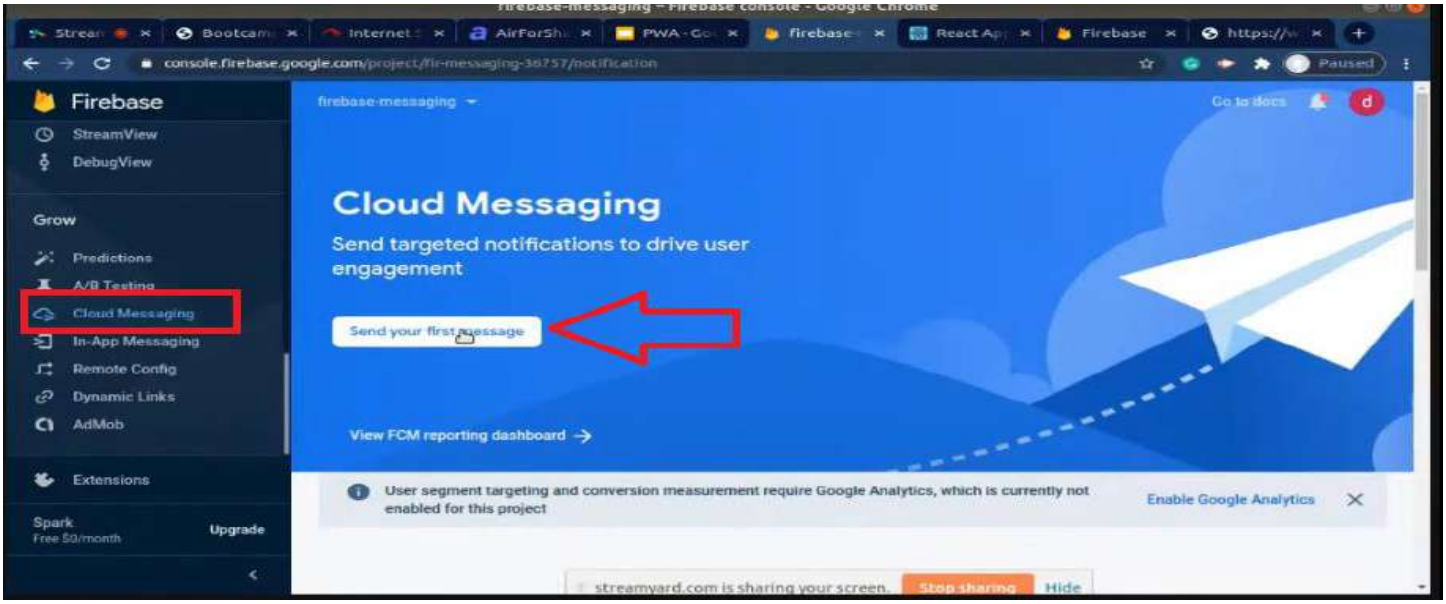
```
apiKey: "AIzaSyA6SV0uYKsl873p2T6M8ZpnH-GdkPAQlUo",
authDomain: "fir-messaging-36757.firebaseapp.com",
databaseURL: "https://fir-messaging-36757.firebaseio.cc
projectId: "fir-messaging-36757",
storageBucket: "fir-messaging-36757.appspot.com",
messagingSenderId: "448424040321",
appId: "1:448424040321:web:30741d3c92443961c83a80"
```

firebase.messaging();

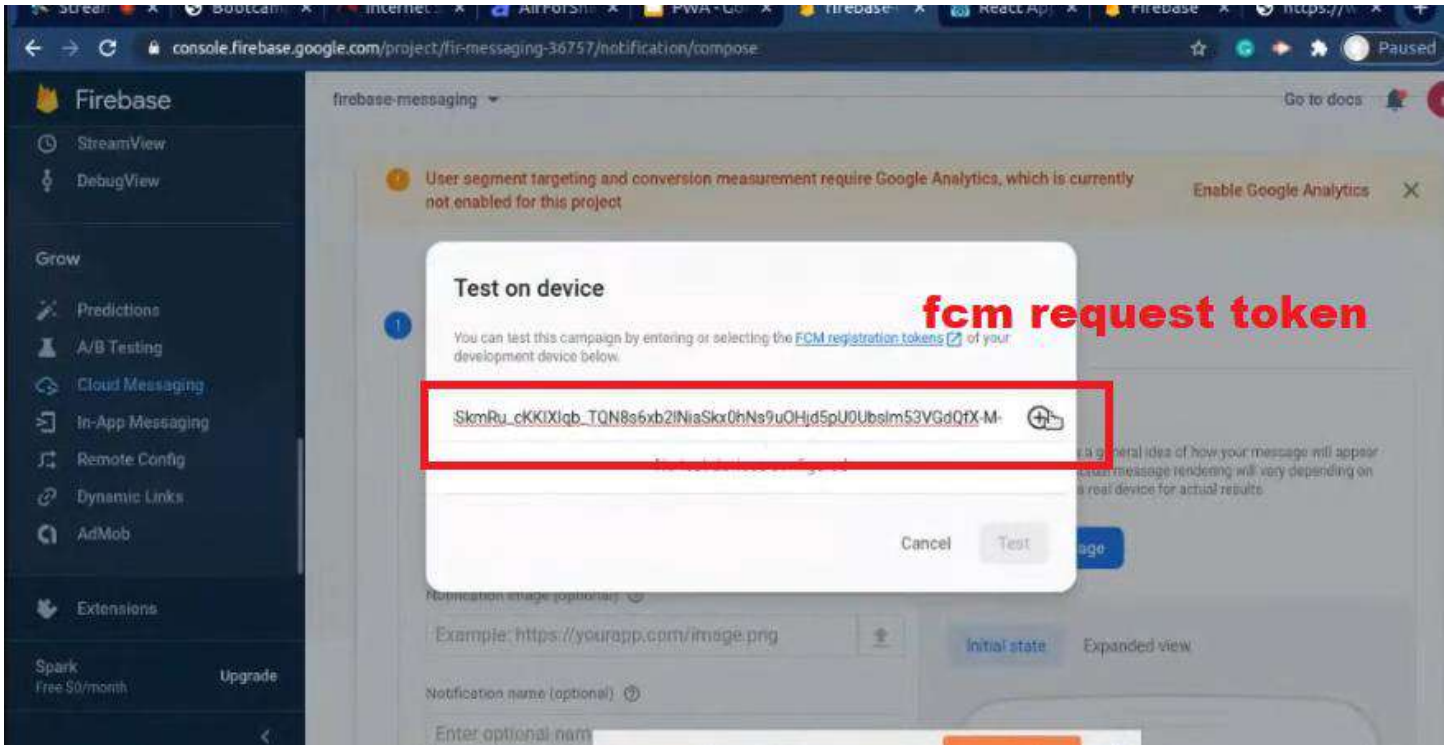
- Now go to the browser and check out the outcome

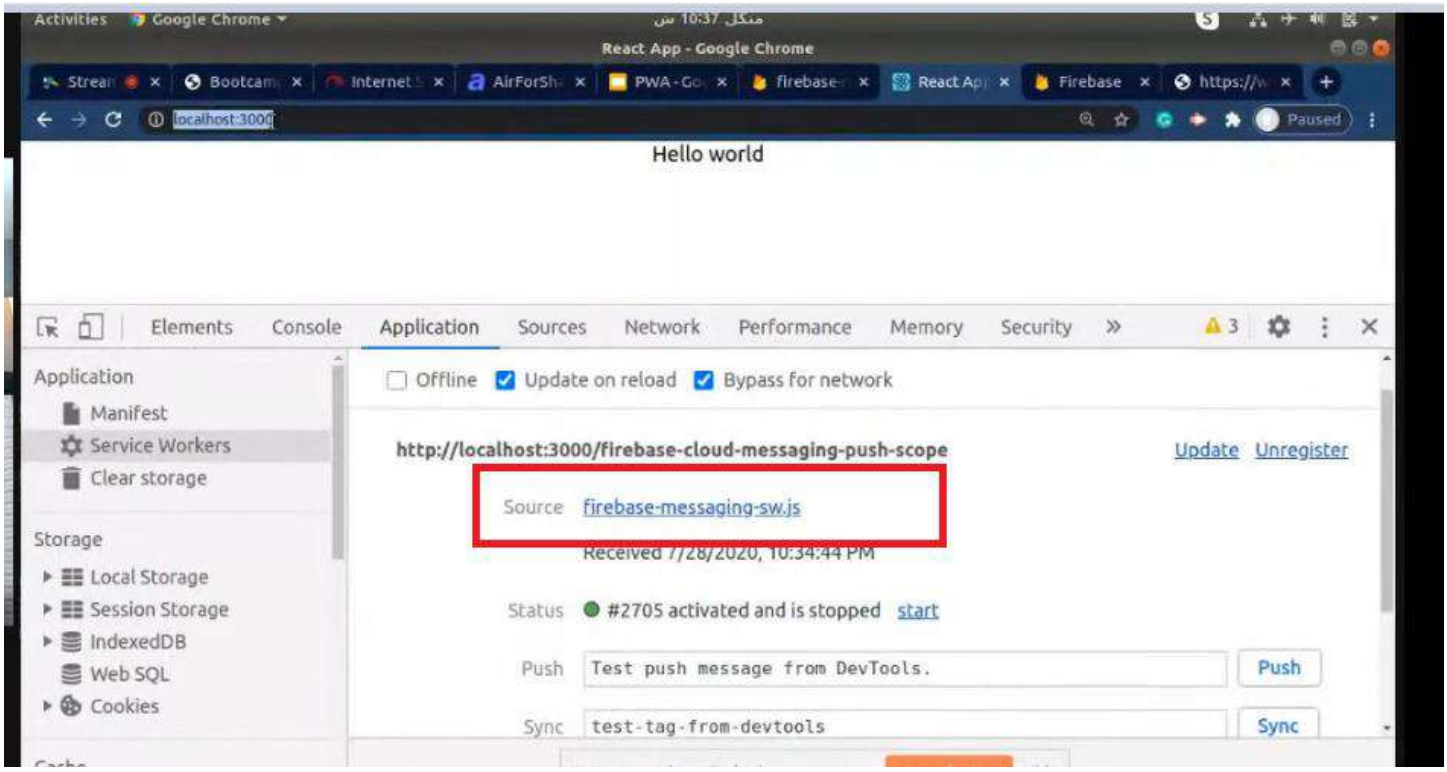


- copy this token this is app-specific for example if any user wants to send a chat message
- go to the cloud messaging click to send your first message

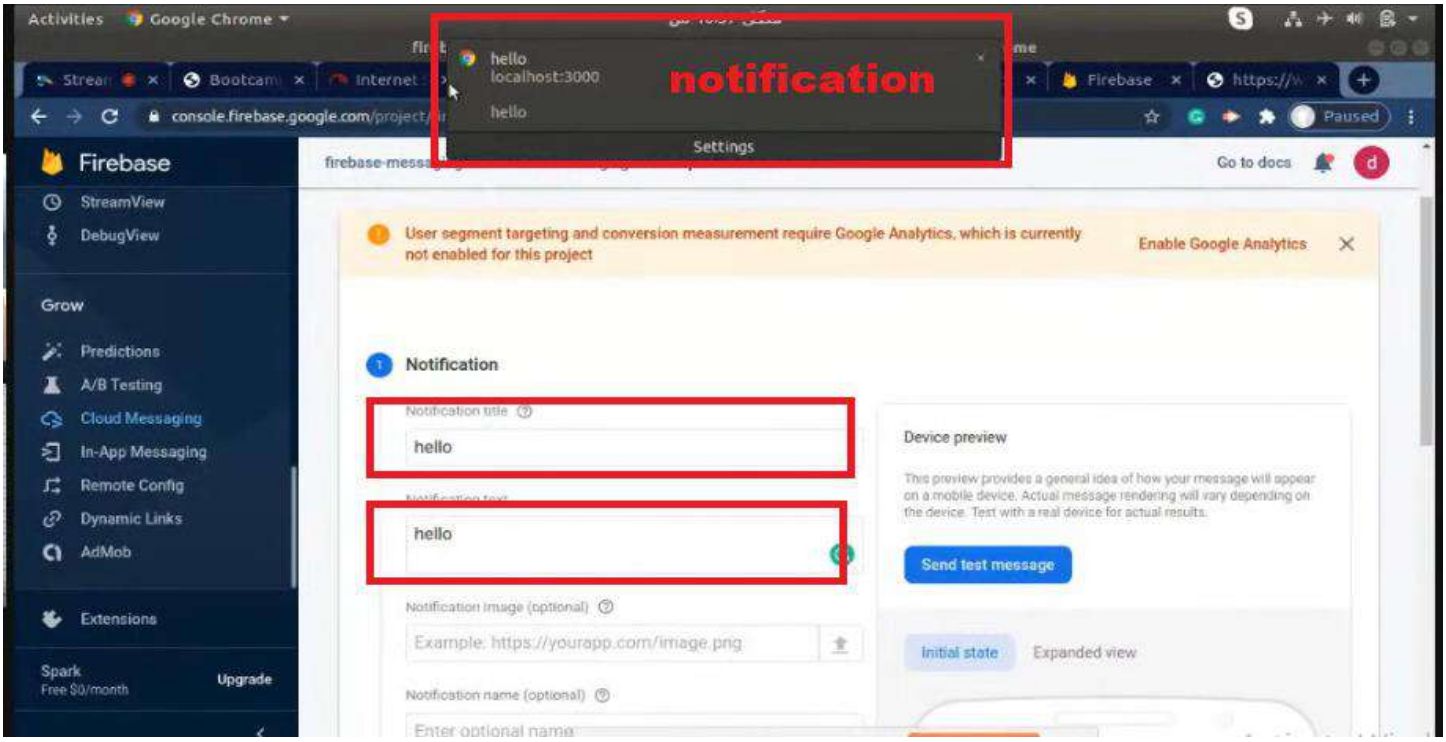


- Now write your first message and click t send
- And put the FCM request token
- Now checkout service worker (firebase.messaging-sw.js is registered)



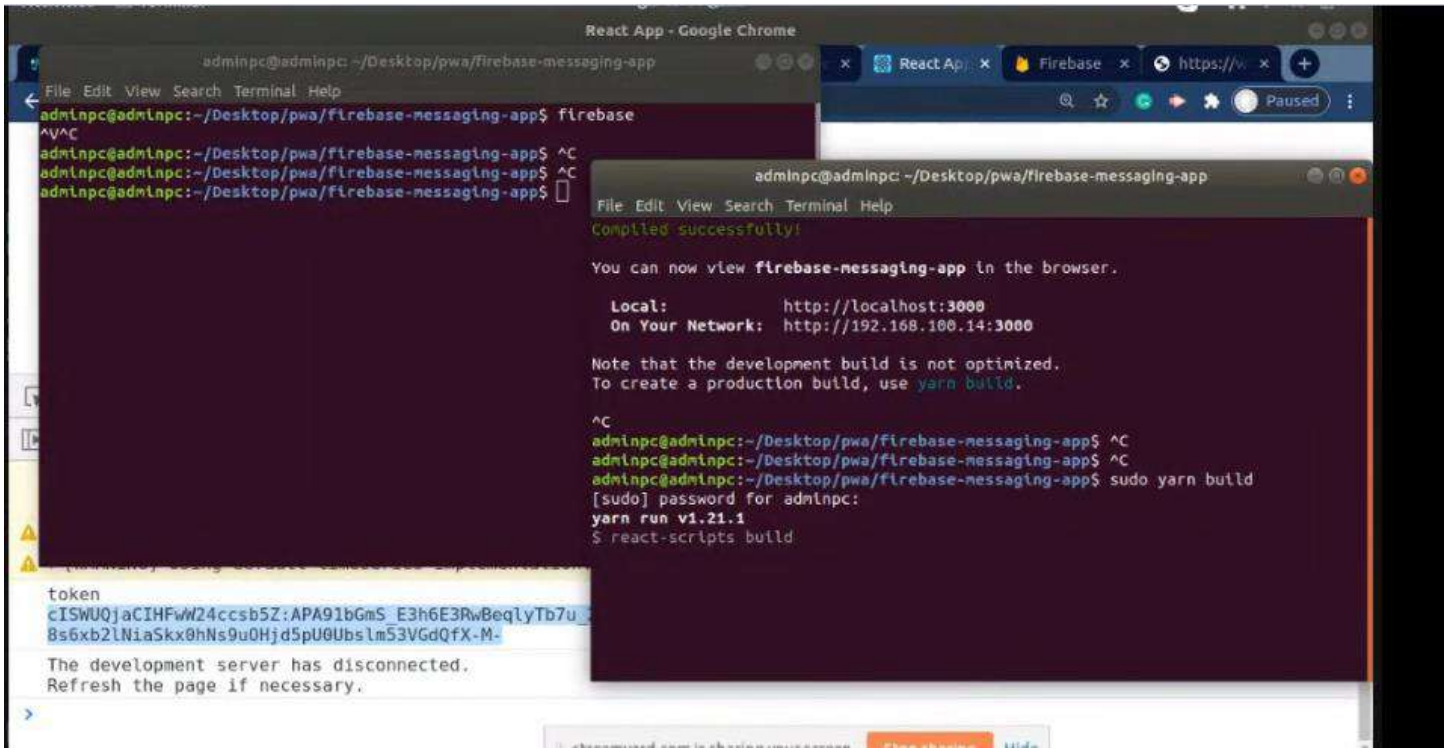


Click to test you will get a notification of message you have written “hello” as shown

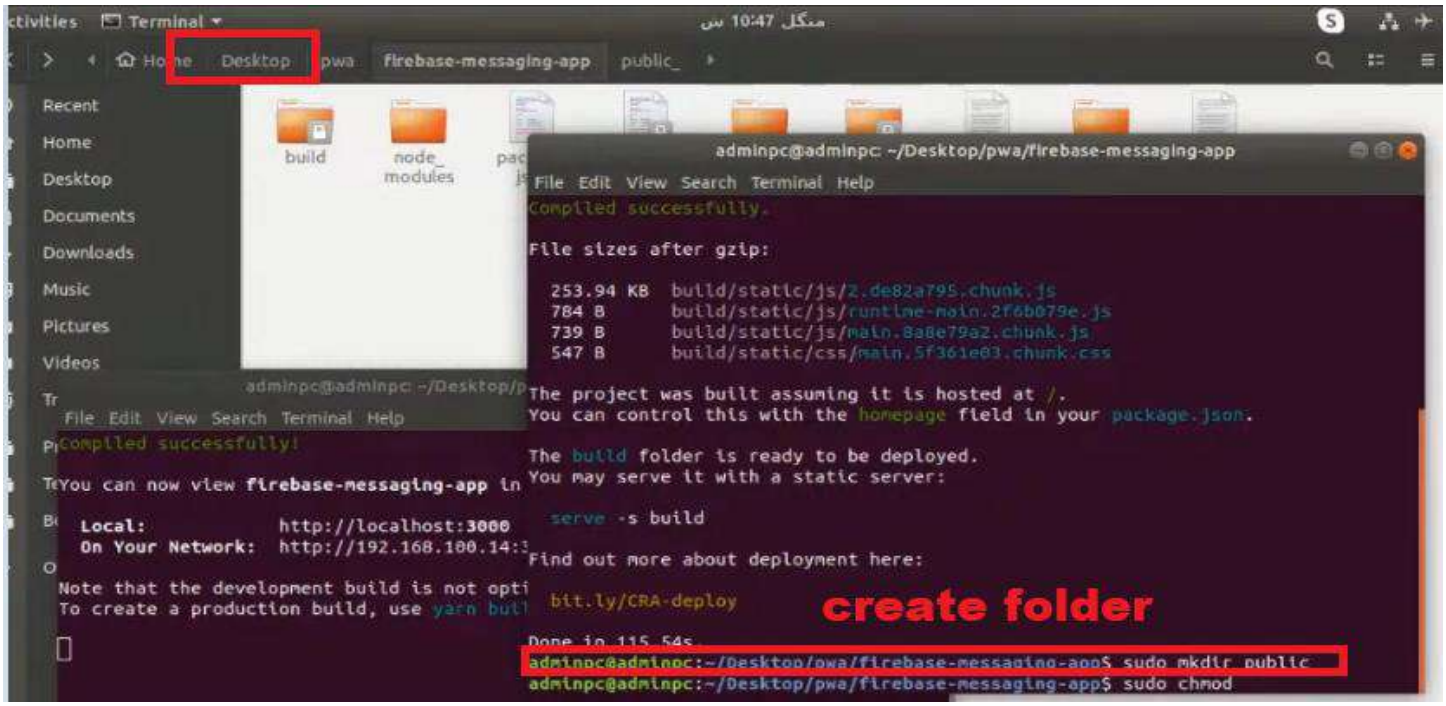


Get a notification on mobile

- build an application for the deployment



- go to the desktop create a folder in the command
- give permissions

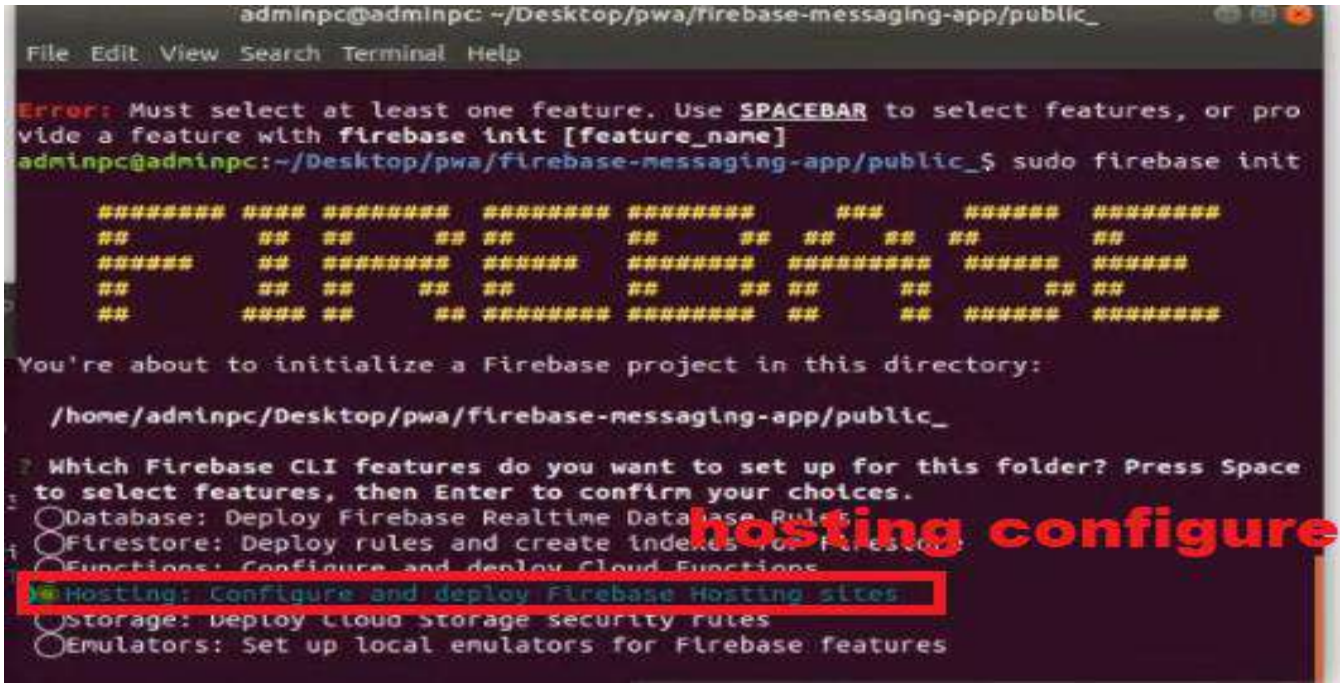


- copy the build into the public folder
- enter commands of firebase init and run this command

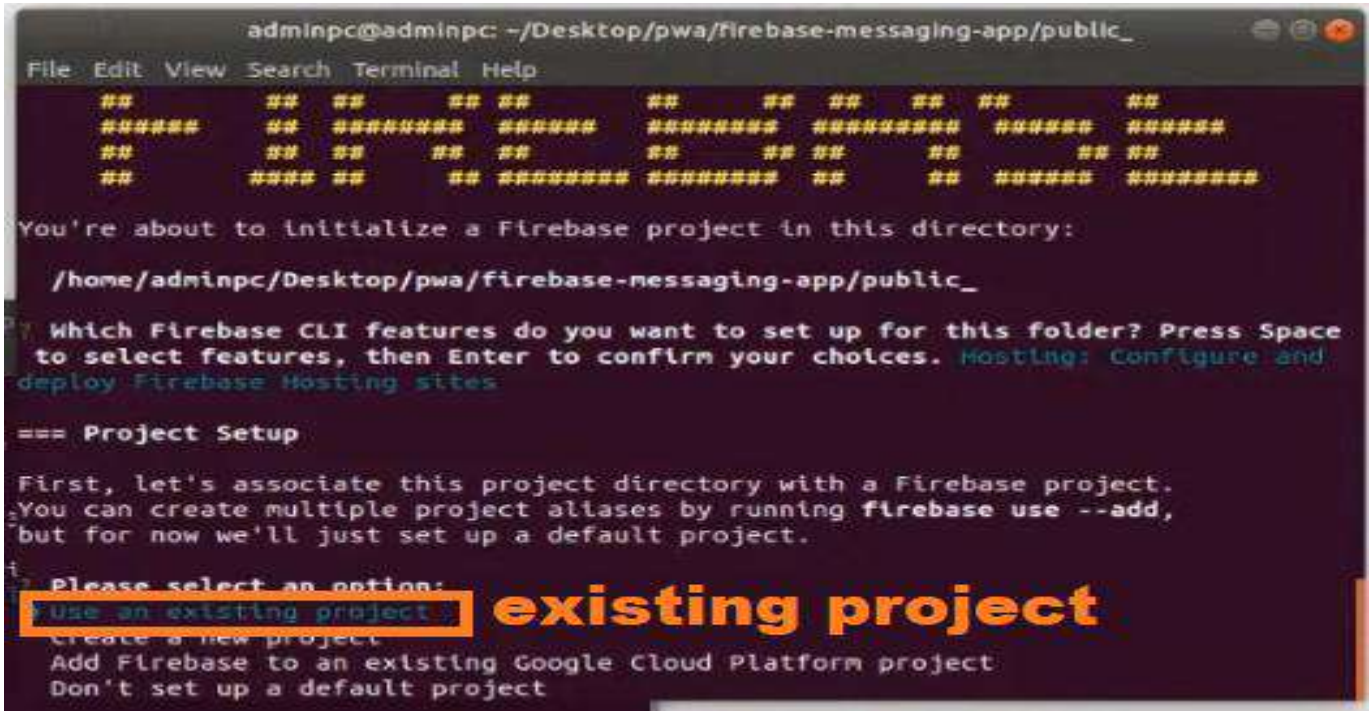
```
adminpc@adminpc: ~/Desktop/pwa/firebase-messaging-app/public_
File  Edit  View  Search  Terminal  Help
  784 B       build/static/js/runtime-main.2f6b079e.js
  739 B       build/static/js/main.8a8e79a2.chunk.js
  547 B       build/static/css/main.5f361e03.chunk.css

The project was built assuming it is hosted at /.
You can control this with the homepage field in your package.json.

The build folder is ready to be deployed.
You may serve it with a static server:

  serve -s build

Find out more about deployment here:

  bit.ly/CRA-deploy

Done in 115.54s.
adminpc@adminpc:~/Desktop/pwa/firebase-messaging-app$ sudo mkdir public_
adminpc@adminpc:~/Desktop/pwa/firebase-messaging-app$ sudo chmod 777 -R build/
adminpc@adminpc:~/Desktop/pwa/firebase-messaging-app$ sudo chmod 777 -R p
package.json        package-lock.json  public/            public_/
adminpc@adminpc:~/Desktop/pwa/firebase-messaging-app$ sudo chmod 777 -R public /
adminpc@adminpc:~/Desktop/pwa/firebase-messaging-app$ cd public_/
adminpc@adminpc:~/Desktop/pwa/firebase-messaging-app/public_$ firebase init
```

```
adminpc@adminpc: ~/Desktop/pwa/firebase-messaging-app/public_
File  Edit  View  Search  Terminal  Help
    at tryModuleLoad (internal/modules/cjs/loader.js:593:12)
    at Function.Module._load (internal/modules/cjs/loader.js:585:3)
adminpc@adminpc:~/Desktop/pwa/firebase-messaging-app/public_$ sudo firebase init

     ######## #### ########  ######## ########     ###     ######  ########
     ##        ##  ##     ## ##       ##     ##   ## ##   ##    ##  ##
     ######    ##  ########  ######   ########  #########  ######  ######
     ##        ##  ##    ##  ##       ##     ## ##     ##       ## ##
     ##       #### ##     ## ######## ########  ##     ##  ######  ########

You're about to initialize a Firebase project in this directory:

  /home/adminpc/Desktop/pwa/firebase-messaging-app/public_

? Which Firebase CLI features do you want to set up for this folder? Press Space
 to select features, then Enter to confirm your choices. (Press <space> to selec
t, <a> to toggle all, <i> to invert selection)

Error: Must select at least one feature. Use SPACEBAR to select features, or pro
vide a feature with firebase init [feature_name]
adminpc@adminpc:~/Desktop/pwa/firebase-messaging-app/public_$
```

- Now click to hosting configure by space bar and pressing enter



- Now after deploying firebase hosting configure click to an existing project



- Now choose your phone device to use an existing project

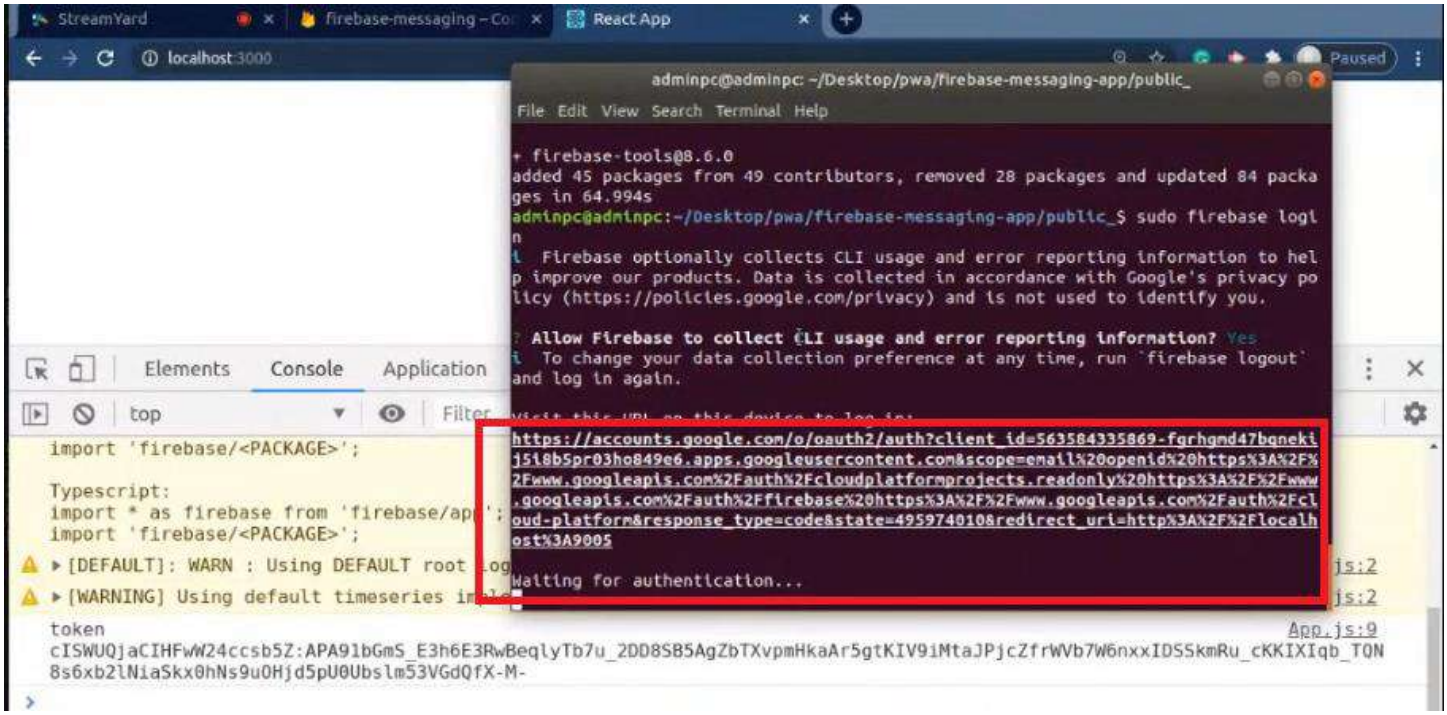
```
adminpc@adminpc: ~/Desktop/pwa/firebase-messaging-app/public_
File  Edit  View  Search  Terminal  Help
You're about to initialize a Firebase project in this directory:

  /home/adminpc/Desktop/pwa/firebase-messaging-app/public_

? Which Firebase CLI features do you want to set up for this folder? Press Space
 to select features, then Enter to confirm your choices. Hosting: Configure and
deploy Firebase Hosting sites

=== Project Setup

First, let's associate this project directory with a Firebase project.
You can create multiple project aliases by running firebase use --add,
but for now we'll just set up a default project.

? Please select an option: Use an existing project
? Select a default Firebase project for this directory:
  luminous-torch-4640 (Panacloud-Dev)
  parkingapp2-c8b26 (parkingapp2)
  parkingapplication-e7d1f (parkingapplication)
❯ phone-auth-b54e8 (phone-auth)
  saturn3-am-process (saturn3 AM Process)
  sbs-piaic (sbs-piaic)
  smoothflowbeta1 (smoothflow)
(Move up and down to reveal more choices)
```

select phone device

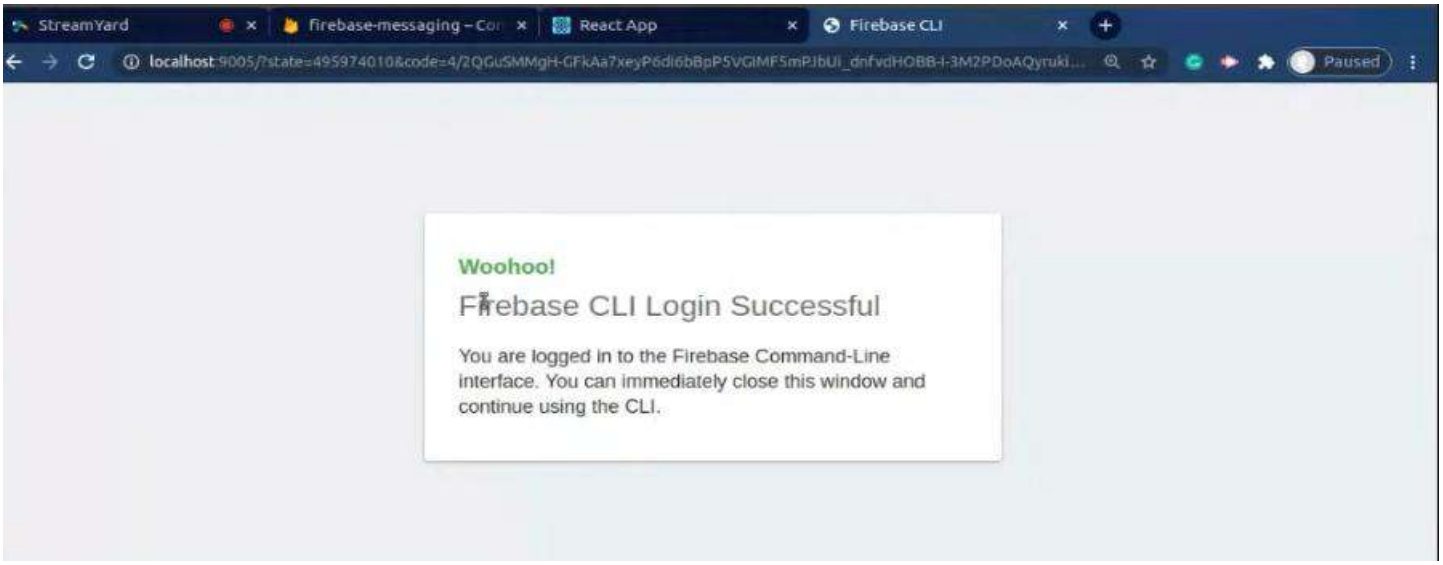
➢ now install this update and log onto another device

```
File  Edit  View  Search  Terminal  Help
? Please select an option: Use an existing project

Error: Failed to list Firebase projects. See firebase-debug.log for more info.

     Update available 7.11.0 → 8.6.6
   Run npm i -g firebase-tools to update

Having trouble? Try again or contact support with contents of firebase-debug.log
adminpc@adminpc:~/Desktop/pwa/firebase-messaging-app/public_$ sudo firebase logi

lreadv logged in as muhammadmohsinkhalid@gmail.com
dminpc@adminpc:~/Desktop/pwa/firebase-messaging-app/public_$ sudo firebase logo
t
⚠  Invalid refresh token, did not need to deauthorize
✔  Logged out from muhammadmohsinkhalid@gmail.com
dminpc@adminpc:~/Desktop/pwa/firebase-messaging-app/public_$ sudo npm i -g fire
ase-tools
```

➢ By using the firebase login command we can use it on another account



Now choose any account and login



- Successfully logged in
- Enter the command of firebase deploy

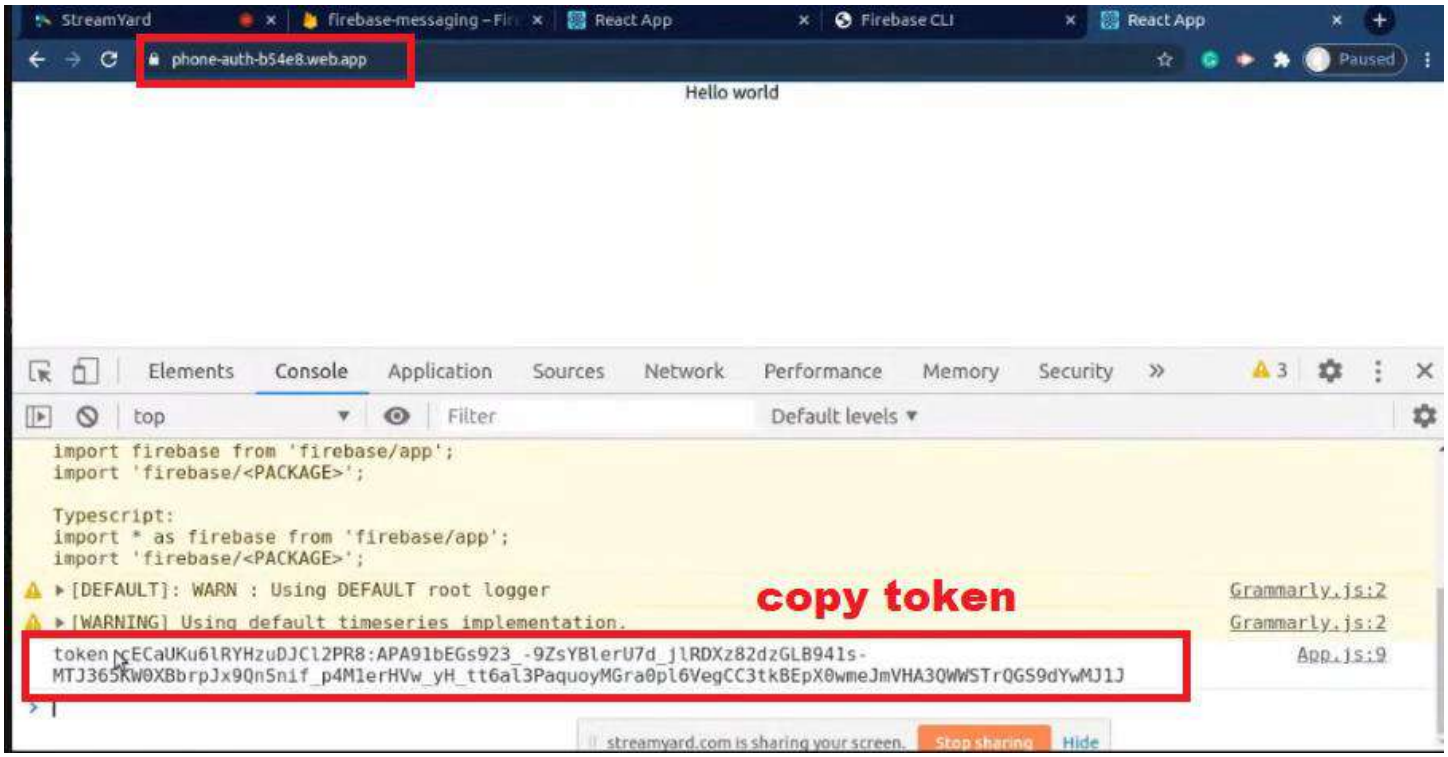
- In the public directory, we will run this command where we have to build a folder
- Now everything will be deployed

```
adminpc@adminpc: ~/Desktop/pwa/firebase-messaging-app/public_
File  Edit  View  Search  Terminal  Help
First, let's associate this project directory with a Firebase project.
You can create multiple project aliases by running firebase use --add,
but for now we'll just set up a default project.

i  .firebaserc already has a default project, using phone-auth-b54e8.

=== Hosting Setup

Your public directory is the folder (relative to your project directory) that
will contain Hosting assets to be uploaded with firebase deploy. If you
have a build process for your assets, use your build's output directory.

? What do you want to use as your public directory? build
? Configure as a single-page app (rewrite all urls to /index.html)? Yes
? File build/index.html already exists. Overwrite? No
i  Skipping write of build/index.html

i  Writing configuration info to firebase.json...
i  Writing project information to .firebaserc...

✔  Firebase initialization complete!
adminpc@adminpc:~/Desktop/pwa/firebase-messaging-app/public_$ sudo firebase depl
oy
```

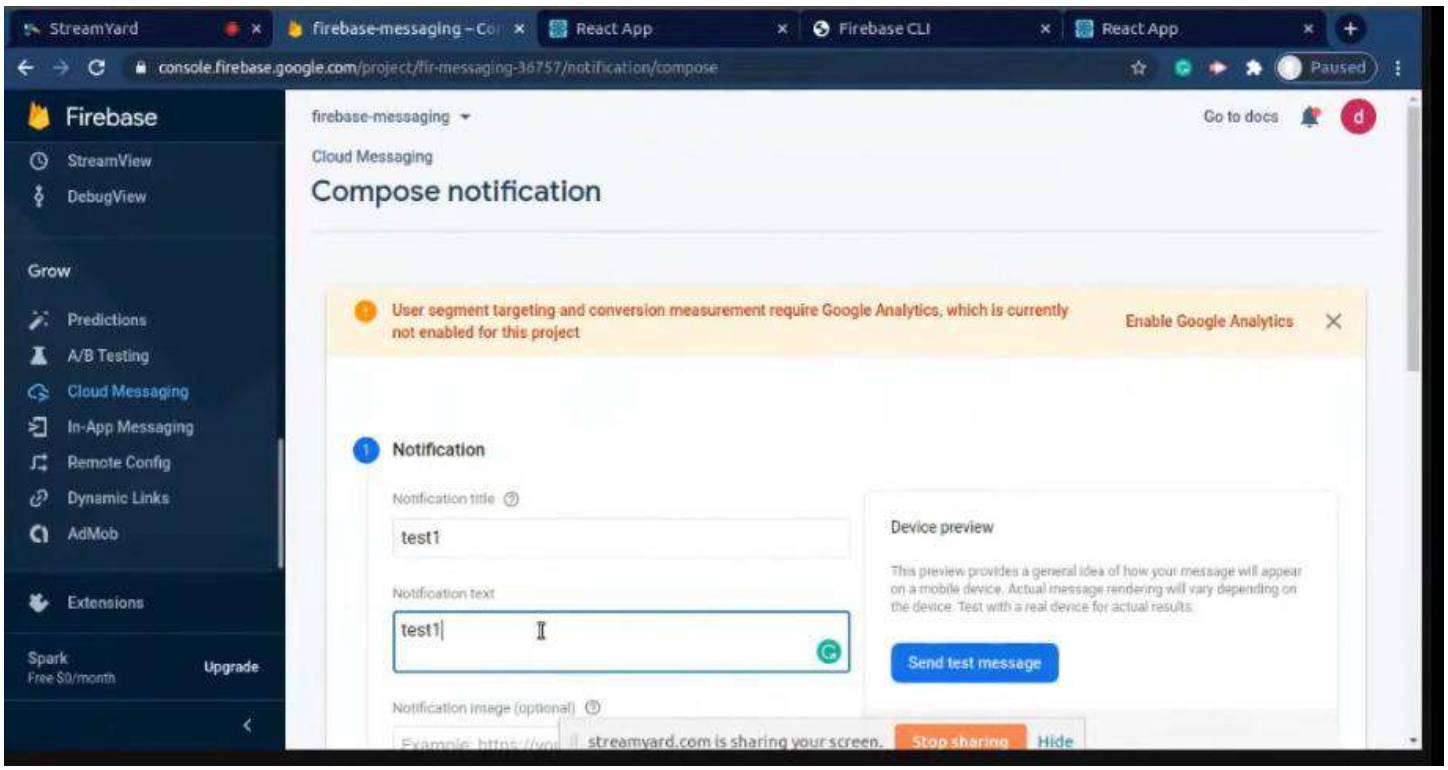
Select your phone project and it will provide a URL

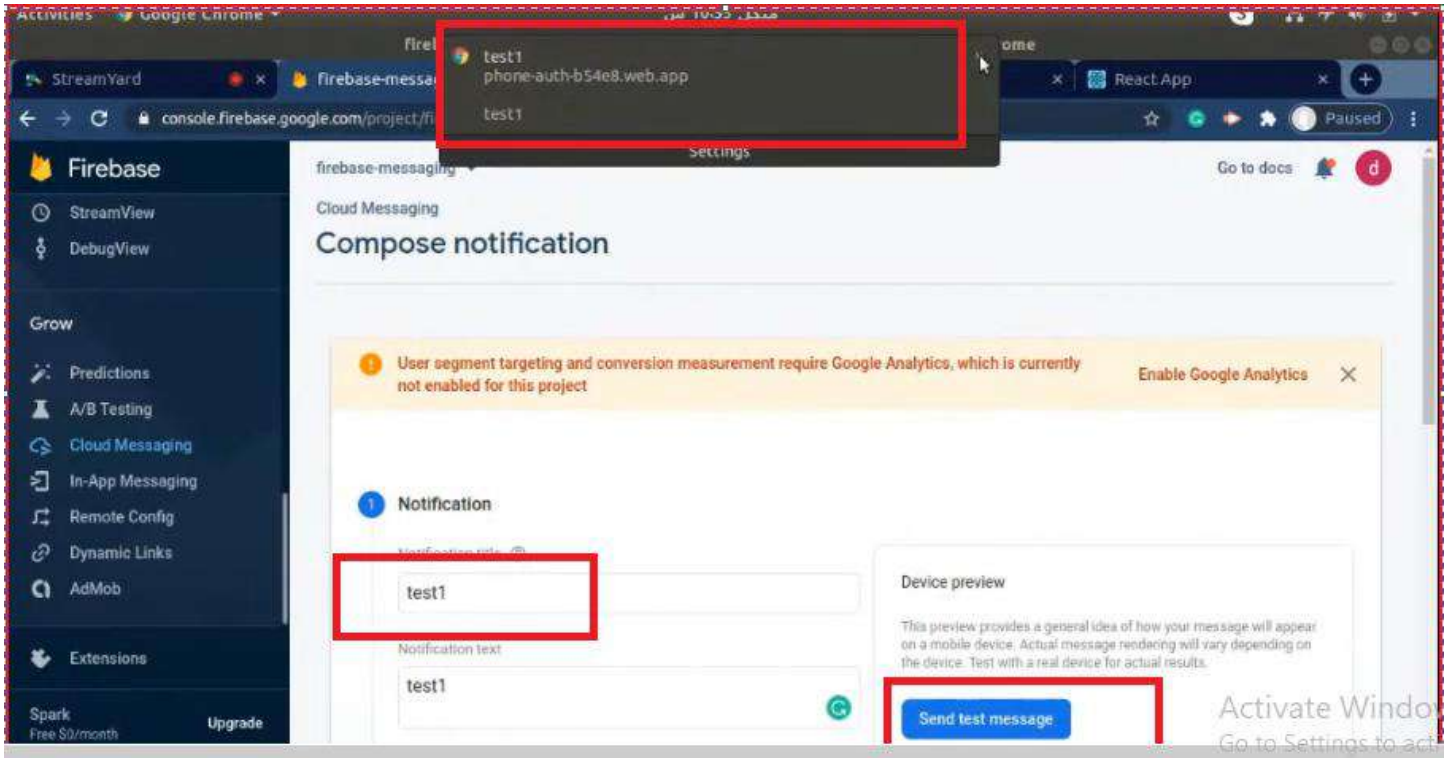
```
adminpc@adminpc: ~/Desktop/pwa/firebase-messaging-app/public_
File  Edit  View  Search  Terminal  Help
i  Writing configuration info to firebase.json...
i  Writing project information to .firebaserc...

✔  Firebase initialization complete!
adminpc@adminpc:~/Desktop/pwa/firebase-messaging-app/public_$ sudo firebase depl
oy

==  Deploying to 'phone-auth-b54e8'...

i  deploying hosting
i  hosting[phone-auth-b54e8]: beginning deploy...
i  hosting[phone-auth-b54e8]: found 19 files in build
✔  hosting[phone-auth-b54e8]: file upload complete
i  hosting[phone-auth-b54e8]: finalizing version...
✔  hosting[phone-auth-b54e8]: version finalized
i  hosting[phone-auth-b54e8]: releasing new version...
✔  hosting[phone-auth-b54e8]: release complete

✔  Deploy complete!

Project Console: https://console.firebase.google.com/project/phone-auth-b54e8/ov
erview
Hosting URL: https://phone-auth-b54e8.web.app
adminpc@adminpc:~/Desktop/pwa/firebase-messaging-app/public_$
```

Now paste it into the browser you will get a notification, allow it and you will get a token for your device



- Now compose a notification and click to send message and
- Enter URL copied from token and click to test
- You will get a notification





➢ By copying the URL on your mobile you can get this message on your mobile

Session 15: Test_React_Components_in_TypeScript

Testing

What is testing?

- The **testing** is important since it discovers defects/bugs before the delivery to the client,
- Testing is a process of executing a program with the aim of finding error. To make our software perform well it should be error free.If testing is done successfully it will remove all the errors from the software.

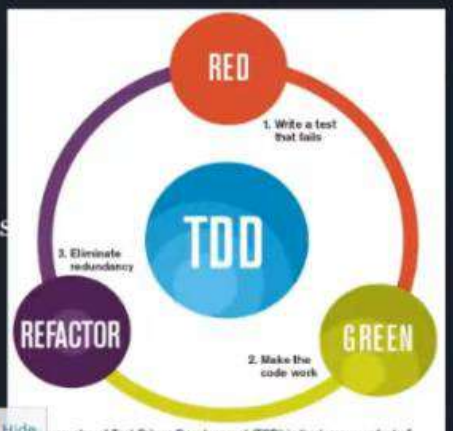
```
PASS  src/App.test.js
PASS  src/List/__tests__/List.test.js

Test Suites: 2 passed, 2 total
Tests:       5 passed, 5 total
Snapshots:   1 passed, 1 total
Time:        0.539s, estimated 1s
Ran all test suites.

Watch Usage: Press w to show more.
```

App.test.js

```
import React from 'react'
import renderer from 'react-test-renderer'
import { shallow, mount } from 'enzyme'
import 'jest-enzyme'
import 'jest-styled-components'

import App from './App.js'

describe('Component: App', () => {
  it('should call props.onChange when button is clicked', () => {
    const component = shallow(<App />)
    expect(component.find('h2').text()).toContain('Test Title')
  })
})
```

Test Driven Development:

- **Test Driven Development** is a software development process that follows these steps:
  - Write a failing test for one requirement.
  - Implement just enough code to make the test pas
  - Refactor with confidence (if it's needed).



e mantra of Test-Driven Development (TDD) is "red, green, refactor."

streamyard.com is sharing your screen. Stop sharing Hide

Types of Tesing

01 Unit Testing:

It focuses on smallest unit of software design. In this we test an individual unit or group of interrelated units.It is often done by programmer by using sample input and observing its corresponding outputs.

02 Unit Testing:

The objective is to take unit tested components and build a program structure that has been dictated by design.Integration testing is testing in which a group of components are combined to produce output.

streamyard.com is sharing your screen. Stop sharing Hide

03 Integration Testing:

1. The objective is to take unit tested components and build a program structure that has been dictated by design.
2. Integration testing is testing in which a group of components are combined to produce output.

04 Regression Testing:

1. Every time new module is added leads to changes in program. This type of testing make sure that whole component works properly even after adding components to the complete program.

05 Smoke Testing:

1. This test is done to make sure that software under testing is ready or stable for further testing.
2. It is called smoke test as testing initial pass is done to check if it did not catch the fire or smoked in the initial switch on.

streamyard.com is sharing your screen. Stop sharing Hide

06 Alpha Testing:

1. This is a type of validation testing.It is a type of *acceptance testing* which is done before the product is released to customers. It is typically done by QA people.

07 Beta Testing:

1. The beta test is conducted at one or more customer sites by the end-user of the software. This version is released for the limited number of users for testing in real time environment.

08 System Testing:

1. In this software is tested such that it works fine for different operating system.It is covered under the black box testing technique. In this we just focus on required input and output without focusing on internal working.
2. In this we have security testing, recovery testing , stress testing and performance testing

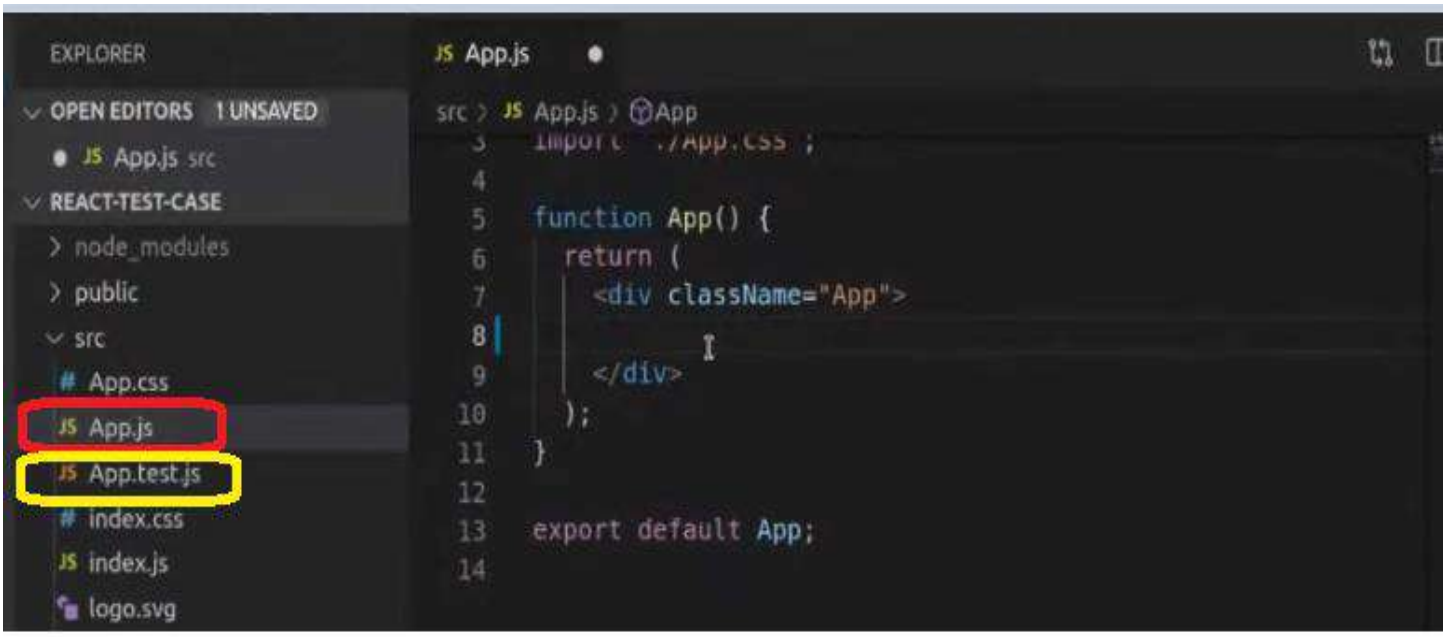
streamyard.com is sharing your screen. Stop sharing Hide

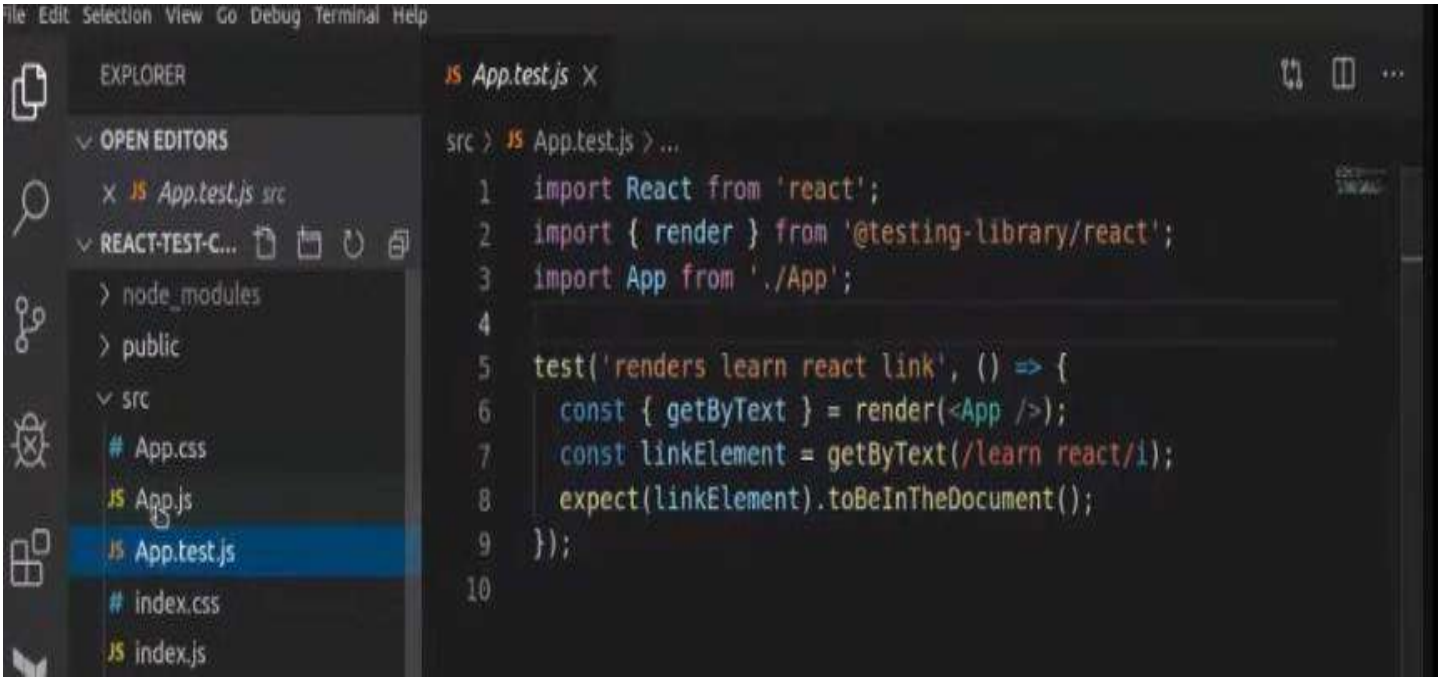
How to Write Test Cases

- react by default provide us test cases lets take into a look at how it does?
- go to terminal
- create project for testing
- write the following command whatever your folder name is choose that
- it will take a few minutes to complete. just let it go…
- after installation provides permissions if it asks

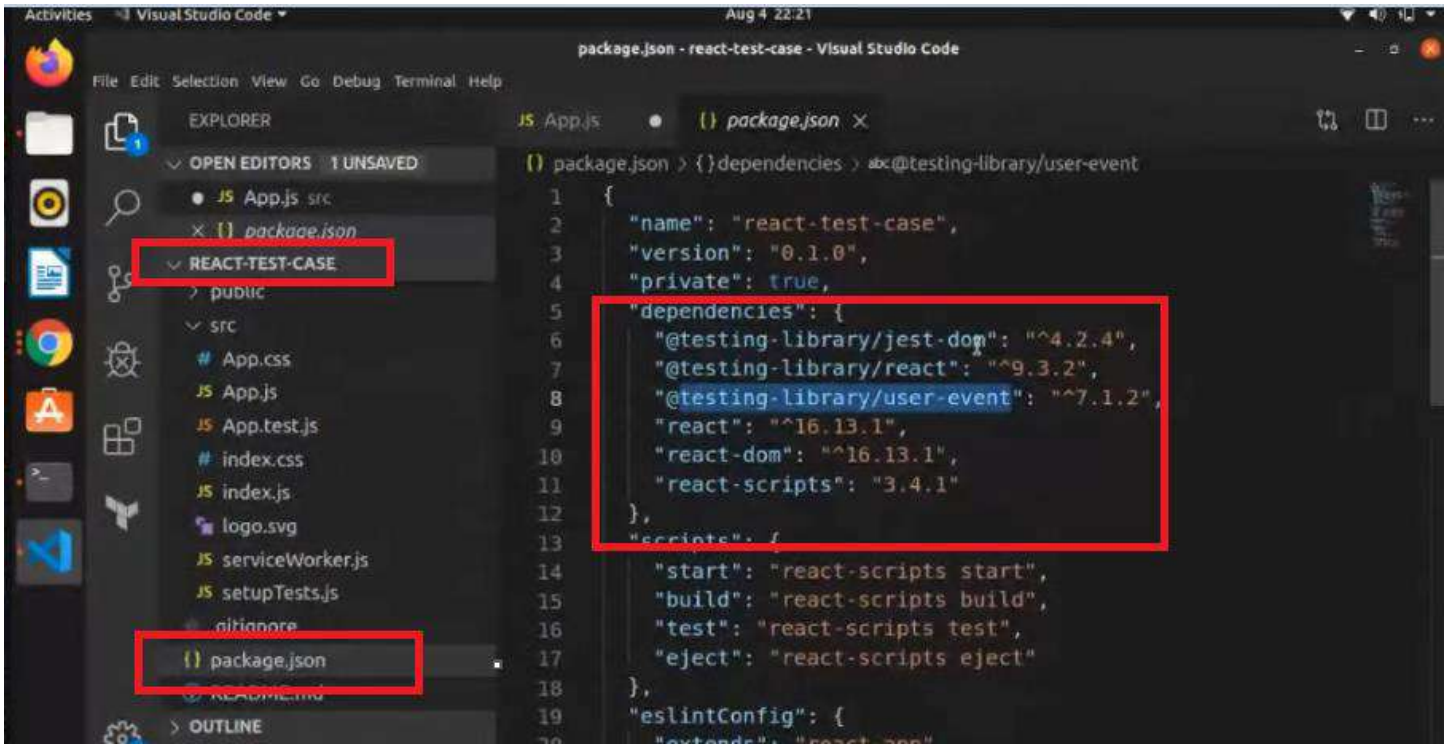
```
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC: ~/test-case
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~$ sudo mkdir test-case
[sudo] password for daniyal:
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~$ cd test-case/
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/test-case$ sudo npx create-react
-app react-test-case
[##................] | fetchMetadata: sill resolveWithNewModule buffer-from@0.1
```

- Whenever you look into your project for test-driven there is a by default created App.js file along with App.test.js file as shown

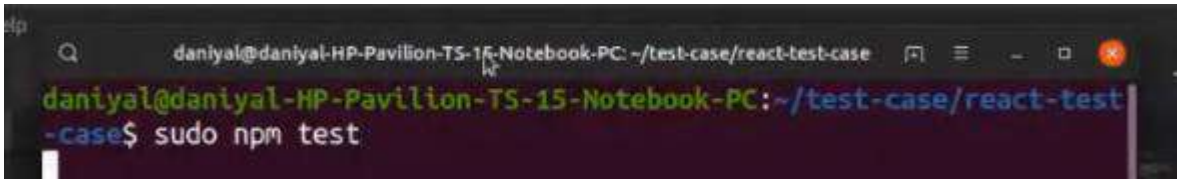




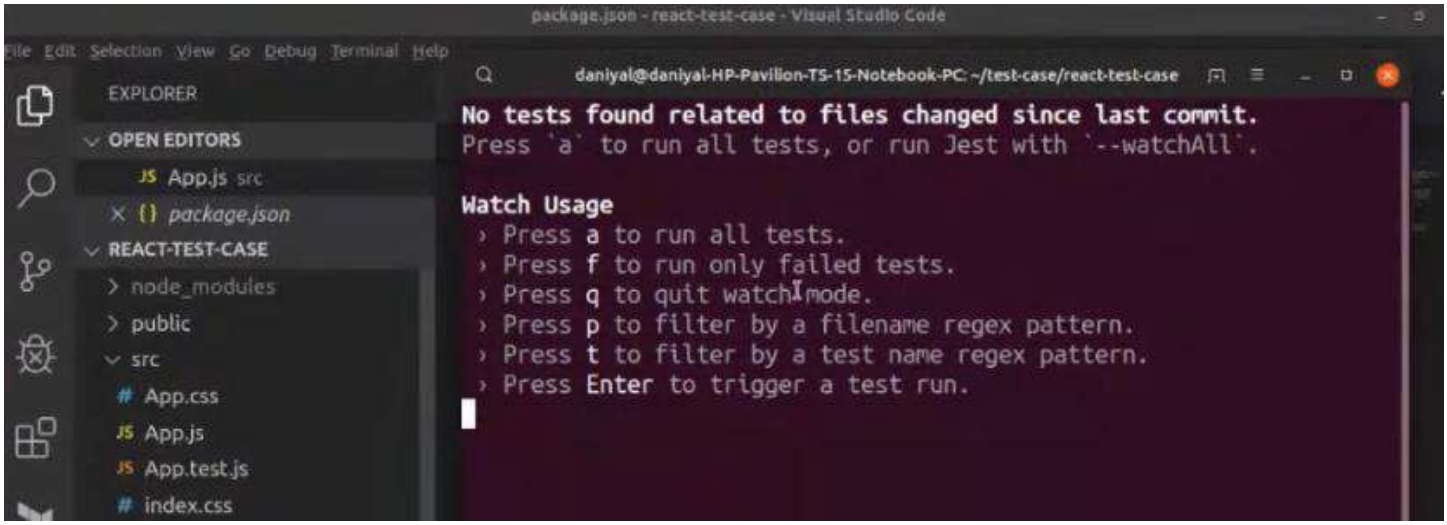
- The beauty of react is that "REACT-TEST-CASE" provide all these dependencies as pack.json is associated with Facebook so it provides all these dependencies by default whenever your run



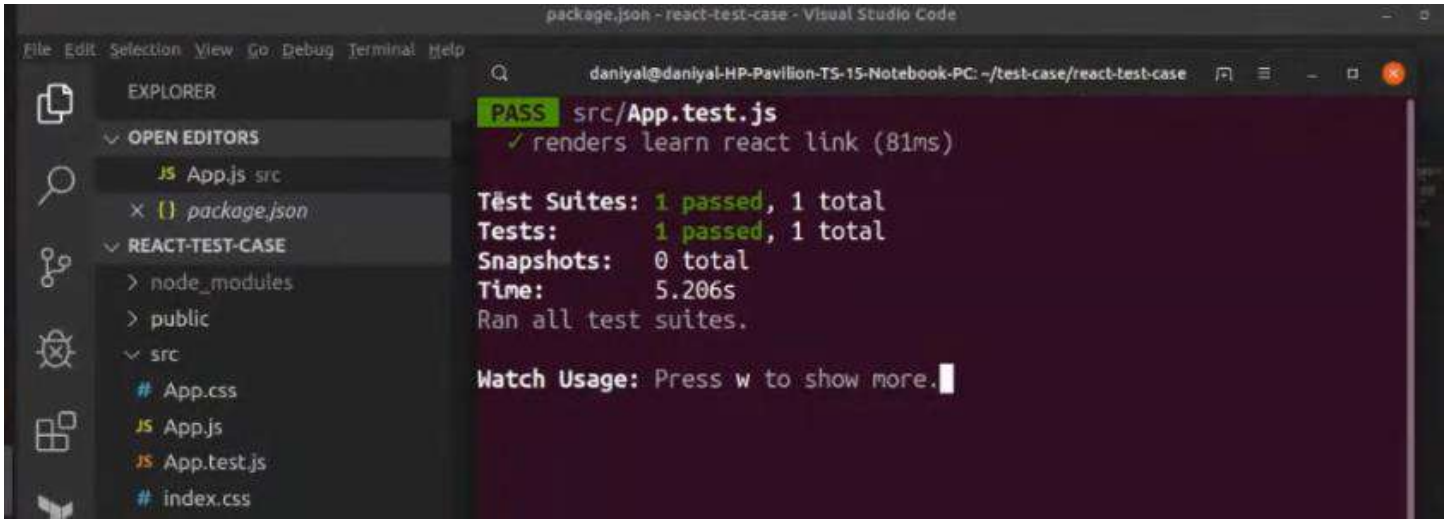
- go to terminal run the command test case with sudo npm test as



- Press a to run all test

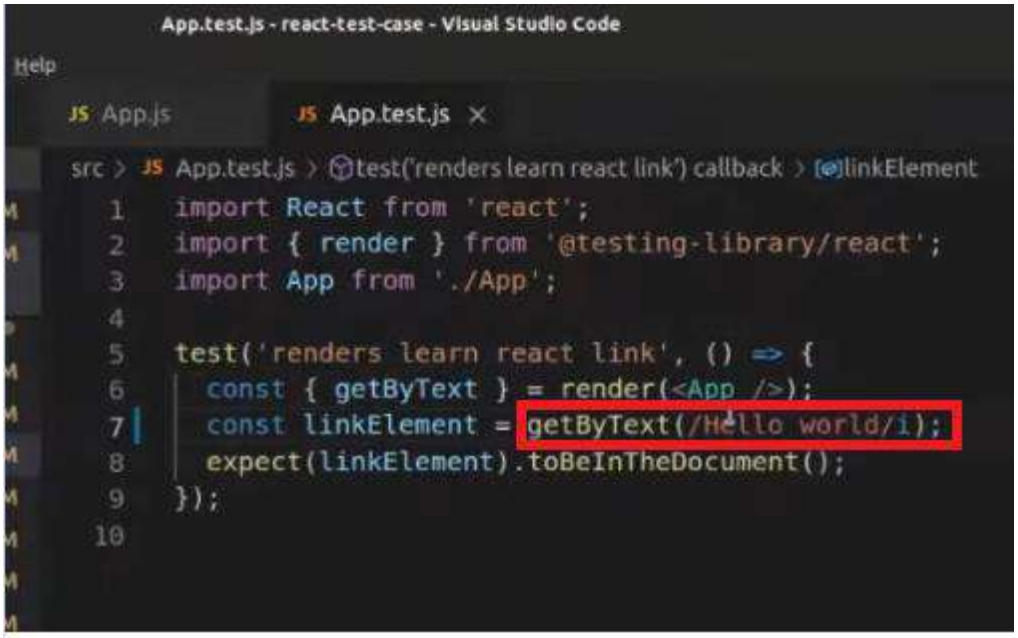


- this App.js file is auto-detected by the compiler as shown

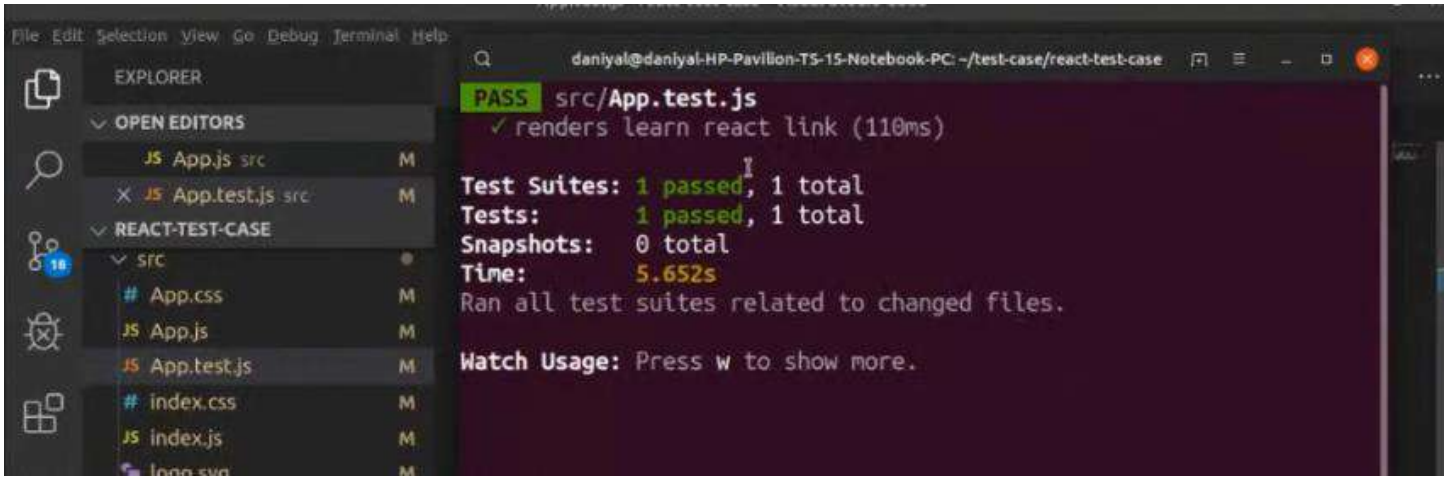


- Now write simple “Hello world “ in the App.js

- in App.test.js pass the argument of “ Hello world”



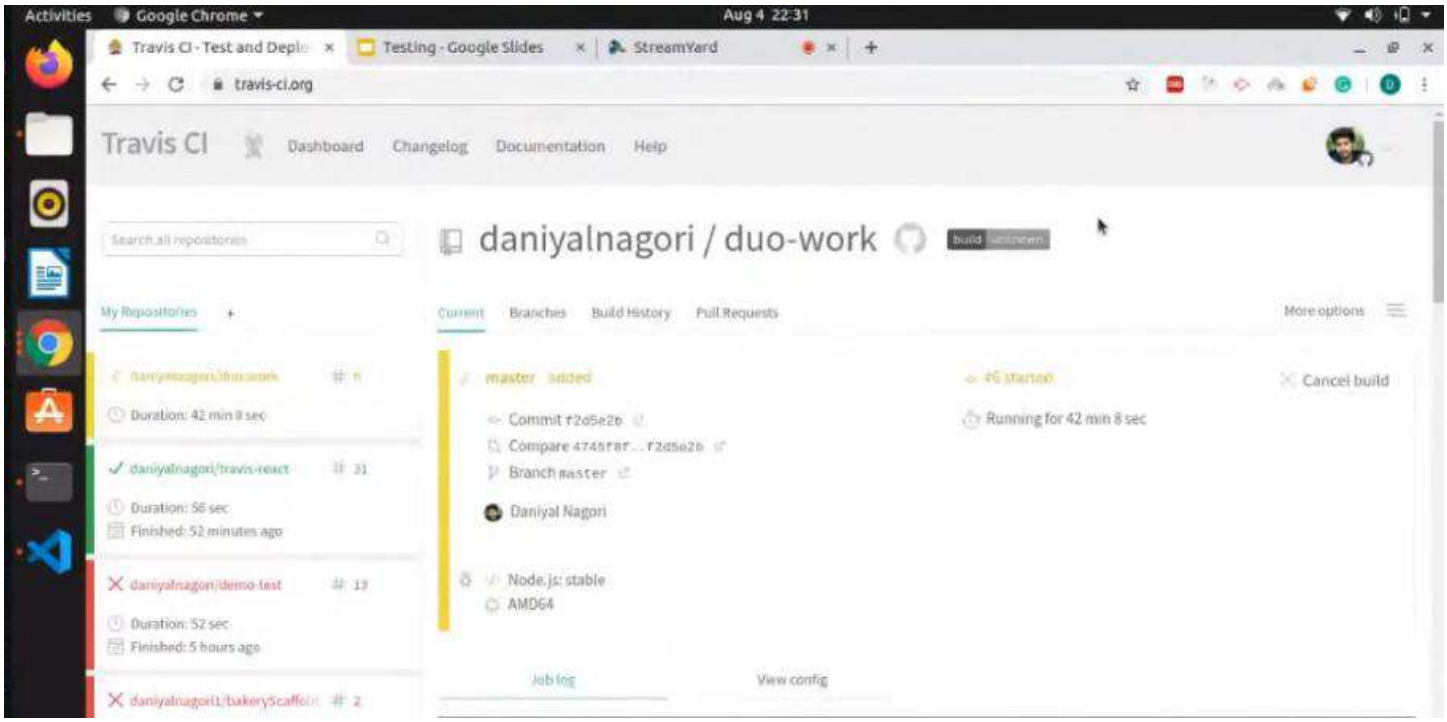
- run this in the command as shown



- So it is been passed
- In this way, you can use multiple test cases
- This is useful to release a bug-free software before it is published
- only the code that is bug-free and is passed in the test case then it will be deployed to any server like surge, firebase, serve to host

Travis CI

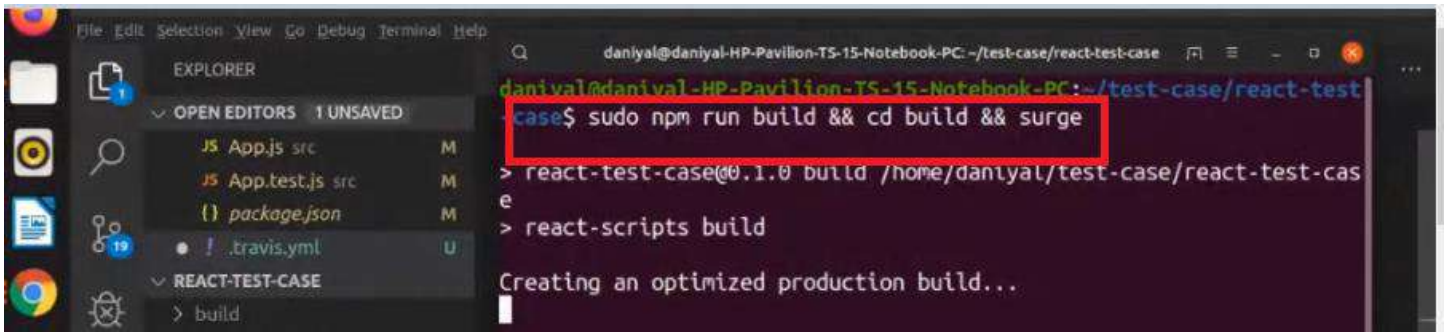
- it runs the test cases and implements these tests cases and once it's passed it will deploy it on firebase etc
- it is completely free and you sing up easily after making your account



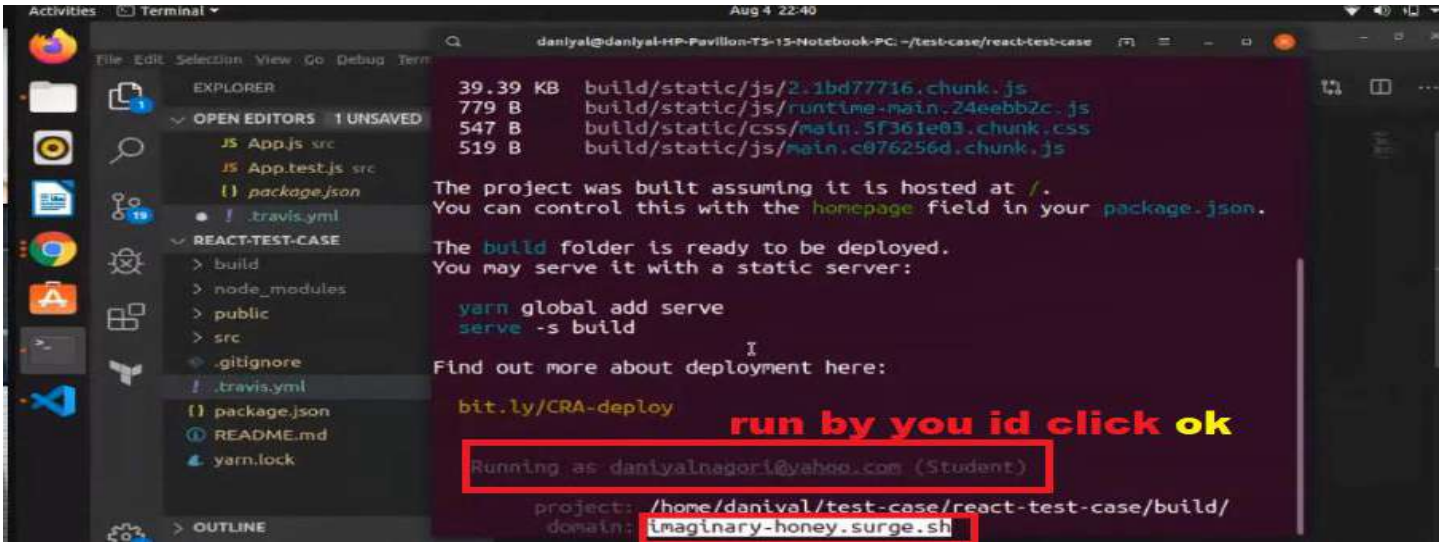
- go to github.com
- create a new repository
- initialize github on your project
- create a file .traves.yml

```
 // to run everything
language:nodejs
node_js:
//stable version
 -"stable"
// npm install, test is passed then nmp run command will built
// then it will be deployed on surge

script:
-npm install
-nmp test
-npm run build
//
deploy:
provider:surge
skip-clean:true
domain:
project:./build/
//branch we want to deploye
on:
```

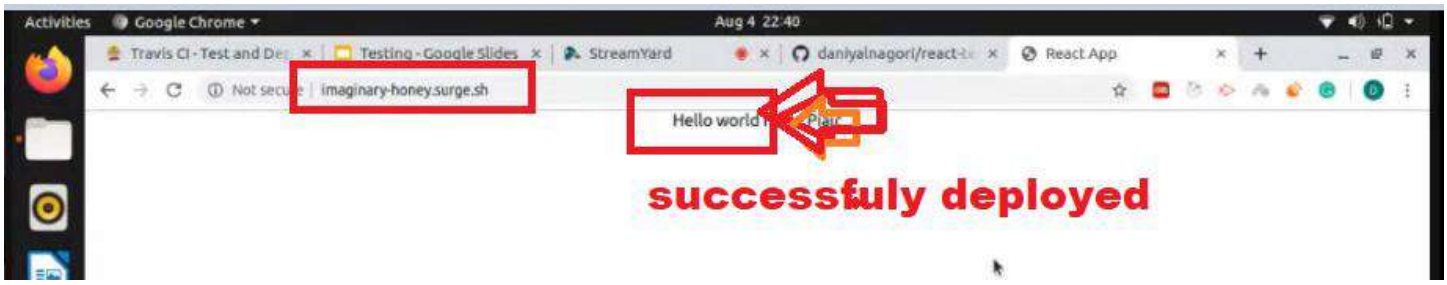
- deploy on surge
- create a build and go to build folder and run surge command



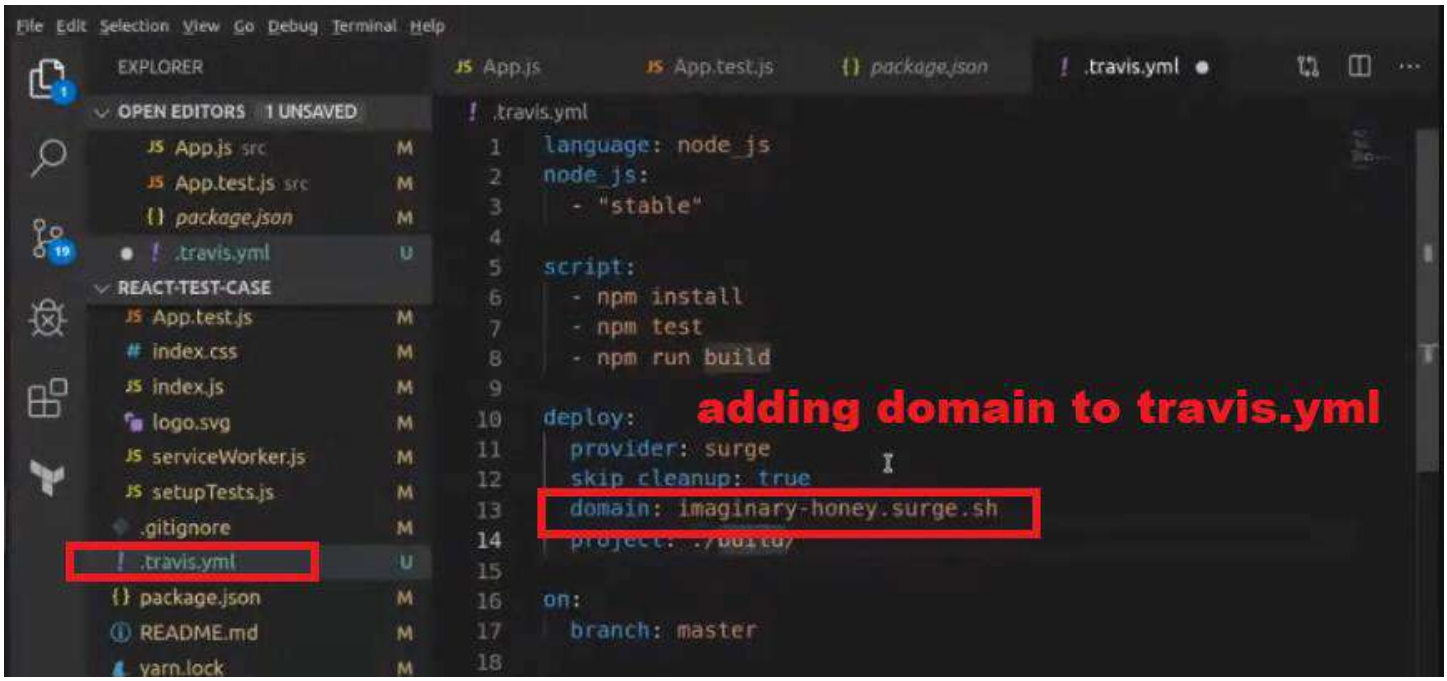
- Run by your id
- Click ok to the domain on surge
- Click ok and publish



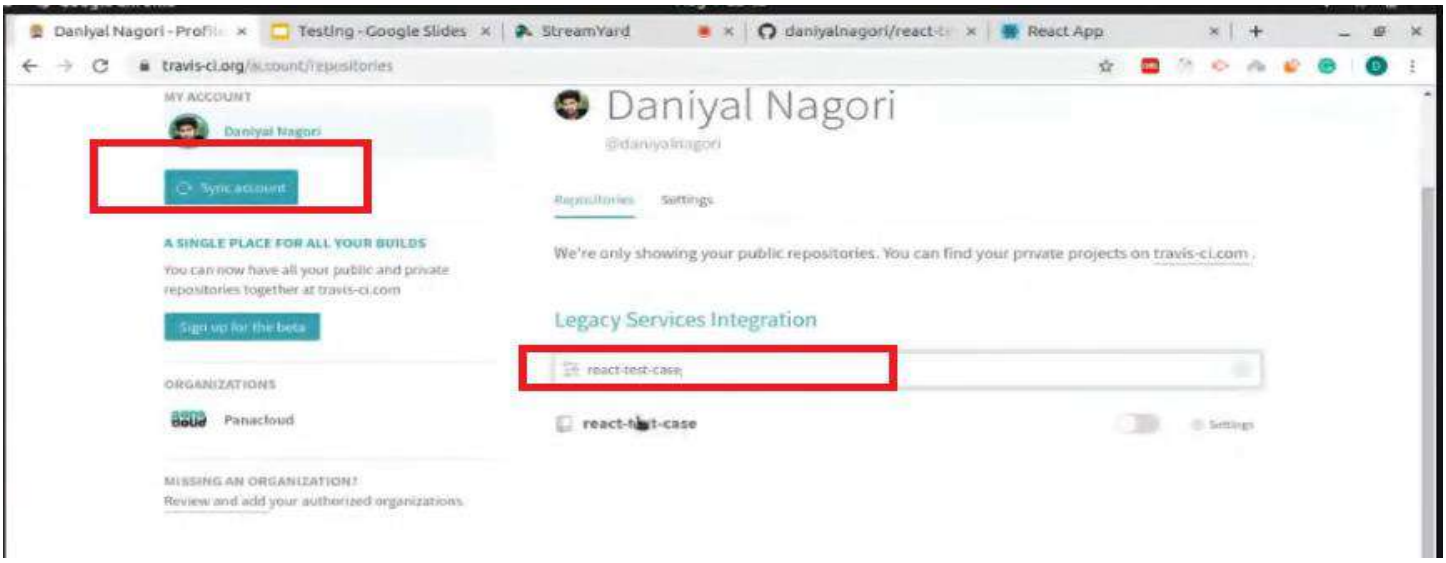
- Now copy this domain and check out on the browser if its been successfully deployed or not
- So it's been successfully deployed as shown



- Now to run it manually add this domain to travis.yml as



- sign up Travis CI with GitHub
- sync account with Github
- Travis will then have access to all repositories of GitHub



- Now write the react test case in it you will get this repository as shown so just activate this by click to an icon on the right side
- Add one more case to travis.yml and as well as node.js modules
- So after running this repository everything will be deployed on Travis CI

```
> react-test-case@0.1.0 test /home/travis/build/daniyalnagori/react-test-case
> react-scripts test

PASS src/App.test.js
  ✓ renders learn react link (26ms)
  ✓ text case (4ms)
  ✓ text case (2ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        1.798s
Ran all test suites.
The command "npm test" exited with 0.
$ npm run build

> react-test-case@0.1.0 build /home/travis/build/daniyalnagori/react-test-case
> react-scripts build

Creating an optimized production build...
Compiled successfully.

File sizes after gzip:

  39.39 KB  build/static/js/2.b97b01b1.chunk.js
  781 B     build/static/js/runtime-main.bb79b400.js
  547 B     build/static/css/main.5f361e03.chunk.css
  494 B     build/static/js/main.f6fd9850.chunk.js

The project was built assuming it is hosted at /.
You can control this with the homepage field in your package.json.
```

Now check this deployment on your browser

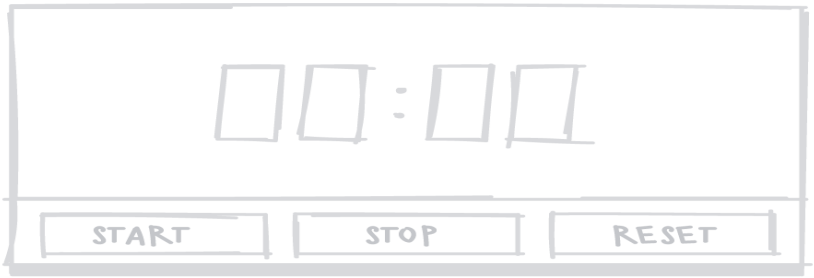
Session 15(A): Test React Components in TypeScript

Basic Pomodoro Timer App-Test Driven React App

We'll be building a basic Pomodoro timer app consisting of some UI components. Each component will have a separate set of tests in a corresponding test file. First of all, we could create epics and user stories as follows based on our project requirements.

EPIC	USER STORY	ACCEPTANCE CRITERIA
As a user, I need to use the timer so that I can manage my time.	As a user, I need to start the timer so that I can count down my time.	Ensure the user can: *start the timer *see the timer start counting down *Counting down the time should not be interrupted even if the user clicks the start button more than once.*
	As a user, I need to stop the timer so that I can count down my time only when needed.	Ensure the user can: *stop the timer *see the timer stopped *Nothing should happen even if the user clicks the stop button more than once.*
	As a user, I need to reset the timer so that I can count down my time from the beginning.	Ensure the user is able to: *reset the timer *see the timer reset to the default

Wireframe



- Go to the terminal
- go to the GitHub document

- create a react-app timer app by using npx with a typescript template

```
Adil:~ adil$ cd Documents/GitHub/
Adil:GitHub adil$ npx create-react-app timer-app --template typescript
npx: installed 98 in 20.825s
```

Once react App installation is completed install jest enzyme for typescript
To install all dependencies run this command

```
npm i -D ts-jest jest-fetch-mock enzyme enzyme-adapter-react-16 enzyme-to-JSON @types/enzym
@types/enzyme-adapter-react-16 --save-exact
```

- go to your project of timer react and in the packet. jason enter the following coding

Packet.jason

```
1.  "jest": {
2.  "collectCoverageFrom": [
3.  "src/**/*.{js,jsx,ts,tsx}",
4.  "!src/**/*.d.ts",
5.  "!src/index.tsx",
6.  "!src/serviceWorker.ts"
7.  ],
8.  "coveragePathIgnorePatterns": [
9.  "./src/*/*.types.{ts,tsx}",
10. "./src/index.tsx",
11. "./src/serviceWorker.ts"
12. ],
13. "coverageReporters": [
14. "json",
15. "lcov",
16. "text-summary",
17. "clover"
18. ],
19. "coverageThreshold": {
20. "global": {
21. "statements": 95,
22. "branches": 95,
23. "lines": 95,
24. "functions": 95
25. }
26. },
27. "snapshotSerializers": [
28. "enzyme-to-json/serializer"
29. ],
30. "transform": {
31. "^.+\\.(js|jsx|ts|tsx)$": "<rootDir>/node_modules/ts-jest"
32. },
33. "transformIgnorePatterns": [
34. "[/\\\\]node_modules[/\\\\].+\\.(js|jsx|ts|tsx)$",
35. "^.+\\.module\\.(css|sass|scss)$"
36. ],
37. "moduleNameMapper": {
38. "^react-native$": "react-native-web",
39. "^.+\\.module\\.(css|sass|scss)$": "identity-obj-proxy",
40. "src/(.*)$": "<rootDir>/src/$1"
41. }
```

In src/setupTests.ts follow this code

setupTests.ts

```
/* eslint-disable import/no-extraneous-dependencies */

import Enzyme from 'enzyme';

import ReactSixteenAdapter from 'enzyme-adapter-react-16';

Enzyme.configure({ adapter: new ReactSixteenAdapter() });
```

- in packet. jason replace the prev script with the new one as

packet.jason

```
"scripts": {

   "start": "react-scripts start",

   "build": "react-scripts build",

   "test": "react-scripts test",

   "test:coverage": "react-scripts test --coverage --runInBand --watchAll=false",

   "eject": "react-scripts eject",

   "lint": "eslint --ext .js,.jsx,.ts,.tsx src --color",

   "format": "prettier --write src/**/*.{ts,tsx,scss,css,json}",

   "isready": "npm run format && npm run lint && npm run test:coverage && npm run build"

 },
```

- now create a file of tests/App.test.tsx
- import react and shallow enzyme in it
- run the test for rendering app component

App.test.tsx

```
import React from 'react';

import { shallow } from 'enzyme';

import App from '../App';

test('renders the component', ( ) => {

  const component = shallow(<App />);

  expect(component).toMatchSnapshot();

});
```

- Run test on your terminal

```
Adil:GitHub adil$ yarn run test
yarn run v1.19.1
warning package.json: No license field
error Command "test" not found.
info Visit https://yarnpkg.com/en/docs/cli/run for documentation about this command.
Adil:GitHub adil$
```

- When running npm run test a new snapshot will be created and there will be a new folder generated __snapshots__ with it.

```
ts-jest[versions] (WARN) Version 24.9.0 of jest installed has not been tested with ts-je
. If you're experiencing issues, consider using a supported version (>=26.0.0 <27.0.0-0)
Please do not report issues in ts-jest if you are using unsupported versions.
ts-jest[versions] (WARN) Version 3.7.5 of typescript installed has not been tested with
-jest. If you're experiencing issues, consider using a supported version (>=3.8.0 <4.0.0
). Please do not report issues in ts-jest if you are using unsupported versions.
 PASS  src/tests/App.test.tsx (10.292s)
  ✓ renders the component (31ms)

  › 1 snapshot written.
Snapshot Summary
  › 1 snapshot written from 1 test suite.

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   1 written, 1 total
Time:        12.15s
Ran all test suites related to changed files.

Watch Usage
  › Press a to run all tests.
  › Press f to run only failed tests.
  › Press q to quit watch mode.
  › Press p to filter by a filename regex pattern.
  › Press t to filter by a test name regex pattern.
  › Press Enter to trigger a test run.
```

- Run the following command for saving

```
Adil:timer-app adil$ npm install --save @types/react-dom
⠇⠸⠸⠸
```

App.spec.tsx

src > components > App > App.spec.tsx > describe('App Component renders') callback

```
1   import React from 'react';
2   import { shallow } from 'enzyme';
3   import App from './App';
4
5   describe('App Component renders', () => {
6       it('should render a div', () => {
7           let container = shallow(<App />);
8           expect(container.find('div').length).toEqual(1);
9       });
10  });
```

- now create a component with App component folder
- add a new file App.tsx

App.tsx

```
import React from 'react';

const App=()<div className='container'/>;

export default App;
```

- run this ,it will be passed as sshown

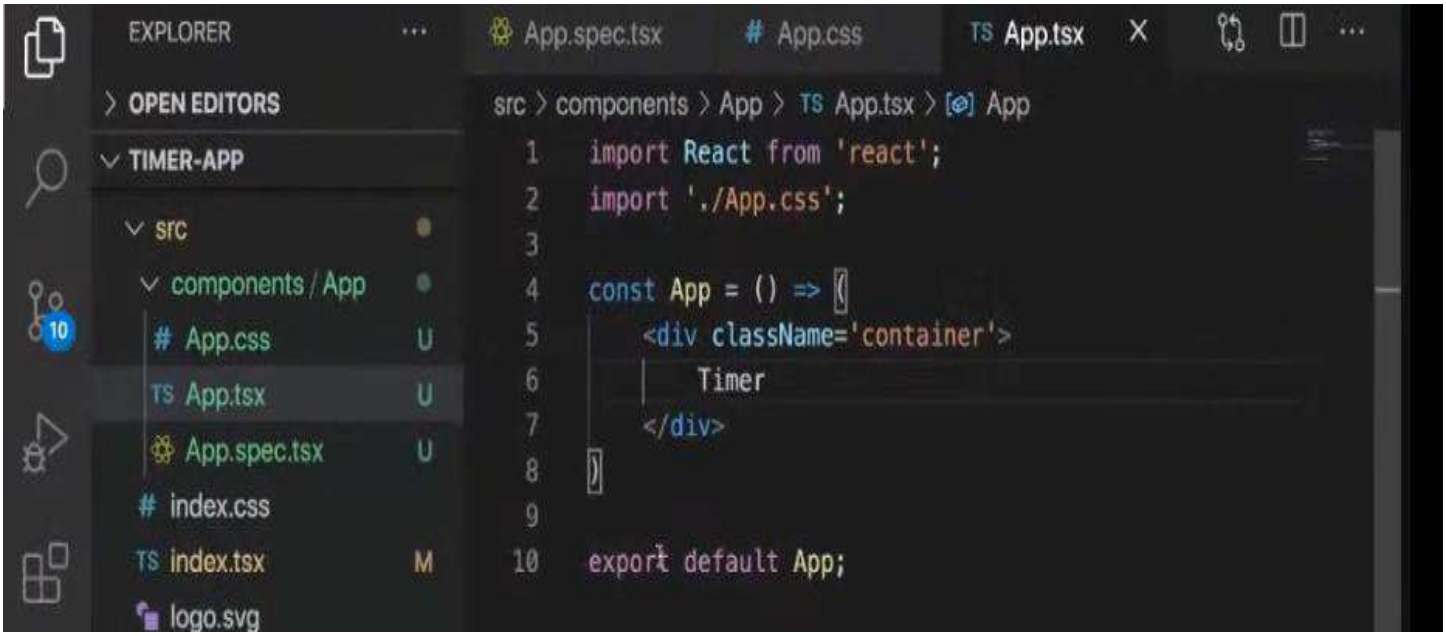
```
PASS src/components/App/App.spec.tsx
  App Component renders
    ✓ should render a div (97ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        3.352s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

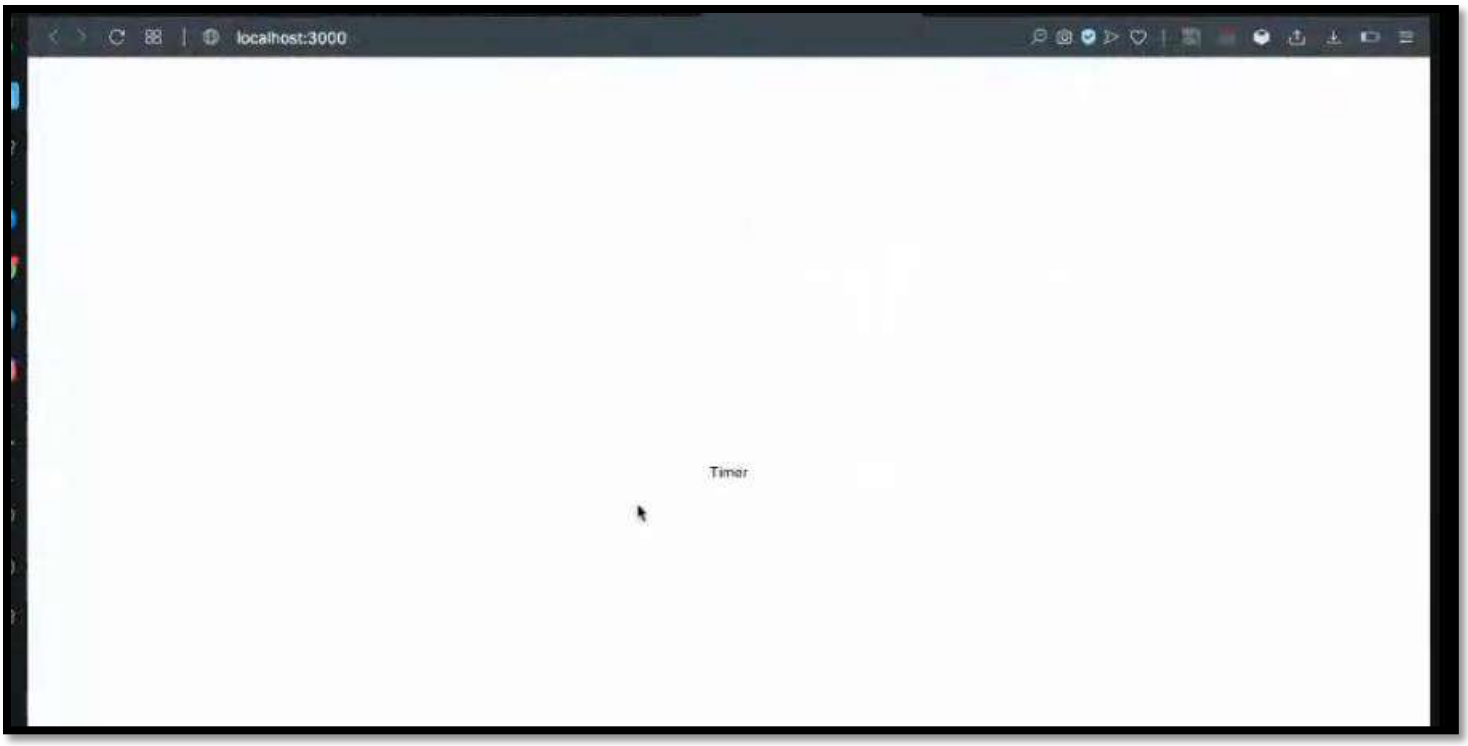
- so write a code test and run it to check if the code works or not
- create another component App.css and follow this CSS style

App.css

```
.app-container {
        height: 100vh;
        width: 100vw;
        align-items: center;
        display: flex;

        justify-content: center;

}
```

Import this App.css in the App.tsx and in the div section enter timer and look into your browser





- create a component of timer and create test compnent of in the App.component
- import timer
- add a second test in our test

```
it('should render the timer component'()=>{
  expect(container.contains matchingElement(<timer/>)).toEqual(true);
});
```

App.spec.tsx × # App.css TS App.tsx

src > components > App > App.spec.tsx > describe('App Component renders') callback

```
1   import React from 'react';
2   import { shallow } from 'enzyme';
3   import App from './App';
4   import Timer from '../Timer/Timer';
5
6   describe('App Component renders', () => {
7
8       let container = shallow(<App />);
9
10      it('should render a div', () => {
11          expect(container.find('div').length).toEqual(1);
12      });
13
14      it('should render the Timer component', () => {
15          expect(container.containsMatchingElement(<Timer />)).toEqual(true);
16      });
17
18  });
```

- create a test case in Timer.Spec.tsx

```
import React from'react';
import { shallow } from 'enzyme';
import Timer from'./Timer';

let container =shallow(<timer>/)
describe('App component renders'), ()=> {
  it('should rendera div ') ( ) => {
    expect(container.find('div')).length).toBeGreaterThanOrEqual(1);
  });
```

```
App.spec.tsx ●   Timer.spec.tsx ×   # App.css   TS App.tsx
src > components > Timer > Timer.spec.tsx > ...
1   import React from 'react';
2   import { shallow } from 'enzyme';
3   import Timer from './Timer';
4
5   describe('App Component renders', () => {
6
7       let container = shallow(<Timer />);
8
9       it('should render a div', () => {
10          expect(container.find('div').length).toBeGreaterThanOrEqual(1);
11      });
```

- create a Timer component

```
import React from "react";
import { shallow } from "enzyme";
import App from "./App";
import Timer from "../Timer/Timer";

describe("App", () => {
  let container

  beforeEach(() => (container = shallow(<App />)))

  it("should render a <div />", () => {
    expect(container.find("div").length).toEqual(1)
  })

  it("should render the Timer Component", () => {
    expect(container.containsMatchingElement(<Timer />)).toEqual(true)
  })

})
```

- let's add custom CSS style
- create index.css style

index.css

```
:root {
        --timer-background-color: #FFFFFF;
--timer-border: 1px solid #000000;
        --timer-height: 70%;
        --timer-width: 70%; }
```

```
body, div, p {
	margin: 0;
	padding: 0;
}

@media screen and (max-width: 1024px) {
	:root {
		--timer-height: 100%;
		--timer-width: 100%;
}
```

- create the Timer.css file under the directory components/Timer:

Timer.css

```
.timer-container {
	background-color: var(--timer-background-color);
	border: var(--timer-border);
	height: var(--timer-height);
	width: var(--timer-width);
```

- to update Timer.jsx to import the Timer.css file

```
import React, { Component } from "react";
import "./Timer.css";
```

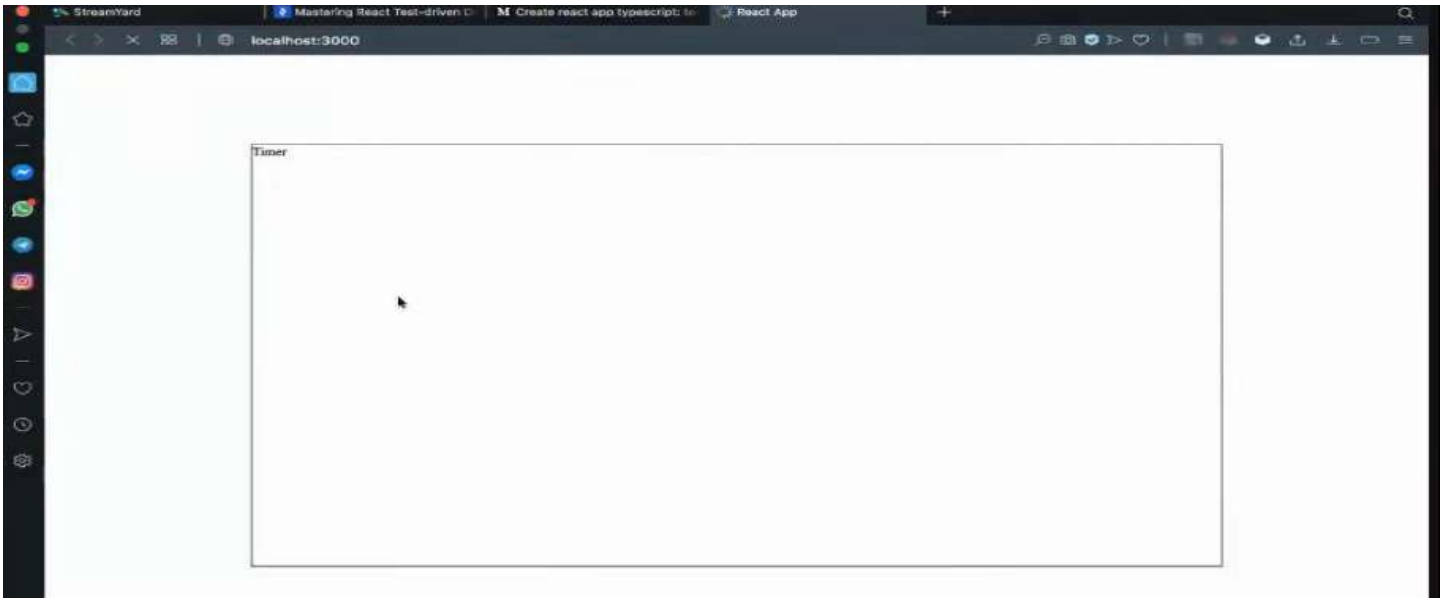
```
# index.css    TS Timer.tsx
src > components > Timer > TS Timer.tsx > ...
1   import React, { Component } from 'react';
2   import './Timer.css';
3
4   class Timer extends Component {
5     constructor(props:any) {
6       super(props);
7       this.state = {
8         minutes: 25,
9         seconds: 0,
10          isOn: false
11      };
12    }
13
14    startTimer() {
15      console.log('Starting timer.');
16    }
17
18    stopTimer() {
19      console.log('Stopping timer.');
20    }
```

```
23    }
24
25    render = () => {
26      return (<div className="timer-container">Timer</
27    };
28  }
29
30  export default Timer;
31
```

Run the test case so three of them are passed

```
ts-jest[versions] (WARN) Version 24.9.0 of jest installed has not been teste
. If you're experiencing issues, consider using a supported version (>=26.0.
Please do not report issues in ts-jest if you are using unsupported versions
ts-jest[versions] (WARN) Version 3.7.5 of typescript installed has not been
-jest. If you're experiencing issues, consider using a supported version (>=
). Please do not report issues in ts-jest if you are using unsupported versi
ts-jest[versions] (WARN) Version 24.9.0 of jest installed has not been teste
. If you're experiencing issues, consider using a supported version (>=26.0.
Please do not report issues in ts-jest if you are using unsupported versions
ts-jest[versions] (WARN) Version 3.7.5 of typescript installed has not been
-jest. If you're experiencing issues, consider using a supported version (>=
). Please do not report issues in ts-jest if you are using unsupported versi
 PASS  src/components/Timer/Timer.spec.tsx (16.289s)
 PASS  src/components/App/App.spec.tsx (16.481s)

Test Suites: 2 passed, 2 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        21.01s
Ran all test suites.

Watch Usage: Press w to show more.
```

- so we have a timer with the border as



- now let us create css style for timer button
- first create a test for it as
- First, we need to update the Timer.spec.js file to check for the existence of the TimerButton component in the Timer component:

```
it("should render instances of the TimerButton component", () => {
    expect(container.find("TimerButton").length).toEqual(3)
  })
```

```
# index.css    TS Timer.tsx    Timer.spec.tsx ●
src > components > Timer > Timer.spec.tsx > describe('App Component renders') callback
1   import React from 'react';
2   import { shallow } from 'enzyme';
3   import Timer from './Timer';
4
5   describe('App Component renders', () => {
6
7       let container = shallow(<Timer />);
8
9       it('should render a div', () => {
10          expect(container.find('div').length).toBeGreaterThanOrEqual(1);
11      });
12
13      it("should render instances of the TimerButton component", () => {
14          expect(container.find("TimerButton").length).toEqual(3)
15       })
16
17  });
```

- Now, let’s add the TimerButton.spec.js file in a new directory called TimerButton under the src/components directory, and let’s add the test to the file like this:

TimerButton.spec.js

```
import React from "react";
import { shallow } from "enzyme";
import TimerButton from "./TimerButton";

describe("TimerButton", () => {
  let container

  beforeEach(() => {
    container = shallow(
      <TimerButton
        buttonAction={jest.fn()}
        buttonValue={""}
      />
    )
  })

  it("should render a <div />", () => {
    expect(container.find("div").length).toBeGreaterThanOrEqual(1)
  })
})
```

- Now, if you run the test, you will see the test fails.
- Let's create the TimerButton.jsx file for the TimerButton component:

```
import React from 'react';
import PropTypes from 'prop-types';

const TimerButton = ({ buttonAction, buttonValue }) => (
  <div className="button-container" />
);

TimerButton.propTypes = {
  buttonAction: PropTypes.func.isRequired,
  buttonValue: PropTypes.string.isRequired,
};


export default TimerButton;
```

- If you run the npm test at this stage, the test should render instances of the TimerButton component but will fail since we haven't added the TimerButton components to the Timer component yet.

- Let's import the TimerButton component and add three TimerButton components in the render method in Timer.jsx:

Timer.jsx:

```
render = () => {
    return (
      <div className="timer-container">
        <div className="time-display"></div>
        <div className="timer-button-container">
          <TimerButton buttonAction={this.startTimer} buttonValue={'Start'} />
          <TimerButton buttonAction={this.stopTimer} buttonValue={'Stop'} />
          <TimerButton buttonAction={this.resetTimer} buttonValue={'Reset'} />
      </div>

     </div>

    );

  };
```

- Also, let's create a file called TimerButton.css in the TimerButton directory under the src/components directory:

```
.button-container {

  flex: 1 1 auto;

  text-align: center;

  margin: 0px 20px;

  border: var(--button-border);

  font-size: var(--button-text-size);

}
```

```
.button-container:hover {

  cursor: pointer;
```

- Let’s update the TimerButton.jsx accordingly to import the TimerButton.css file and to display the button value:

TimerButton.jsx

```
import React from 'react';

import PropTypes from 'prop-types';

import './TimerButton.css';


const TimerButton = ({ buttonAction, buttonValue }) => (

  <div className="button-container">

    <p className="button-value">{buttonValue}</p>

  </div>

);


TimerButton.propTypes = {

  buttonAction: PropTypes.func.isRequired,

  buttonValue: PropTypes.string.isRequired,

};


export default TimerButton;
```

- Also, we need to update the Timer.css to align the three buttons horizontally, so let’s update the Timer.css file as well:

```
import React from 'react';
import PropTypes from 'prop-types';
import './TimerButton.css';

const TimerButton = ({ buttonAction, buttonValue }) => (

<div
 className="button-container">
    <p className="button-value">{buttonValue}</p>
  </div>
);

TimerButton.propTypes = {
  buttonAction: PropTypes.func.isRequired,
  buttonValue: PropTypes.string.isRequired,
};

export default TimerButton;
```

- If you run the React app now, you will see a screen as follows:

Refactoring the Timer

We're going to refactor the Timer since we want to implement functions like startTimer, stopTimer, restartTimer, and resetTimer.

- Let's update the Timer.spec.js file first:

```
describe('mounted Timer', () => {
  let container;

  beforeEach(() => (container = mount(<Timer />)));

  it('invokes startTimer when the start button is clicked', () => {
    const spy = jest.spyOn(container.instance(), 'startTimer');
    container.instance().forceUpdate();
    expect(spy).toHaveBeenCalledTimes(0);
    container.find('.start-timer').first().simulate('click');
```

```
  it('invokes stopTimer when the stop button is clicked', () => {
    const spy = jest.spyOn(container.instance(), 'stopTimer');
    container.instance().forceUpdate();
    expect(spy).toHaveBeenCalledTimes(0);
    container.find('.stop-timer').first().simulate('click');
    expect(spy).toHaveBeenCalledTimes(1);
  });

  it('invokes resetTimer when the reset button is clicked', () => {
    const spy = jest.spyOn(container.instance(), 'resetTimer');
    container.instance().forceUpdate();
    expect(spy).toHaveBeenCalledTimes(0);
    container.find('.reset-timer').first().simulate('click');
    expect(spy).toHaveBeenCalledTimes(1);
  });
});
```

- If you run the test, you will see the added tests fail since we haven't updated the TimerButton component yet. Let's update the TimerButton component to add the click event:

```
const TimerButton = ({ buttonAction, buttonValue }) => (
  <div className="button-container" onClick={() => buttonAction()}>
    <p className="button-value">{buttonValue}</p>
  </div>
```

```
    container.instance().forceUpdate();
    container.find('.start-timer').first().simulate('click');
    expect(container.instance().state.isOn).toEqual(true);
  });

  it('should change isOn state false when the stop button is clicked', () => {
    container.instance().forceUpdate();
    container.find('.stop-timer').first().simulate('click');

it('should change isOn state false when the reset button is clicked', () => {
    container.instance().forceUpdate();
    container.find('.stop-timer').first().simulate('click');
    expect(container.instance().state.isOn).toEqual(false);
    expect(container.instance().state.minutes).toEqual(25);
    expect
  });
```

- If you run the tests, you will see them fail since we haven't implemented each method yet. So let's implement each function to pass the tests:

```
startTimer() {
  this.setState({ isOn: true });
 }

 stopTimer() {
  this.setState({ isOn: false });
 }

 resetTimer() {
  this.stopTimer();
 this.setState({
    minutes: 25,
```

- You will see the tests pass if you run them. Now, let’s implement the remaining functions in Timer.jsx:

```
import React, { Component } from 'react';
import './Timer.css';
import TimerButton from '../TimerButton/TimerButton';

class Timer extends Component {
constructor(props) {
super(props);
this.state = {
minutes: 25,
seconds: 0,
isOn: false,
};

this.startTimer = this.startTimer.bind(this);
this.stopTimer = this.stopTimer.bind(this);
this.resetTimer = this.resetTimer.bind(this);
}

startTimer() {
if (this.state.isOn === true) {
```

```
return;
}
this.myInterval = setInterval(() => {
const { seconds, minutes } = this.state;

if (seconds > 0) {
this.setState(({ seconds }) => ({
seconds: seconds - 1,
}));
}
if (seconds === 0) {
if (minutes === 0) {
clearInterval(this.myInterval);
} else {
this.setState(({ minutes }) => ({
minutes: minutes - 1,
seconds: 59,
}));
}
}
}, 1000);
this.setState({ isOn: true });
}

stopTimer() {
clearInterval(this.myInterval);
this.setState({ isOn: false });
}


                resetTimer() {
                this.stopTimer();
                this.setState({
                minutes: 25,

      seconds: 0,
    });
  }

render = () => {
    const { minutes, seconds } = this.state;

    return (
      <div className="timer-container">
        <div className="time-display">
          {minutes}:{seconds < 10 ? `0${seconds}` : seconds}
        </div>
        <div className="timer-button-container">
          <TimerButton
```

```
          buttonAction={this.startTimer}
          buttonValue={'Start'}
        />
        <TimerButton
          className="stop-timer"
          buttonAction={this.stopTimer}
          buttonValue={'Stop'}
        />
        <TimerButton
          className="reset-timer"
          buttonAction={this.resetTimer}
          buttonValue={'Reset'}
        />
      </div>
    </div>
  );
};
}

export default Timer;
```

- So, that's how we have developed a basic React app using TDD. If the user stories and the acceptance criteria are more detailed, the test cases can be written more precisely, thus contributing even more.

Session 16: Intro to GraphQL

Basics Tutorial - Introduction

GraphQL is a new API standard that provides a more efficient, powerful and flexible alternative to REST. It was developed and open-sourced by Facebook and is now maintained by a large community of companies and individuals from all over the world.

APIs have become ubiquitous components of software infrastructures. In short, an API defines how a client can load data from a server.

At its core, GraphQL enables declarative data fetching where a client can specify exactly what data it needs from an API. Instead of multiple endpoints that return fixed data structures, a GraphQL server only exposes a single endpoint and responds with precisely the data a client asked for.

GraphQL - A Query Language for APIs

Most applications today have the need to fetch data from a server where that data is stored in a database. It's the responsibility of the API to provide an interface to the stored data that fits an application's needs.

GraphQL is often confused with being a database technology. This is a misconception, GraphQL is a query language for APIs - not databases. In that sense it's database agnostic and effectively can be used in any context where an API is used.

Core Concepts

In this chapter, you'll learn about some fundamental language constructs of GraphQL. That includes a first glimpse at the syntax for defining types as well as sending queries and mutations.

The Schema Definition Language (SDL)

GraphQL has its own type system that's used to define the schema of an API. The syntax for writing schemas is called Schema Definition Language (SDL).

Here is an example of how we can use the SDL to define a simple type called Person:

```
type Person {
  name: String!
  age: Int!
}
```

This type has two fields, they're called name and age and are respectively of type String and Int. The ! following the type means that this field is required.

It's also possible to express relationships between types. In the example of a blogging application, a Person could be associated with a Post:

```
type Post {
  title: String!
  author: Person!
}
```

Conversely, the other end of the relationship needs to be placed on the Person type:

```
type Person {
  name: String!
  age: Int!
  posts: [Post!]!
}
```

Note that we just created a one-to-many-relationship between Person and Post since the posts field on Person is actually an array of posts.

Fetching Data with Queries

When working with REST APIs, data is loaded from specific endpoints. Each endpoint has a clearly defined structure of the information that it returns. This means that the data requirements of a client are effectively encoded in the URL that it connects to.

The approach that's taken in GraphQL is radically different. Instead of having multiple endpoints that return fixed data structures, GraphQL APIs typically only expose a single endpoint. This works because the structure of the data that's returned is not fixed. Instead, it's completely flexible and lets the client decide what data is actually needed.

That means that the client needs to send more information to the server to express its data needs - this information is called a query.

Basic Queries

Let's take a look at an example query that a client could send to a server:

```
{
  allPersons {
    name
  }
}
```

The allPersons field in this query is called the root field of the query. Everything that follows the root field, is called the payload of the query. The only field that's specified in this query's payload is name.

This query would return a list of all persons currently stored in the database. Here's an example response:

```
{
  "allPersons": [
    { "name": "Johnny" },
    { "name": "Sarah" },
    { "name": "Alice" }
  ]
}
```

Notice that each person only has the name in the response, but the age is not returned by the server. That's exactly because name was the only field that was specified in the query.

If the client also needed the persons' age, all it has to do is slightly adjust the query and include the new field in the query's payload:

```
{
  allPersons {
    name
    age
  }
}
```

One of the major advantages of GraphQL is that it allows for naturally querying nested information. For example, if you wanted to load all the posts that a Person has written, you could simply follow the structure of your types to request this information:

```
{
  allPersons {
    name
    age
    posts {
      title
    }
  }
}
```

Queries with Arguments

In GraphQL, each field can have zero or more arguments if that's specified in the schema. For example, the allPersons field could have a last parameter to only return up to a specific number of persons. Here's what a corresponding query would look like:

```
{
  allPersons(last: 2) {
    name
  }
}
```

Writing Data with Mutations

Next to requesting information from a server, the majority of applications also need some way of making changes to the data that's currently stored in the backend. With GraphQL, these changes are made using so-called mutations. There generally are three kinds of mutations:

- creating new data
- updating existing data
- deleting existing data

Mutations follow the same syntactical structure as queries, but they always need to start with the mutation keyword. Here's an example for how we might create a new Person:

```
mutation {
  createPerson(name: "Bob", age: 36) {
    name
    age
  }
}
```

Notice that similar to the query we wrote before, the mutation also has a root field - in this case it's called createPerson. We also already learned about the concepts of arguments for fields. In this case,

the createPerson field takes two arguments that specify the new person's name and age.

Like with a query, we're also able to specify a payload for a mutation in which we can ask for different properties of the new Person object. In our case, we're asking for the name and the age - though admittedly that's not super helpful in our example since we obviously already know them as we pass them into the mutation. However, being able to also query information when sending mutations can be a very powerful tool that allows you to retrieve new information from the server in a single roundtrip!
The server response for the above mutation would look as follows:

```
"createPerson": {
  "name": "Bob",
  "age": 36,
}
```

One pattern you'll often find is that GraphQL types have unique IDs that are generated by the server when new objects are created. Extending our Person type from before, we could add an id like this:

```
type Person {
  id: ID!
  name: String!
  age: Int!
}
```

Now, when a new Person is created, you could directly ask for the id in the payload of the mutation, since that is information that wasn't available on the client beforehand:

```
mutation {
  createPerson(name: "Alice", age: 36) {
    id
  }
}
```

Realtime Updates with Subscriptions

Another important requirement for many applications today is to have a realtime connection to the server in order to get immediately informed about important events. For this use case, GraphQL offers the concept of subscriptions.

When a client subscribes to an event, it will initiate and hold a steady connection to the server. Whenever that particular event then actually happens, the server pushes the corresponding data to the client. Unlike queries and mutations that follow a typical "request-response-cycle", subscriptions represent a stream of data sent over to the client.

Subscriptions are written using the same syntax as queries and mutations. Here's an example where we subscribe on events happening on the Person type:

```
subscription {
  newPerson {
    name
    age
  }
}
```

After a client sent this subscription to a server, a connection is opened between them. Then,

whenever a new mutation is performed that creates a new Person, the server sends the information about this person over to the client:

```
{
  "newPerson": {
    "name": "Jane",
    "age": 23
  }
}
```

Defining a Schema

Now that you have a basic understanding of what queries, mutations, and subscriptions look like, let's put it all together and learn how you can write a schema that would allow you to execute the examples you've seen so far.

The schema is one of the most important concepts when working with a GraphQL API. It specifies the capabilities of the API and defines how clients can request the data. It is often seen as a contract between the server and client.

Generally, a schema is simply a collection of GraphQL types. However, when writing the schema for an API, there are some special root types:

```
type Query { ... }
type Mutation { ... }
type Subscription { ... }
```

The Query, Mutation, and Subscription types are the entry points for the requests sent by the client. To enable the allPersons-query that we saw before, the Query type would have to be written as follows:

```
type Query {
  allPersons: [Person!]!
}
```

allPersons is called a root field of the API. Considering again the example where we added the last argument to the allPersons field, we'd have to write the Query as follows:

```
type Query {
  allPersons(last: Int): [Person!]!
}
```

Similarly, for the createPerson-mutation, we'll have to add a root field to the Mutation type:

```
type Mutation {
  createPerson(name: String!, age: Int!): Person!
}
```

Notice that this root field takes two arguments as well, the name and the age of the new Person.

Finally, for the subscriptions, we'd have to add the newPerson root field:

```
type Subscription {
  newPerson: Person!
}
```

Putting it all together, this is the full schema for all the queries and mutation that you have seen in this

chapter:

```
type Query {
  allPersons(last: Int): [Person!]!
}

type Mutation {
  createPerson(name: String!, age: Int!): Person!
}

type Subscription {
  newPerson: Person!
}

type Person {
  name: String!
  age: Int!
  posts: [Post!]!
}

type Post {
  title: String!
  author: Person!
```

Big Picture (Architecture)

GraphQL has been released only as a specification. This means that GraphQL is in fact not more than a long document that describes in detail the behaviour of a GraphQL server.

Use Cases

In this section, we'll walk you through 3 different kinds of architectures that include a GraphQL server:

- GraphQL server with a connected database
- GraphQL server that is a thin layer in front of a number of third party or legacy systems and integrates them through a single GraphQL API
- A hybrid approach of a connected database and third party or legacy systems that can all be accessed through the same GraphQL API

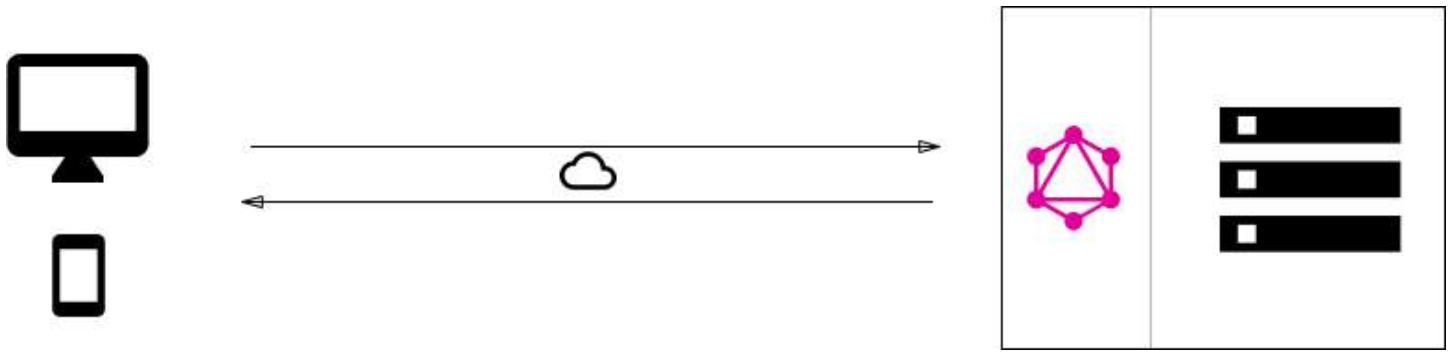
All three architectures represent major use cases of GraphQL and demonstrate the flexibility in terms of the context where it can be used.

1. GraphQL server with a connected database

This architecture will be the most common for greenfield projects. In the setup, you have a single (web) server that implements the GraphQL specification. When a query arrives at the GraphQL server, the server reads the query's payload and fetches the required information from the database. This is called resolving the query. It then constructs the response object as described in the official specification and returns it to the client.

It's important to note that GraphQL is actually transport-layer agnostic. This means it can potentially be used with any available network protocol. So, it is potentially possible to implement a GraphQL server based on TCP, WebSockets, etc.

GraphQL also doesn't care about the database or the format that is used to store the data. You could use a SQL database like AWS Aurora or a NoSQL database like MongoDB.

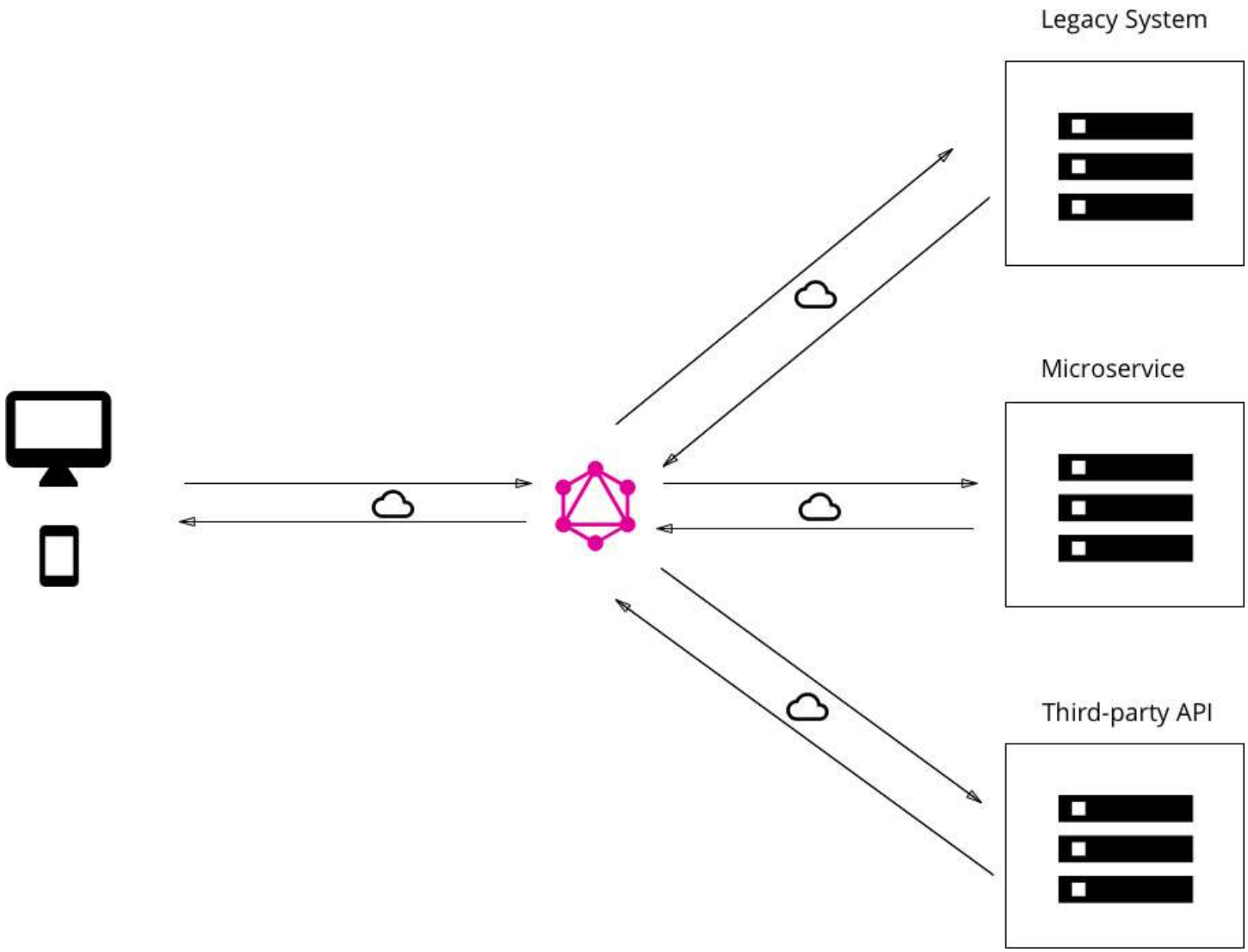


2. GraphQL layer that integrates existing systems

Another major use case for GraphQL is the integration of multiple existing systems behind a single, coherent GraphQL API. This is particularly compelling for companies with legacy infrastructures and many different APIs that have grown over years and now impose a high maintenance burden. One major problem with these legacy systems is that they make it practically impossible to build innovative products that need access to multiple systems.

In that context, GraphQL can be used to unify these existing systems and hide their complexity behind a nice GraphQL API. This way, new client applications can be developed that simply talk to the GraphQL server to fetch the data they need. The GraphQL server is then responsible for fetching the data from the existing systems and package it up in the GraphQL response format.

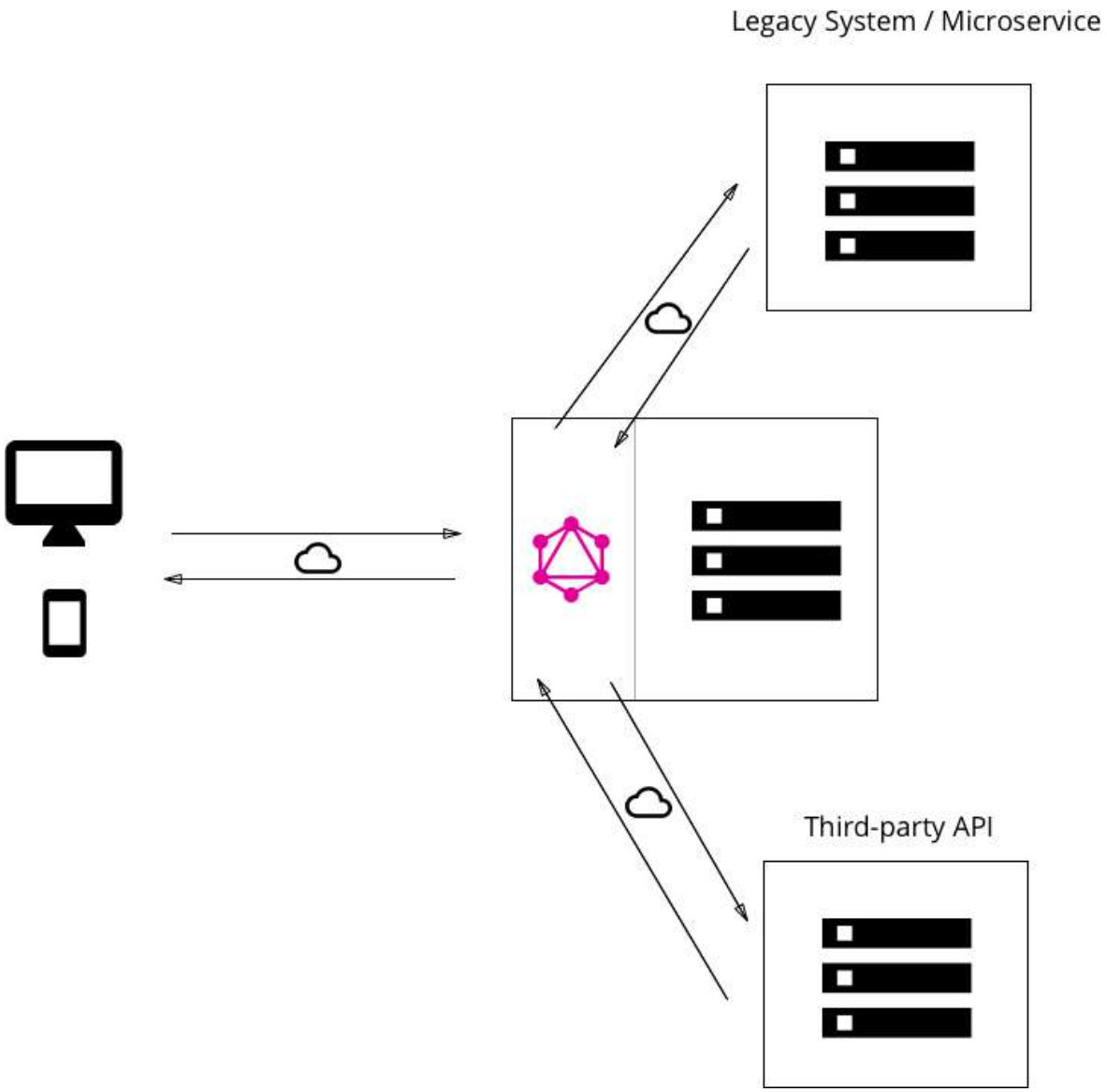
Just like in the previous architecture where the GraphQL server didn't care about the type of database being used, this time it doesn't care about the data sources that it needs to fetch the data that's needed to resolve a query.



3. Hybrid approach with connected database and integration of existing system

Finally, it's possible to combine the two approaches and build a GraphQL server that has a connected database but still talks to legacy or third—party systems.

When a query is received by the server, it will resolve it and either retrieve the required data from the connected database or some of the integrated APIs.



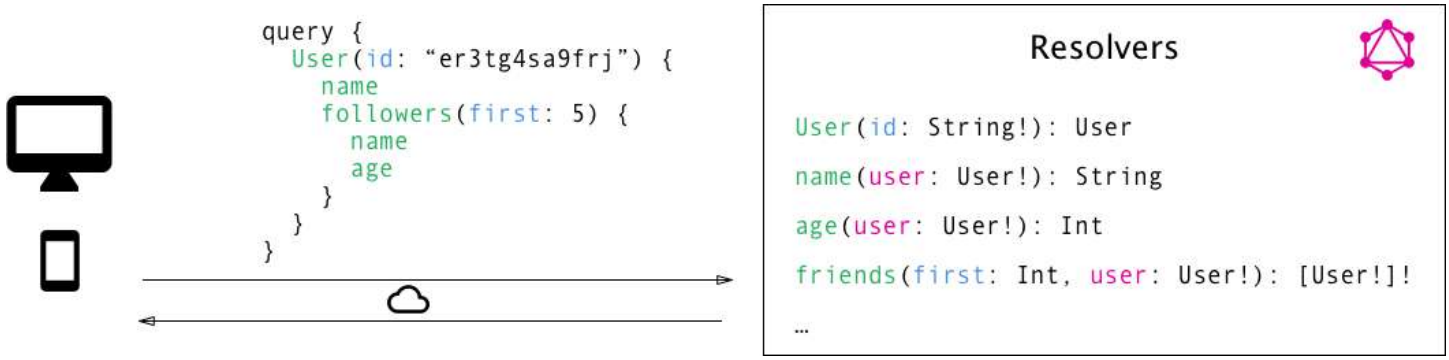
Resolver Functions

But how do we gain this flexibility with GraphQL? How is it that it's such a great fit for these very different kinds of use cases?

As you learned in the previous chapter, the payload of a GraphQL query (or mutation) consists of a set of fields. In the GraphQL server implementation, each of these fields actually corresponds to

exactly one function that's called a resolver. The sole purpose of a resolver function is to fetch the data for its field.

When the server receives a query, it will call all the functions for the fields that are specified in the query's payload. It thus resolves the query and is able to retrieve the correct data for each field. Once all resolvers returned, the server will package data up in the format that was described by the query and send it back to the client.



GraphQL Client Libraries

GraphQL is particularly great for frontend developers since it completely eliminates many of the inconveniences and shortcomings that are experienced with REST APIs, such as over- and underfetching. Complexity is pushed to the server-side where powerful machines can take care of the heavy computation work. The client doesn't have to know where the data that it fetches is actually coming from and can use a single, coherent and flexible API.

Let's consider the major change that's introduced with GraphQL in going from a rather imperative data fetching approach to a purely declarative one. When fetching data from a REST API, most applications will have to go through the following steps:

construct and send HTTP request (e.g. with fetch in Javascript)

- receive and parse server response
- store data locally (either simply in memory or persistent)
- display data in the UI

With the ideal declarative data fetching approach, a client shouldn't be doing more than the following two steps:

- describe data requirements
- display data in UI

All the lower-level networking tasks as well as storing the data should be abstracted away and the declaration of data dependencies should be the dominant part.

This is precisely what GraphQL client libraries like Relay or Apollo will enable you to do. They provide the abstraction that you need to be able to focus on the important parts of your application rather than having to deal with the repetitive implementation of infrastructure.

Queries and Mutations

you'll learn in detail about how to query a GraphQL server.

Fields

At its simplest, GraphQL is about asking for specific fields on objects. Let's start by looking at a very simple query and the result we get when we run it:

```
{
  hero {
    name
    appearsIn
  }
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2"
    }

  }
```

- We start with a special "root" object
- We select the hero field on that

For the object returned by hero, we select the name and appearsIn fields

Because the shape of a GraphQL query closely matches the result, you can predict what the query will return without knowing that much about the server. But it's useful to have an exact description of the data we can ask for - what fields can we select? What kinds of objects might they return? What fields are available on those sub-objects? That's where the schema comes in.

Every GraphQL service defines a set of types which completely describe the set of possible data you can query on that service. Then, when queries come in, they are validated and executed against that schema.

Type language

GraphQL services can be written in any language. Since we can't rely on a specific programming language syntax, like JavaScript, to talk about GraphQL schemas, we'll define our own simple language. We'll use the "GraphQL schema language" - it's similar to the query language, and allows us to talk about GraphQL schemas in a language-agnostic way

Object types and fields #

The most basic components of a GraphQL schema are object types, which just represent a kind of object you can fetch from your service, and what fields it has. In the GraphQL schema language, we might represent it like this:

```
type Character {
  name: String!
  appearsIn: [Episode!]!
}
```

The language is pretty readable, but let's go over it so that we can have a shared vocabulary:

- Character is a GraphQL Object Type, meaning it's a type with some fields. Most of the types in your schema will be object types.
- name and appearsIn are fields on the Character type. That means that name and appearsIn are the only fields that can appear in any part of a GraphQL query that operates on the Character type.
- String is one of the built-in scalar types - these are types that resolve to a single scalar object, and can't have sub-selections in the query. We'll go over scalar types more later.
- String! means that the field is non-nullable, meaning that the GraphQL service promises to always give you a value when you query this field. In the type language, we'll represent those with an exclamation mark.
- [Episode!]! represents an array of Episode objects. Since it is also non-nullable, you can always expect an array (with zero or more items) when you query the appearsIn field. And since Episode! is also non-nullable, you can always expect every item of the array to be an Episode object.

Now you know what a GraphQL object type looks like, and how to read the basics of the GraphQL type language.

Arguments

Every field on a GraphQL object type can have zero or more arguments, for example the length field below:

```
type Starship {
  id: ID!
  name: String!
  length(unit: LengthUnit = METER): Float
}
```

All arguments are named. Unlike languages like JavaScript and Python where functions take a list of ordered arguments, all arguments in GraphQL are passed by name specifically. In this case, the length field has one defined argument, unit.

Arguments can be either required or optional. When an argument is optional, we can define a default value - if the unit argument is not passed, it will be set to METER by default.

The Query and Mutation types

Most types in your schema will just be normal object types, but there are two types that are special within a schema:

```
schema {
  query: Query
  mutation: Mutation
}
```

Every GraphQL service has a query type and may or may not have a mutation type. These types are the same as a regular object type, but they are special because they define the entry point of every GraphQL query. So if you see a query that looks like:

```
query {
  hero {
    name
  }
  droid(id: "2000") {
    name
  }

}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2"
    },
    "droid": {
      "name": "C-3PO"
    }
  }
```

That means that the GraphQL service needs to have a Query type with hero and droid fields:

```
type Query {
  hero(episode: Episode): Character
  droid(id: ID!): Droid
}
```

Mutations work in a similar way - you define fields on the Mutation type, and those are available as the root mutation fields you can call in your query.

It's important to remember that other than the special status of being the "entry point" into the schema, the Query and Mutation types are the same as any other GraphQL object type, and their fields work exactly the same way.

Scalar types

A GraphQL object type has a name and fields, but at some point those fields have to resolve to some concrete data. That's where the scalar types come in: they represent the leaves of the query.

In the following query, the name and appearsIn fields will resolve to scalar types:

```
{
  hero {
    name
    appearsIn
  }
}{
```

```
"appearsIn": [
      "NEWHOPE",
      "EMPIRE",
      "JEDI"
    ]
  }
}

}
```

We know this because those fields don't have any sub-fields - they are the leaves of the query.

GraphQL comes with a set of default scalar types out of the box:

- Int: A signed 32-bit integer.
- Float: A signed double-precision floating-point value.
- String: A UTF-8 character sequence.
- Boolean: true or false.
- ID: The ID scalar type represents a unique identifier, often used to refetch an object or as the key for a cache. The ID type is serialized in the same way as a String; however, defining it as an ID signifies that it is not intended to be human-readable.

In most GraphQL service implementations, there is also a way to specify custom scalar types. For example, we could define a Date type:

```
scalar Date
```

Then it's up to our implementation to define how that type should be serialized, deserialized, and validated. For example, you could specify that the Date type should always be serialized into an integer timestamp, and your client should know to expect that format for any date fields.

Enumeration types

Also called Enums, enumeration types are a special kind of scalar that is restricted to a particular set of allowed values. This allows you to:

- Validate that any arguments of this type are one of the allowed values
- Communicate through the type system that a field will always be one of a finite set of values
- Here's what an enum definition might look like in the GraphQL schema language:

```
enum Episode {
  NEWHOPE
  EMPIRE
  JEDI
}
```

This means that wherever we use the type Episode in our schema, we expect it to be exactly one of NEWHOPE, EMPIRE, or JEDI.

Note that GraphQL service implementations in various languages will have their own language-specific way to deal with enums. In languages that support enums as a first-class citizen, the implementation might take advantage of that; in a language like JavaScript with no enum support, these values might be internally mapped to a set of integers. However, these details don't leak out to the client, which can operate entirely in terms of the string names of the enum values.

Lists and Non-Null

Object types, scalars, and enums are the only kinds of types you can define in GraphQL. But when you use the types in other parts of the schema, or in your query variable declarations, you can apply additional type modifiers that affect validation of those values. Let's look at an example:

```
type Character {
  name: String!
  appearsIn: [Episode]!
}
```

Here, we're using a String type and marking it as Non-Null by adding an exclamation mark, ! after the type name. This means that our server always expects to return a non-null value for this field, and if it ends up getting a null value that will actually trigger a GraphQL execution error, letting the client know that something has gone wrong.

The Non-Null type modifier can also be used when defining arguments for a field, which will cause the GraphQL server to return a validation error if a null value is passed as that argument, whether in the GraphQL string or in the variables.

```
query DroidById($id: ID!) {
  droid(id: $id) {
    name
  }
}
{
  "id": null
}
{
```

```
  "errors": [
    {
      "message": "Variable \"$id\" of required
type \"ID!\" was not provided.",
      "locations": [
        {
          "line": 1,
          "column": 17
        }
      ]
```

Lists work in a similar way: We can use a type modifier to mark a type as a List, which indicates that this field will return an array of that type. In the schema language, this is denoted by wrapping the

type in square brackets, [and]. It works the same for arguments, where the validation step will expect an array for that value.

The Non-Null and List modifiers can be combined. For example, you can have a List of Non-Null Strings:

```
myField: [String!]
```

This means that the list itself can be null, but it can't have any null members. For example, in JSON:

```
myField: null // valid
myField: [] // valid
myField: ['a', 'b'] // valid
myField: ['a', null, 'b'] // error
```

Now, let's say we defined a Non-Null List of Strings:

```
myField: [String]!
```

This means that the list itself cannot be null, but it can contain null values:

```
myField: null // error
myField: [] // valid
myField: ['a', 'b'] // valid
myField: ['a', null, 'b'] // valid
```

You can arbitrarily nest any number of Non-Null and List modifiers, according to your needs.

Interfaces

Like many type systems, GraphQL supports interfaces. An Interface is an abstract type that includes a certain set of fields that a type must include to implement the interface.

For example, you could have an interface Character that represents any character in the Star Wars trilogy:

```
interface Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
}
```

This means that any type that implements Character needs to have these exact fields, with these arguments and return types.

For example, here are some types that might implement Character:

```
type Human implements Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  starships: [Starship]
  totalCredits: Int
}
type Droid implements Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  primaryFunction: String
```

```
}
```

You can see that both of these types have all of the fields from the Character interface, but also bring in extra fields, totalCredits, starships and primaryFunction, that are specific to that particular type of character.

Interfaces are useful when you want to return an object or set of objects, but those might be of several different types.

For example, note that the following query produces an error:

```
query HeroForEpisode($ep: Episode!) {
  hero(episode: $ep) {
    name
    primaryFunction
  }
}
{
  "ep": "JEDI"
}
```

```
{
  "errors": [
    {
      "message": "Cannot query field \"primaryFunction\" on type \"Character\". Did you mean to use an inline fragment on \"Droid\"?",
      "locations": [
        {
          "line": 4,
          "column": 5
        }
```

The hero field returns the type Character, which means it might be either a Human or a Droid depending on the episode argument. In the query above, you can only ask for fields that exist on the Character interface, which doesn't include primaryFunction.

To ask for a field on a specific object type, you need to use an inline fragment:

```
query HeroForEpisode($ep: Episode!) {
  hero(episode: $ep) {
    name
    ... on Droid {
      primaryFunction
    }
  }
}
{
  "ep": "JEDI"
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "primaryFunction": "Astromech"
    }
  }
}
```

Learn more about this in the inline fragments section in the query guide.

Union types

Union types are very similar to interfaces, but they don't get to specify any common fields between the types.

```
union SearchResult = Human | Droid | Starship
```

Wherever we return a SearchResult type in our schema, we might get a Human, a Droid, or a Starship. Note that members of a union type need to be concrete object types; you can't create a union type out of interfaces or other unions.

In this case, if you query a field that returns the SearchResult union type, you need to use a conditional fragment to be able to query any fields at all:

```
{
  search(text: "an") {
    __typename
    ... on Human {
      name
      height
    }
    ... on Droid {
      name
      primaryFunction
    }
    ... on Starship {
      name
      length
    }
  }
}
{
```

```
  "data": {
    "search": [
      {
        "__typename": "Human",
        "name": "Han Solo",
        "height": 1.8
      },
      {
        "__typename": "Human",
        "name": "Leia Organa",
        "height": 1.5
      },
      {
        "__typename": "Starship",
        "name": "TIE Advanced x1",
        "length": 9.2
      }
    ]
```

The __typename field resolves to a String which lets you differentiate different data types from each other on the client.

Also, in this case, since Human and Droid share a common interface (Character), you can query their common fields in one place rather than having to repeat the same fields across multiple types:

```
{
  search(text: "an") {
    __typename
    ... on Character {
      name
    }
    ... on Human {
      height
    }
    ... on Droid {
      primaryFunction
    }
    ... on Starship {
      name
      length
    }
  }
}
```

Note that name is still specified on Starship because otherwise it wouldn't show up in the results given that Starship is not a Character!

Input types

So far, we've only talked about passing scalar values, like enums or strings, as arguments into a field. But you can also easily pass complex objects. This is particularly valuable in the case of mutations, where you might want to pass in a whole object to be created. In the GraphQL schema language, input types look exactly the same as regular object types, but with the keyword input instead of type:

```
input ReviewInput {

  stars: Int!

  commentary: String

}
```

Here is how you could use the input object type in a mutation:

```
mutation CreateReviewForEpisode($ep: Episode!, $review: ReviewInput!) {
  createReview(episode: $ep, review: $review) {
    stars
    commentary
  }
}
```

```
{
  "ep": "JEDI",

  "review": {
    "stars": 5,
    "commentary": "This is a great movie!"
  }
}
```

```
 "data": {
    "createReview": {
      "stars": 5,
      "commentary": "This is a great movie!"
    }
  }
}
```

The fields on an input object type can themselves refer to input object types, but you can't mix input and output types in your schema. Input object types also can't have arguments on their fields.

Set Up Apollo Server

A schema is only useful if our data graph conforms to the schema's structure. Enforcing a schema's structure is one of the core features of Apollo Server, a production-ready, open-source library that helps you implement your data graph's API.

- From the start/server directory, let's install Apollo Server (along with our project's other dependencies)

```
cd start/server && npm install
```

The two packages you need to get started with Apollo Server are apollo-server and graphql, both of which are installed with the above command.

Now, let's navigate to src/index.js so we can create our server. Paste the code below into the file:

src/index.js

```
const { ApolloServer } = require('apollo-server');
const typeDefs = require('./schema');

const server = new ApolloServer({ typeDefs });
```

This code imports the ApolloServer class from apollo-server, along with our (currently undefined) schema from src/schema.js. It then creates a new instance of ApolloServer and passes it the imported schema via the typeDefs property.

Now that Apollo Server is prepared to receive our schema, let's define it.

Your GraphQL schema defines what types of data a client can read and write to your data graph. Schemas are strongly typed, which unlocks powerful developer tooling.

Define Your Schema Types

Your schema's structure should support the actions that your clients will take. Our example app needs to be able to:

- Fetch a list of all upcoming rocket launches
- Fetch a specific launch by its ID
- Log in the user
- Book a launch for a logged-in user
- Cancel a previously booked launch for a logged-in user
- Let's design our schema to make these actions straightforward.
- In src/schema.js, import gql from apollo-server and create a variable called typeDefs for your schema

src/schema.js

```
const { gql } = require('apollo-server');

const typeDefs = gql`
  # Your schema will go here
`;

module.exports = typeDefs;
```

The schema will go inside the gql function (between the backticks). The language we'll use to write the schema is GraphQL's schema definition language (SDL).

Object Types

Most of the definitions in a GraphQL schema are object types. Each object type you define should represent an object that an application client might need to interact with.

For example, our example app needs to be able to fetch a list of upcoming rocket launches, so we should define a Launch type to support that behavior.

Paste the following inside the backticks of the typeDefs declaration in src/schema.js:

src/schema.js

```
type Launch {
  id: ID!
  site: String
  mission: Mission
  rocket: Rocket
  isBooked: Boolean!
}
```

The Launch object type has a collection of fields, and each field has a type of its own. A field's type can be either an object type or a scalar type. A scalar type is a primitive (like ID, String, Boolean, or Int) that resolves to a single value. In addition to GraphQL's built-in scalar types, you can define custom scalar types.

- In the Launch definition above, Mission and Rocket refer to other object types. Let's add definitions for those, along with the User type (again, all inside the backticks):

```
src/schema.js

type Rocket {
  id: ID!
  name: String
  type: String
}

type User {
  id: ID!
  email: String!
  trips: [Launch]!
}

type Mission {
  name: String

  missionPatch(size: PatchSize): String

}
```

The Query Type

We've defined the objects that exist in our data graph, but clients don't yet have a way to fetch those objects. To resolve that, our schema needs to define queries that clients can execute against the data graph.

You define your data graph's supported queries as fields of a special type called the Query type. Paste the following into your schema definition:

```
src/schema.js
type Query {
  launches: [Launch]!
  launch(id: ID!): Launch
  me: User
```

This Query type defines three available queries for clients to execute: launches, launch, and me.

- The launches query will return an array of all upcoming Launches.

- The launch query will return a single Launch that corresponds to the id argument provided to the query.
- The me query will return details for the User that's currently logged in.

The Mutation type

Queries enable clients to fetch data, but not to modify data. To enable clients to modify data, our schema needs to define some mutations.

The Mutation type is a special type that's similar in structure to the Query type. Paste the following

```
src/schema.js

type Mutation {

  bookTrips(launchIds: [ID]!): TripUpdateResponse!

  cancelTrip(launchId: ID!): TripUpdateResponse!

  login(email: String): String #
```

This Mutation type defines three available mutations for clients to execute: bookTrips, cancelTrip, and login.

The bookTrips mutation enables a logged-in user to book a trip on one or more Launches (specified by an array of launch IDs).

The cancelTrip mutation enables a logged-in user to cancel a trip that they previously booked.

The login mutation enables a user to log in by providing their email address.

The bookTrips and cancelTrip mutations return the same object type: a TripUpdateResponse. A mutation's return type is entirely up to you, but we recommend defining special object types specifically for mutation responses.

- Add the definition for TripUpdateResponse to your schema:

```
src/schema.js
type TripUpdateResponse {
  success: Boolean!
  message: String
  launches: [Launch]
}
```

This response type contains a success status, a corresponding message, and an array of any Launches that were modified by the mutation. It's good practice for a mutation to return whatever objects it modifies so the requesting client can update its cache and UI without needing to make a followup query.

Our example app's schema is now complete!

Run your server

Return to src/index.js and add a call to server.listen(), like so:

src/index.js

```
const { ApolloServer } = require('apollo-server');
const typeDefs = require('./schema');

const server = new ApolloServer({ typeDefs });

server.listen().then(({ url }) => {
  console.log(`  Server ready at ${url}`);
```

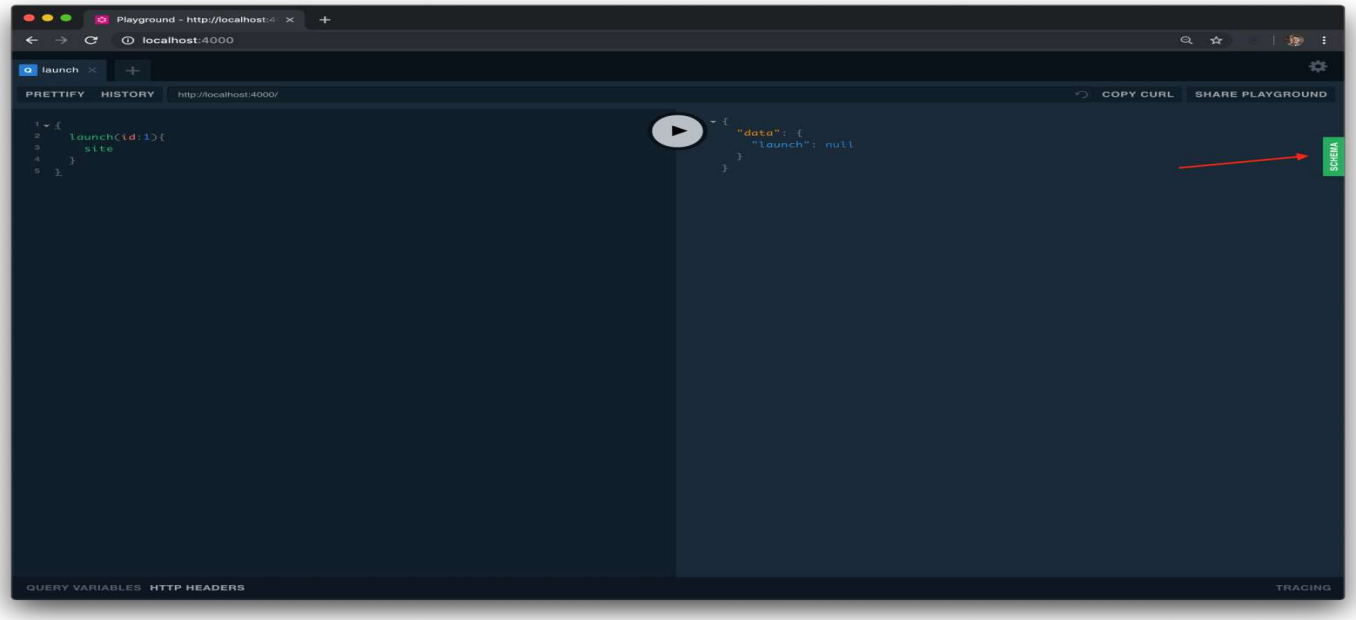
After saving, run npm start to start your server! Apollo Server will now be available on port 4000.

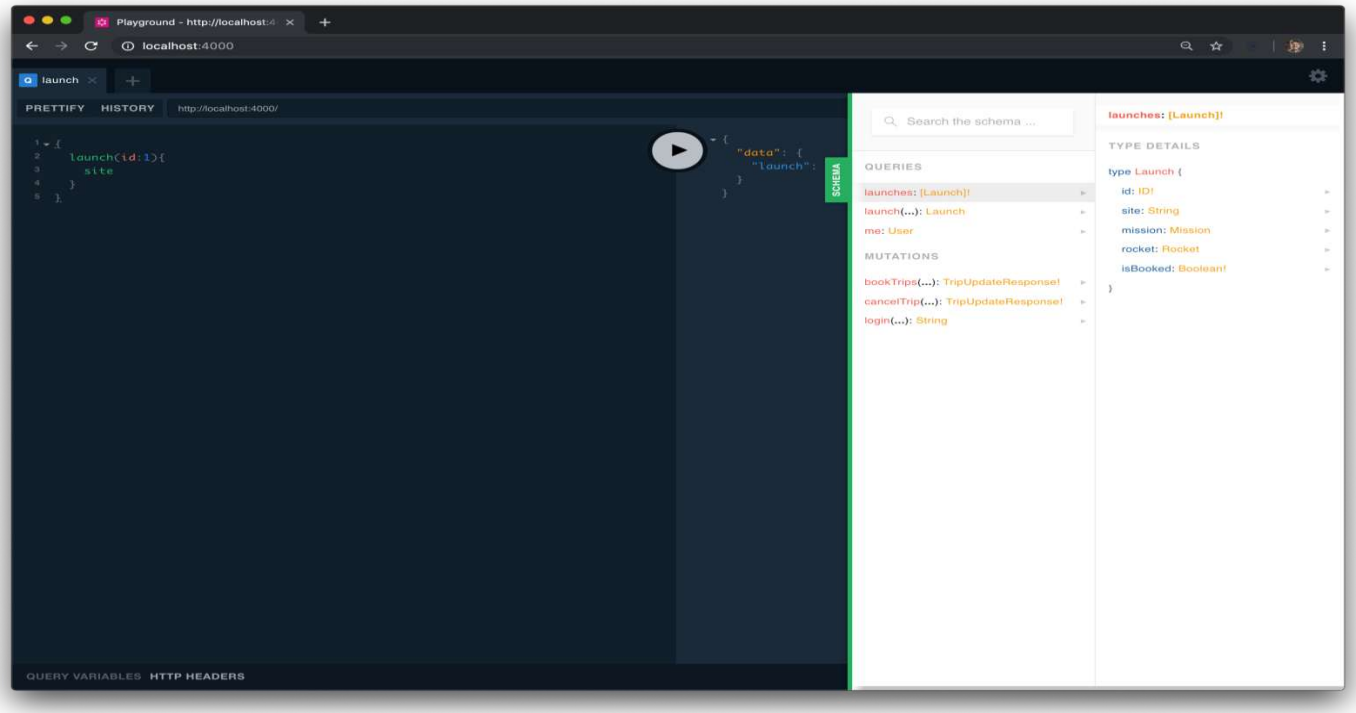
Explore your schema

With your server running, visit localhost:4000 in your browser to open GraphQL Playground, an IDE that enables you to introspect your schema and test out queries. Introspection is a helpful feature of a GraphQL server that enables you to obtain its schema.

Note that introspection should be disabled for a production GraphQL server. Apollo Server disables introspection automatically if the NODE_ENV environment variable is set to production.

To introspect your server's schema, click the Schema button on the right side of GraphQL Playground:


Playground - http://localhost:4
localhost:4000
launch
PRETTIFY
HISTORY
http://localhost:4000/
COPY CURL
SHARE PLAYGROUND
{
launch(id:1){
site
}
}
{
"data": {
"launch": null
}
}
SCHEMA
QUERY VARIABLES
HTTP HEADERS
TRACING

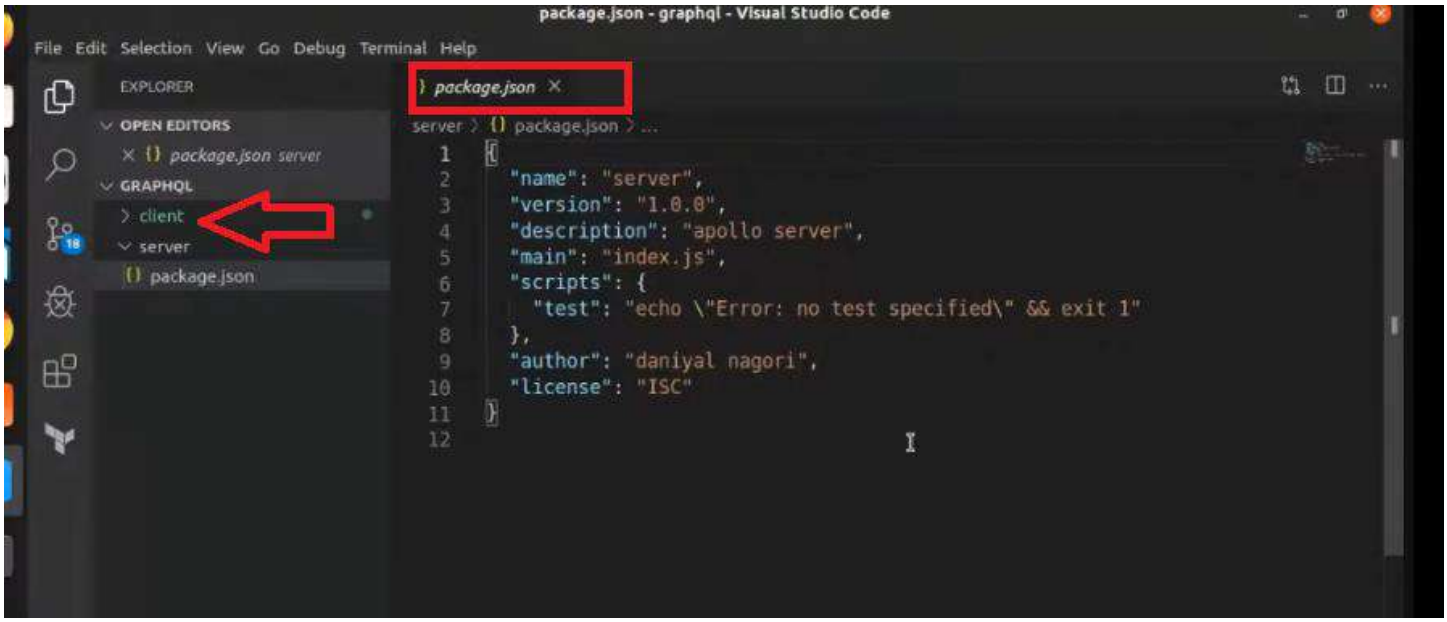

Playground - http://localhost:4
localhost:4000
launch
PRETTIFY
HISTORY
http://localhost:4000/
{
launch(id:1){
site
}
}
{
"data": {
"launch":
}
}
SCHEMA
Search the schema ...
QUERIES
launches: [Launch]!
launch(...): Launch
me: User
MUTATIONS
bookTrips(...): TripUpdateResponse!
cancelTrip(...): TripUpdateResponse!
login(...): String
launches: [Launch]!
TYPE DETAILS
type Launch {
id: ID!
site: String
mission: Mission
rocket: Rocket
isBooked: Boolean!
}
QUERY VARIABLES
HTTP HEADERS

Session 16(A): GraphQL in Detail

Queries And Mutation: Apollo Server & Apollo Client

1. **Apollo Server**

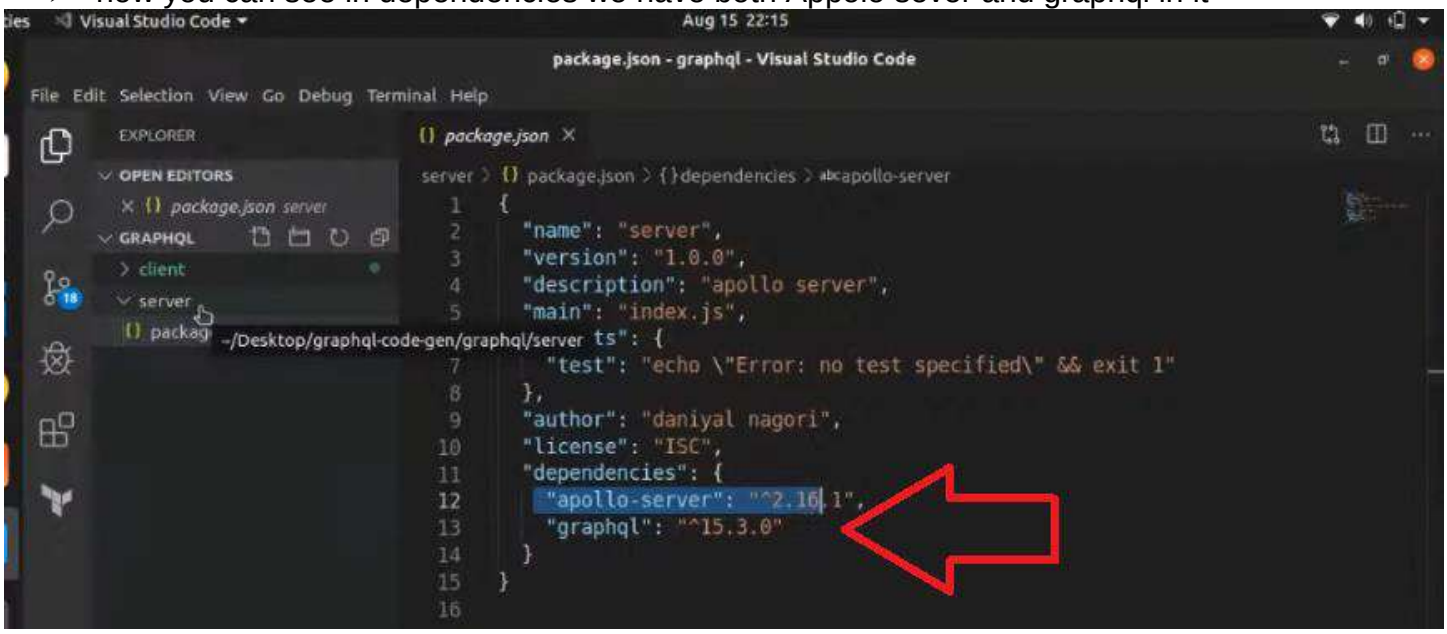
- Create a new project
- From your preferred development directory, create a directory for a new project and cd into it:
- Initialize a new Node.js project with npm (or another package manager you prefer, such as Yarn):.
- create a folder of graphql server as well as a client
- from npm server create the structure of the project
- Your project directory now contains a package.json file.



- install Appolo server with yarn or npm with a simple command

  npm install apollo-server graphql

- now you can see in dependencies we have both Appolo sever and graphql in it



- create index.js file for the server

```
let {AppoloServer}=require('apollo-server')
let typeDefs=require(./Schema/schema)
let resolvers=require('./Resolvers')
let
ApolloServer{nemAppoloServer({typeDefs,resolvers})
apolloServer.listen().then(({url}))=>
   console.log(``server is running on port ${url}`
```

- create schema folder
- create schema
- create a query type
- to get schema we create the resolver with the same name as in the query

```
const{gql}=require('apollo-server')
const typeDef=gql`
type CreateMessageType{
   message:String
}
 type Mutation{
    createMessage:string
}
```

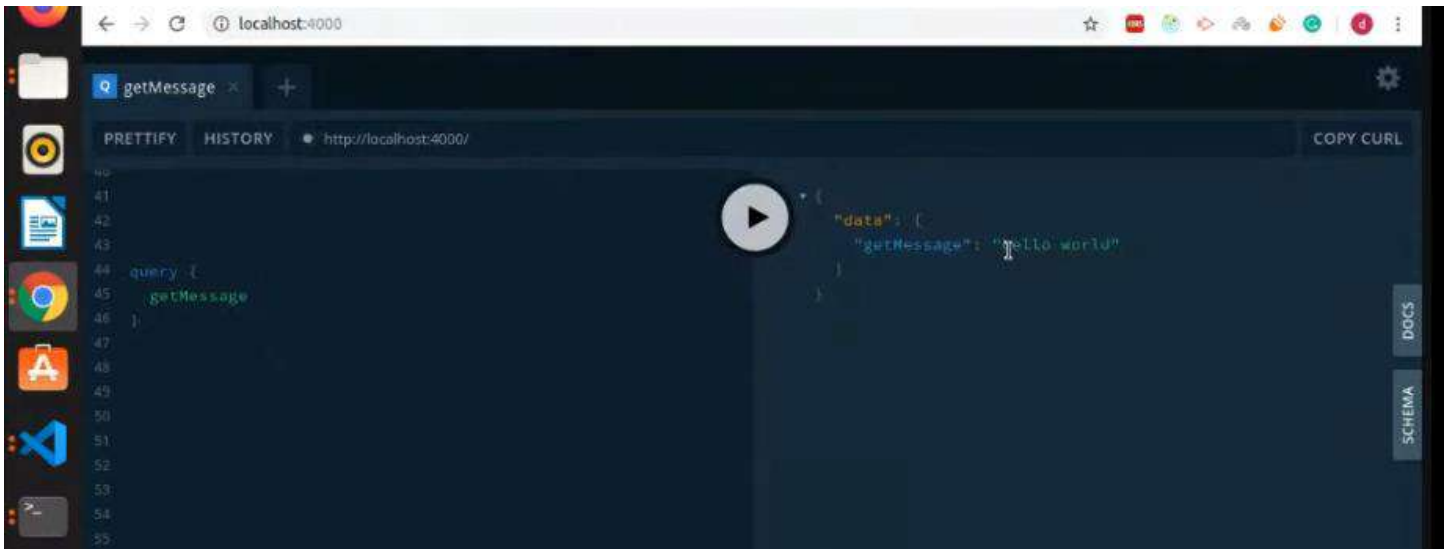
type Query

```
    getMessage:String

     getName:string

  }

  module.exports=resolve
```

- creaet a folder of resolver

```
  const resolver={
  Query:{
   getMessage()=>
     return"hello world"
    }
  }

  Mutation:{

  }
```

- in the packet, Jason script write server: "node index.js",
- run this in terminal and checkout in the browser
- write a message in the query and play it we will get Hello world as shown



- in this way can get mutation and all on this polo server
- to make any change, we need to restart the server and then check it in the browser
  - install nodemon and run the file on it to avoid refreshing the system again and again
- now similarly if we want to get a name then we can return it as in the resolver

```
getName: ()=>{
    return "Danyal Nagori" }
```

- now go to the schema and add mutation
- create message input

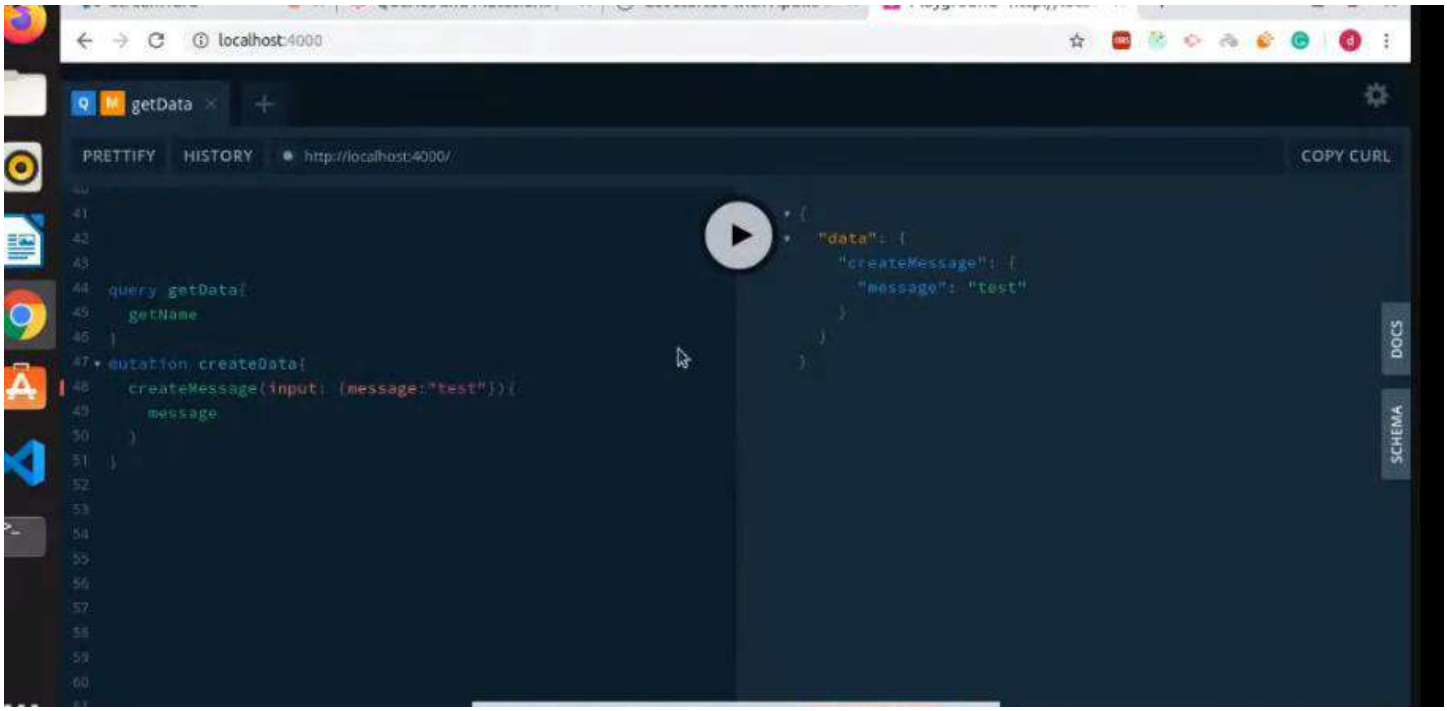
Schema.js

```
const{gql}=require('apollo-server')
const typeDef=gql`
type messageType{
   message:String
}
input messageInput{
  message:String

 type Mutation{
    createMessage:(input:messageInput):messageType
}
  type Query
   getMessage:String
    getName:string
}
module.exports=resolvers
```

- describe local data
- create an object of createMessage
- Now go to resolver.js and follow up the following code with mutation
- Provide any random id

```
let crypto=require('crypto')
const resolver={
Query:{
 getMessage()=>
    return"danyal nagori"
  }
}

Mutation:{
createMessaeg:(_{nput})=>{
let id =crypto.randomBytes(10).toString()
 // let  id=123
console.log('id',id)
localData{id}={id…input.message}
return  {id…input}
  }
```

- now run this and create data and play in the browser



2. <u>Apollo Client</u>

- now run client-side

```
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC: ~/Desktop/graphql-code-ge...   daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC: ~/Desktop/graphql-code-ge...
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/graphql-code-gen/graphql/server$ cd ..
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/graphql-code-gen/graphql$ cd client/
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/graphql-code-gen/graphql/client$ yarn start
yarn run v1.22.0
$ react-scripts start
```

- install the package we need as

npm install @apollo/client graphql

```
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/graphql-code-gen/graphql
$ cd client/
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/graphql-code-gen/graphql
/client$ sudo npm install @apollo/client graphql
[sudo] password for daniyal:
                    ] / rollbackFailedOptional: verb npm-session c83f78ce3e580cd
```

- go to the client folder index.js

```
import {ApIloClient,InMemoryCache} from '@apollo/client
const graphQLClient=new ApolloClient({
      uri: "http://localhost:400/",
      cache: new InMemoryCache()
})
export default graphgQLClient;
```

- import apollo client in index.js
- it will take a few minutes to install the apollo client
- import apollo provider from apollo client
- import gqlConfig

```
Resolvers    JS index.js .../Config    JS index.js .../src ×
client > src > JS index.js
1   import React from 'react';
2   import ReactDOM from 'react-dom';
3   import './index.css';
4   import App from './App';
5   import * as serviceWorker from './serviceWor
6   import { ApolloProvider } from '@apollo/clie
7   import gqlConfig from "./Config"
8   ReactDOM.render(
9     <React.StrictMode>
10      <ApolloProvider client={gqlConfig}>
11        <App />
12      </ApolloProvider>
13    </React.StrictMode>,
14    document.getElementById('root')
15  );
```

- Now that we have all the dependencies we need, let's initialize an ApolloClient instance. You'll need to provide it the URL of a running GraphQL server, such as this CodeSandbox instance.

- In index.js, let's import ApolloClient from @apollo/client and provide our GraphQL server's URL as the uri property of the constructor's configuration object
- create a folder of gqlSchema and index.js

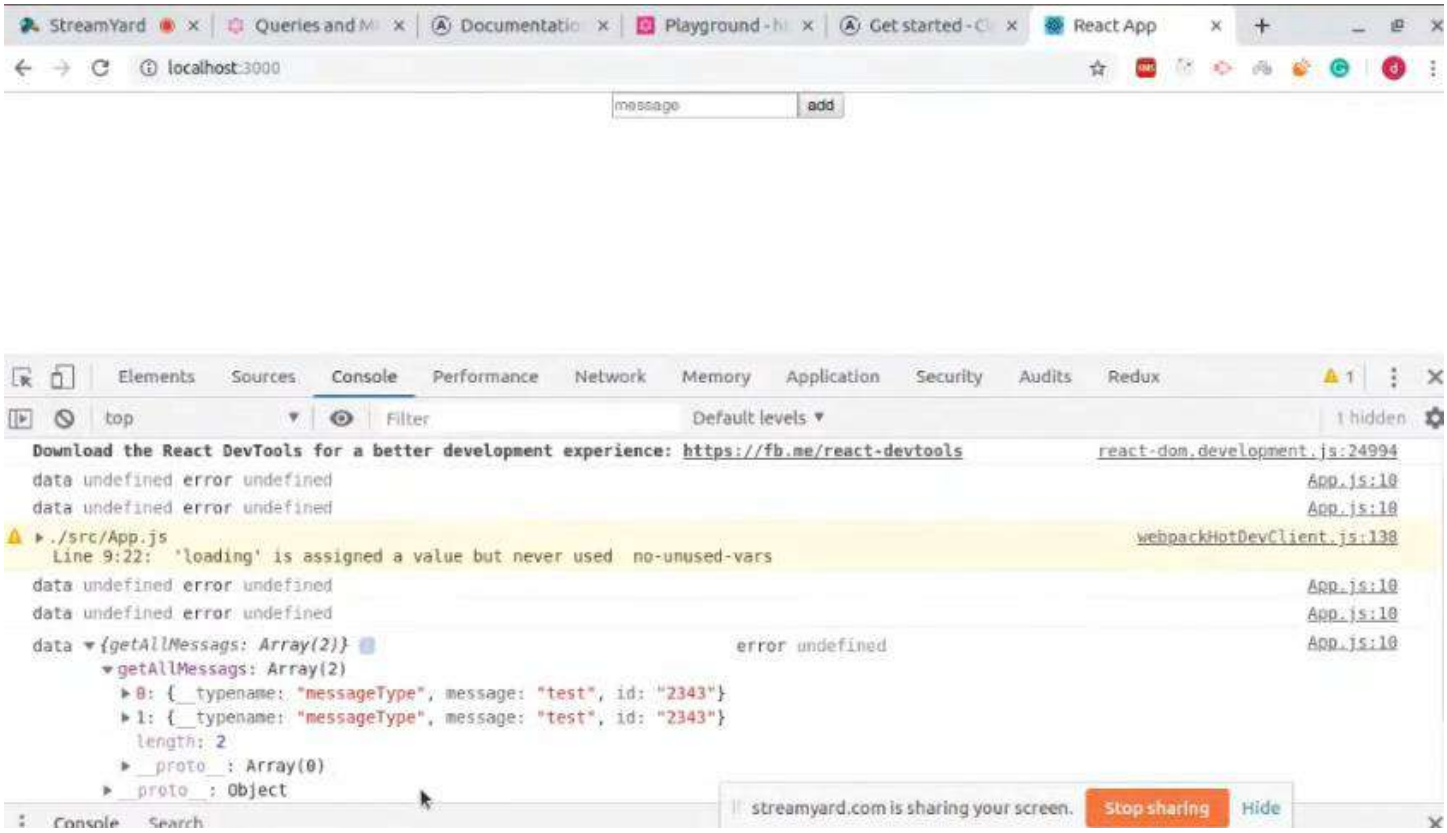
```
import{gql} from "@apollo/client"
const creaetMessageMutation=gql`

mutation createMessage($data:messageInput)[
creaetMessage(input:$data){
message
id
}
}
const getALLMessages=gql;
query getAllMessages{
   getAllMessages{
  message
  id
}
export{
createMessageMutation,
getAllMessages
```

App.js

```
import React ,{createRef} from 'react';
import './App.css;
import {useMuttion} from "@apollo/client";
import {createMessageMutation} from "./gql";
function App(){
const messageValue=createRef()
const [addMessage]=useMutation(createMessageMutation)
let{data,error,loading}=useQuery(getAllMessages)
const submit+(=>{
let message=messageValue.current.value;
addMessage({variable:{data:{message}}
.then(data=>{
console.log('response',data)
}).catch("error",error)
})
```

```
eturn(
div className="App">
input ref={messageValue}palceholder=
button onClick={submit}>add</button>
/div>

xport default App;
```

- run react application on localhost 4000 and connect with the console. log



Session 17: Building_SpaceX_App_with_QraphQL

Building Space X App By Using React, TypeScript, GraphQL

- open up terminal
- build a repository from your GitHub folder and create a react app Space --typescript

```
Adil:GitHub adil$ npx create-react-app spacex --typescript
npx: installed 98 in 15.008s

Creating a new React app in /Users/adil/Documents/GitHub/spacex.

The --typescript option has been deprecated and will be removed in a future rel
ease.
In future, please use --template typescript.

warning ../package.json: No license field
Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template-typescript...

yarn add v1.19.1
warning ../package.json: No license field
[1/4] 🔍  Resolving packages...
[2/4] 🚚  Fetching packages...
[3/4] 🔗  Linking dependencies...
warning "react-scripts > @typescript-eslint/eslint-plugin > tsutils@3.17.1" has
 unmet peer dependency "typescript@>=2.8.0 || >= 3.2.0-dev || >= 3.3.0-dev || >
= 3.4.0-dev || >= 3.5.0-dev || >= 3.6.0-dev || >= 3.6.0-beta || >= 3.7.0-dev ||
 >= 3.7.0-beta".
[------------------------------------------------------------------------] 0/1487_
```

- install graphql
- add graphql codegen/cli (all objects are generated)

```
Adil:spacex adil$ yarn add -D @graphql-codegen/cli
yarn add v1.19.1
warning ../package.json: No license field
[1/4] 🔍  Resolving packages...
warning @graphql-codegen/cli > request@2.88.2: request has been deprecated, see
 https://github.com/request/request/issues/3142
. de-indent@^1.0.2_
```

- now add yarn grapghql-codegen init
- the code generator will ask you several questions, answer them accordingly

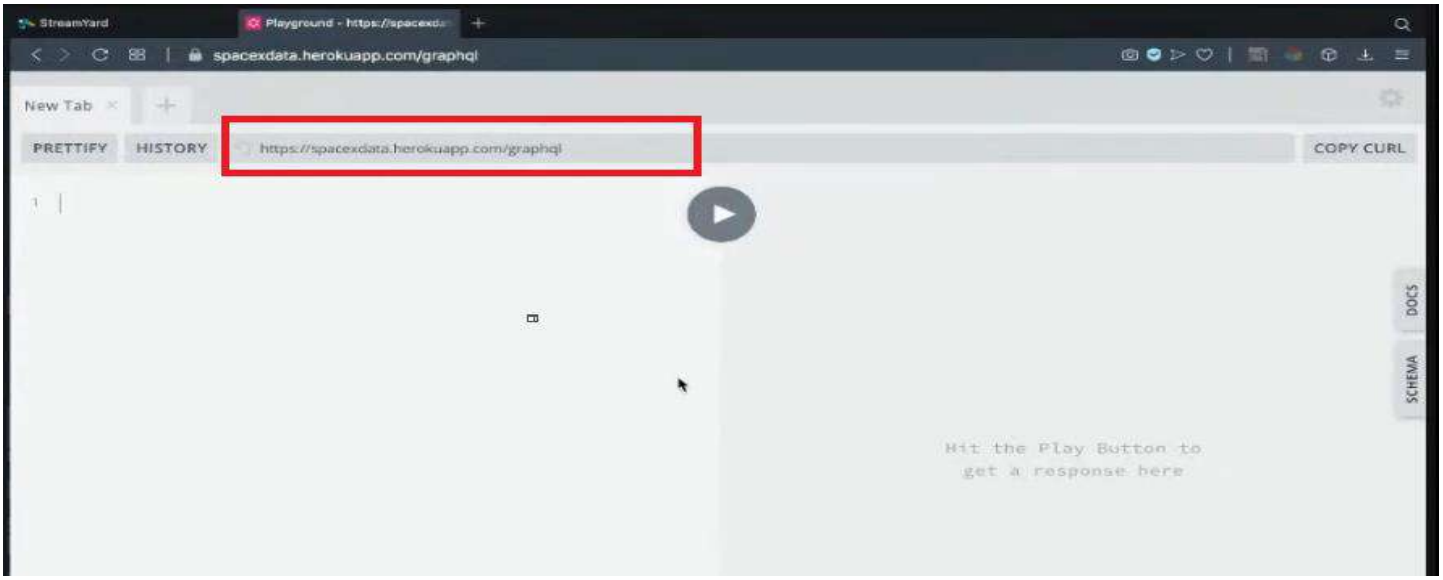
```
Adil:spacex adil$ yarn graphql-codegen init
yarn run v1.19.1
warning ../package.json: No license field
$ /Users/adil/Documents/GitHub/spacex/node_modules/.bin/graphql-codegen init

    Welcome to GraphQL Code Generator!
    Answer few questions and we will setup everything for you.

? What type of application are you building? Application built with React
? Where is your schema?: (path or url) https://spacexdata.herokuapp.com/graphql


? Where are your operations and fragments?: ./src/components/**/*.{ts,tsx}
? Pick plugins: TypeScript (required by other typescript plugins), TypeScript O
perations (operations and fragments), TypeScript React Apollo (typed components
 and HOCs)
? Where to write the output: src/generated/graphql.tsx
? Do you want to generate an introspection file? No
? How to name the config file? codegen.yml
? What script in package.json should run the codegen? codegen_
```

- open up your project
- look at codegen.yml add config withHooks as true

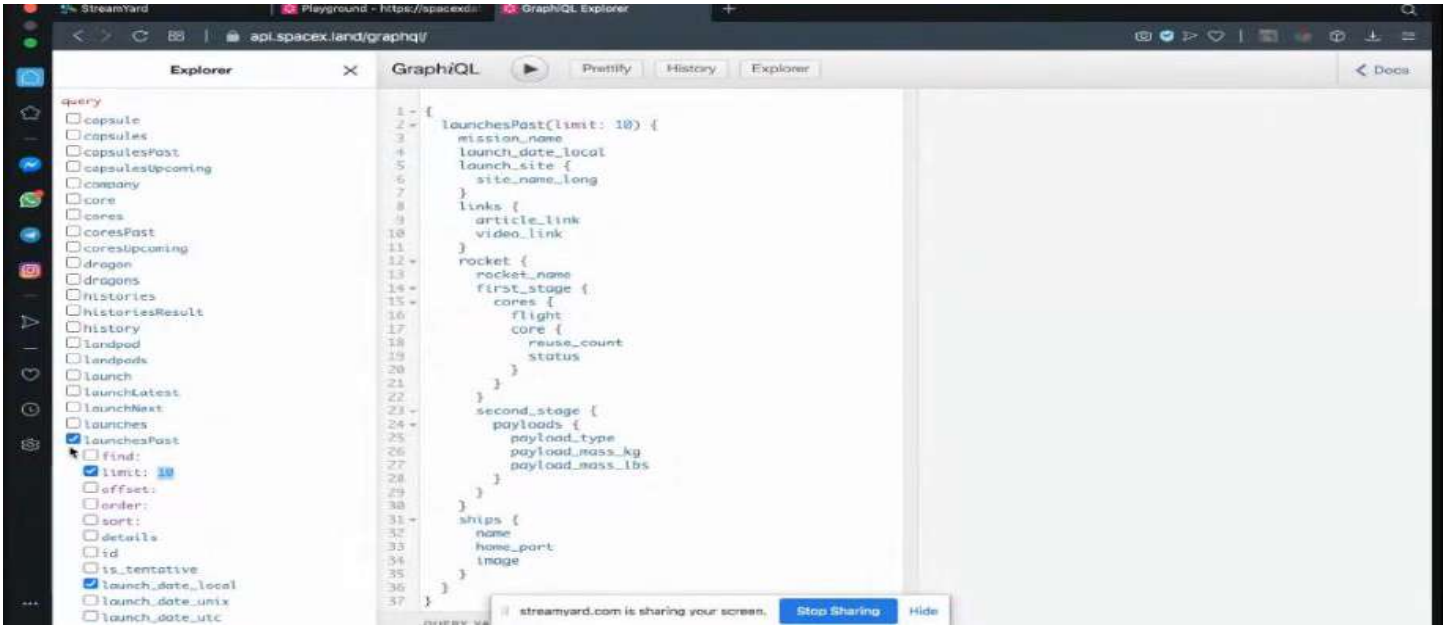
```
EXPLORER
> OPEN EDITORS
> SPACEX
  > node_modules
  > public
  > src
    .gitignore
    codegen.yml                U
    package.json               M
    README.md
    tsconfig.json
    yarn.lock                  M

codegen.yml > {} generates > {} src/generated/graphql.tsx > config > withHooks
1  overwrite: true
2  schema: "https://spacexdata.herokuapp.com/graphql"
3  documents: "./src/components/**/*.{ts,tsx}"
4  generates:
5    src/generated/graphql.tsx:
6      plugins:
7        - "typescript"
8        - "typescript-operations"
9        - "typescript-react-apollo"
10     config:
11       withHooks: true
```

- go to the browser and see the actual specexdata.herokuapp.com/graphql

this will give us access to the playground where we can create graphql queries and run them against SpaceX

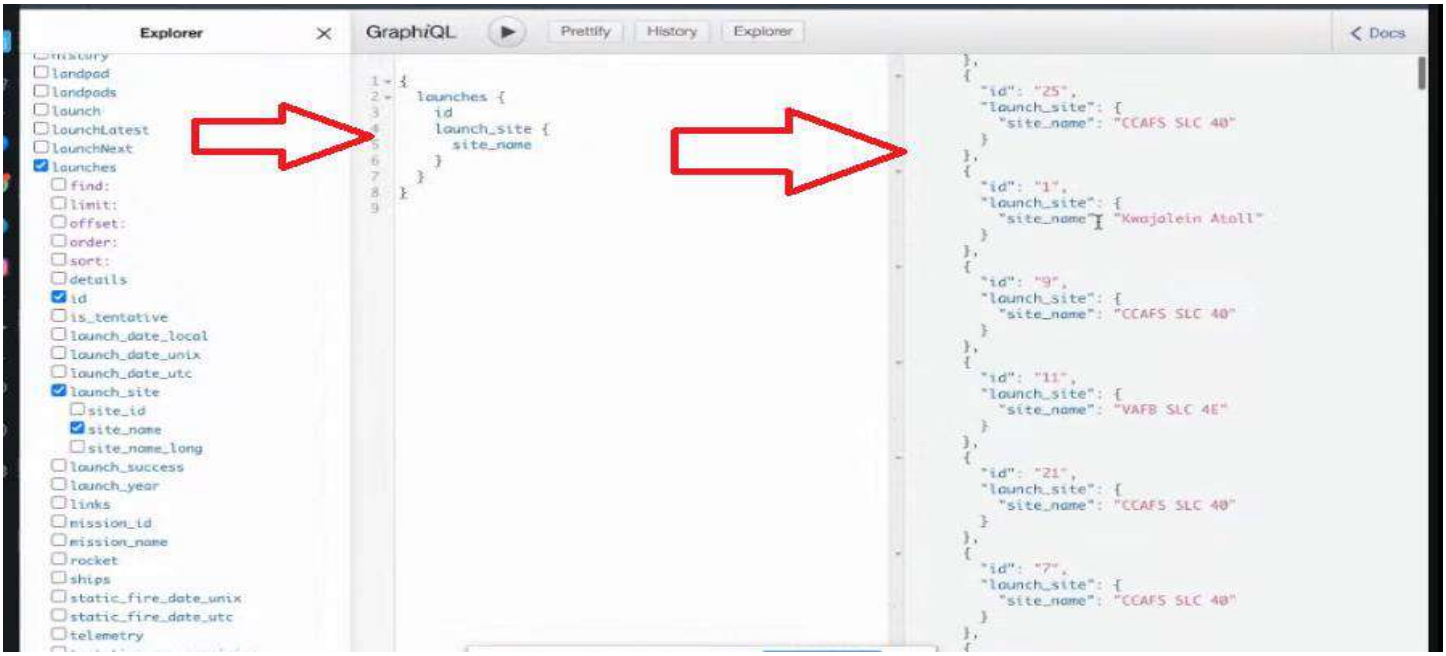


- now look at the actual call of the data on API.spacex,land/graphql/
- it will have all schemas as shown

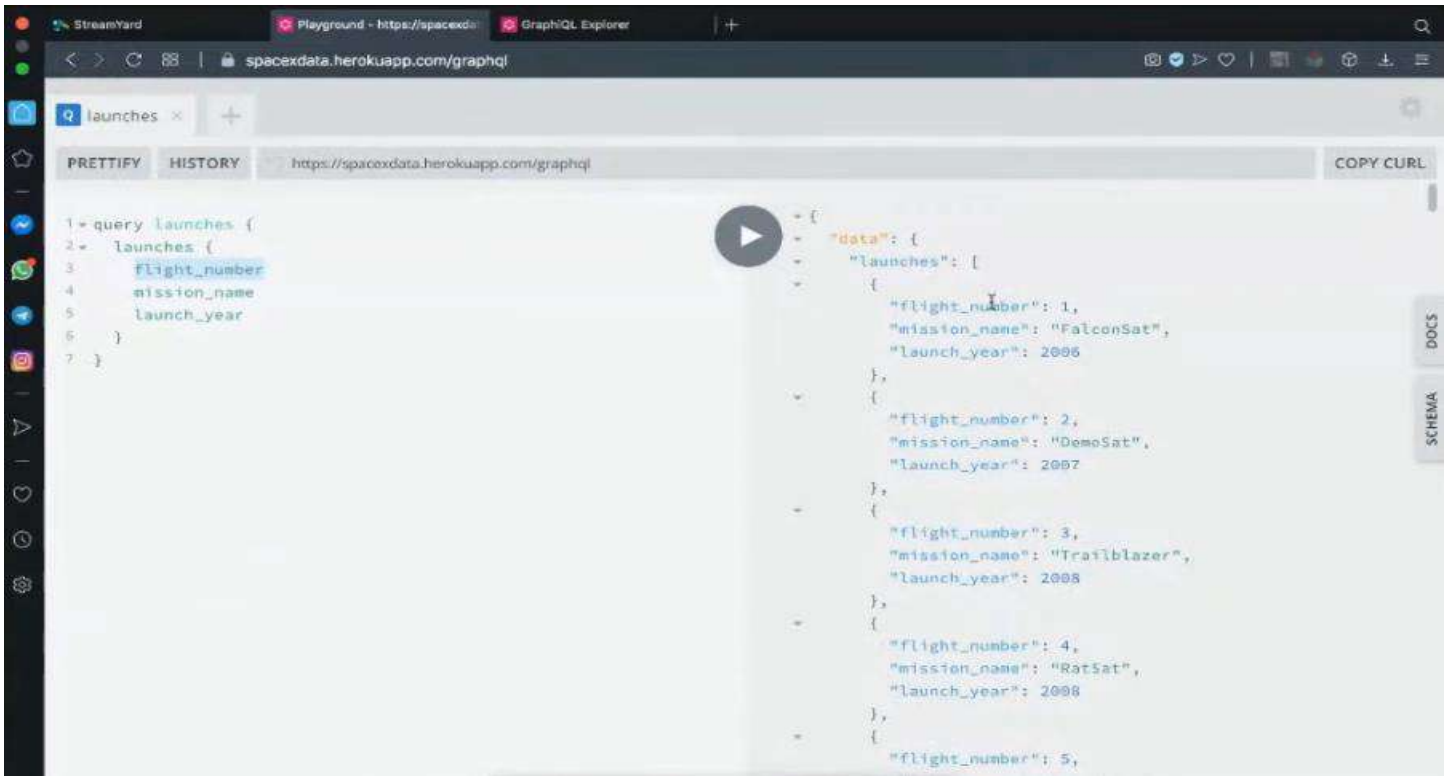


- you can get data of launches, id, launch site, site name details just by click to the explorer menus

lunch success ., mission name, and all with the help of graphql explorer



- Similar data you can get by writing the series in the playground and then running them As



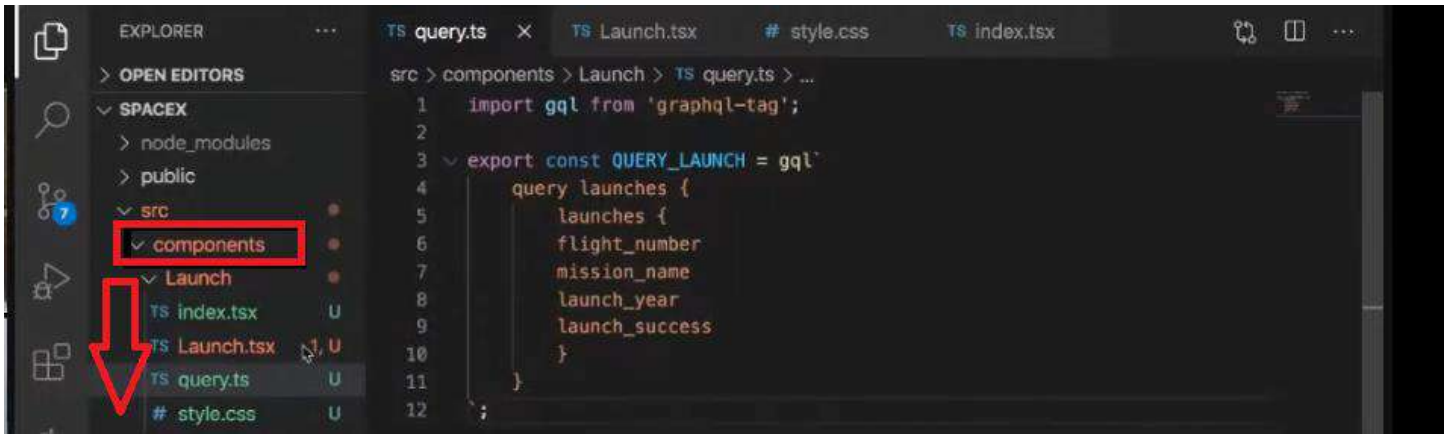
- go to the project to build up components
- create a folder for component
  - launch
  - details
- inside the launch, the folder creates some file

- launch.tsx
- style.css
- index.tsx
- query.ts

➢ go to the query.ts
➢ import gql from 'graphql-tag';
➢ export const query launch

query.ts

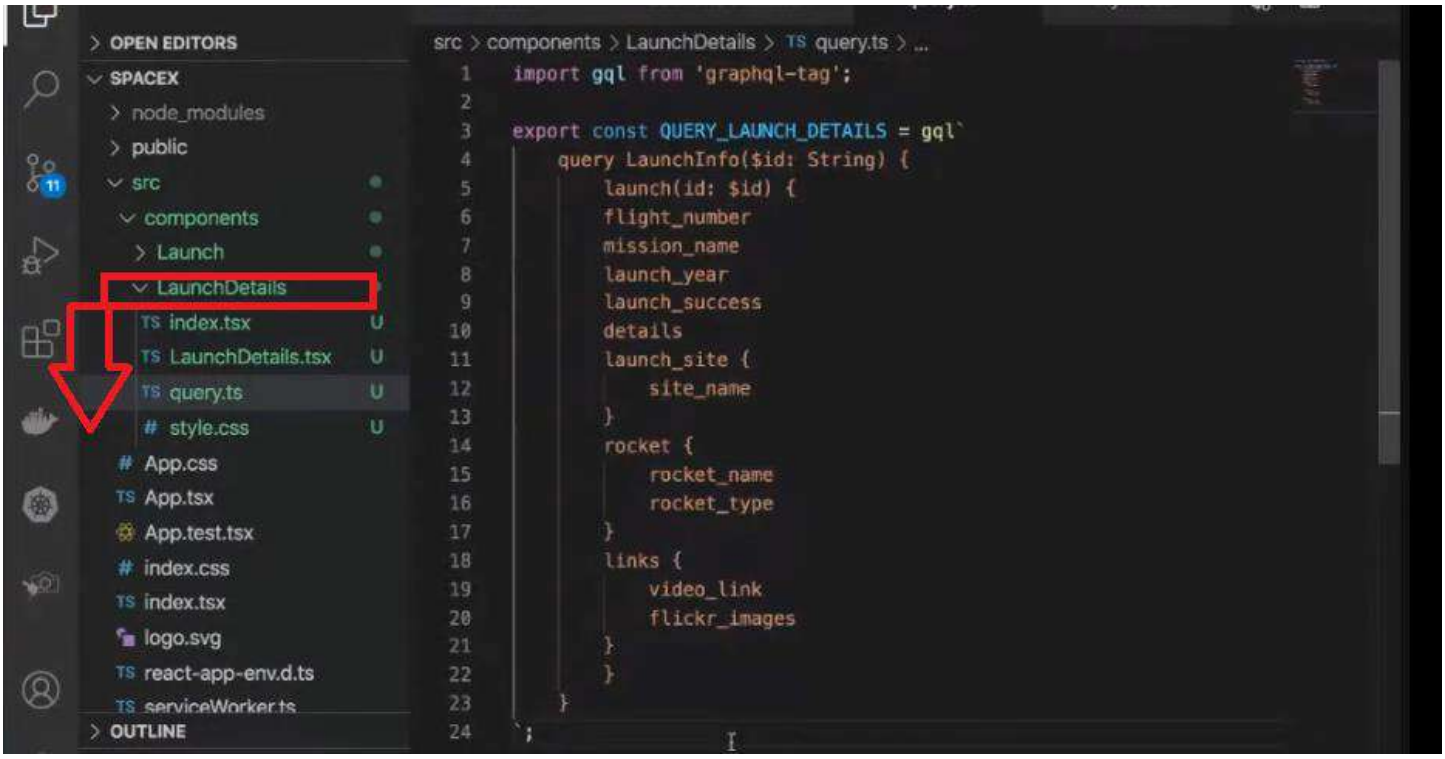
```
import gql from'graphql-tag';

export const QUERY-LUANCH=gql`
  query launches{
   flight-number
   mission name
   launch-year
   launch success
      }
}
;
```



- create second query for lunchdetails create all files as for first query

query.ts

```
import gql from 'graphql-tag';

export const QUERY_LAUNCH_DEDTAILS=gql`;
query LaunchInfo($id:String)[
flight-number
mission-name
launch-year
lauch-success
details
luanch-site{
  site-name
}
rocket{
  rocket-name
  rocket-type
}
links{
  video-link
  flickr-images
  }
 }
}
```



- generate typescript

```
? Please choose a version of "@graphql-codegen/typescript-react-apollo" from th
is list: 2.0.7-alpha-67af76ba.0
[2/4] 🚚  Fetching packages...
[3/4] 🔗  Linking dependencies...
warning " > @testing-library/user-event@7.2.1" has unmet peer dependency "@test
ing-library/dom@>=5".
warning " > @apollo/client@3.1.3" has unmet peer dependency "subscriptions-tran
sport-ws@^0.9.0".
[------------------------------------------------------------------------] 0/1993_
```

- add yarn codegen
- click to load graphgql schemas from generate src/generated/graphql.tsx
- this will generate the output that we can see in our project

```
Adil:spacex adil$ yarn codegen
yarn run v1.19.1
warning ../package.json: No license field
$ graphql-codegen --config codegen.yml
  ✔ Parse configuration
  ❯ Generate outputs
    ❯ Generate src/generated/graphql.tsx
      ✔ Load GraphQL schemas
      ⠹ Load GraphQL documents
        Generate
```

```
EXPLORER
> OPEN EDITORS
v SPACEX
  > node_modules
  > public
  v src
    v components
      > Launch
      v LaunchDetails
        TS index.tsx U
        TS LaunchDetails.tsx U
        TS query.ts U
        # style.css U
      v generated
        TS graphql.tsx U
    # App.css
    TS App.tsx
    App.test.tsx
    # index.css
    TS index.tsx
    logo.svg
> OUTLINE
> TIMELINE
> NPM SCRIPTS
```

TS graphql.tsx ×

src > generated > TS graphql.tsx

```
1   import { gql } from '@apollo/client';
2   import * as Apollo from '@apollo/client';
3   export type Maybe<T> = T | null;
4   export type Exact<T extends { [key: string]: unknown }> = { [K in key
5   /** All built-in and custom scalars, mapped to their actual values */
6   export type Scalars = {
7     ID: string;
8     String: string;
9     Boolean: boolean;
10    Int: number;
11    Float: number;
12    /** The `Upload` scalar type represents a file upload. */
13    Upload: any;
14  };
15
16  export type Query = {
17    __typename?: 'Query';
18    capsule?: Maybe<Capsule>;
19    capsules?: Maybe<Array<Maybe<Capsule>>>;
20    core?: Maybe<Core>;
21    cores?: Maybe<Array<Maybe<Core>>>;
22    dragon?: Maybe<Dragon>;
23    dragons?: Maybe<Array<Maybe<Dragon>>>;
24    history?: Maybe<Array<Maybe<History>>>;
25    info?: Maybe<Info>;
26    landingpad?: Maybe<Landingpad>;
```

Activate Windows

- everything has been exported from the server here which we can access whenever we want
- let us work on UI
- initialize apollo client and use the apollo provider so we can use components in react'
- import gql,Apolloprovider , InMemoryCache,ApolloClient from apollo server
- initialize apollo client and give it uri for SpaceX
- cache: in memory
- wrap the app component with apollo provider and pass to the client

index.ts

```
import React from 'react';

import ReactDOM from 'react-dom';

import {gql,Apolloprovider , InMemoryCache,ApolloClient} from 'apolloClient';

import './index.css';

import app from './App';

import * as serviceworker from './serviceWorker';

const client =new ApolloClient({

url: 'https//spacexdata.herokuapp.com/graphql';

cache: new InMemoryCache()

})

 ReactDOM.reader(

<ApolloProvider client ={client}>

</App>

<ApolloProvider>,

document.getElementById('root')

);

serviceWorker.unregister();
```

- let's work on the component
- build a container component which will include the actual launch component

index.ts

```
import React from 'react';
import {useLaunchQuery} from '../../generated /graphql';
import launch from './Launch';
const LaunchContainer=()=>{
  const {data,error,loading}=useLaunchQuery();
  if (loading)
  return <div> Data is Loading</div>
}
if (error||!data){
   return <div>there was an error</div>
}
 return <Launch data=I{data}/>
};
export default LaunchContainer;
```

launch.tsx

```
import React from'react';
import{LaunchesQuery} from '../../generated/graphql';
import'./style.css'
interface props{
  data:LaunchQuery;
}
const Launch:React.FC<props>=({data})=>{
<div className=Launches'>
   <h3>All space X Launches<h3/>
     <ol className='LaunchesOL'>
        !!data.Launchs && data.launches.map(\(launch, i)=.!!Launch $$(
              <li key{i}className='LaunchesItem'>
                   <Launch.mission-name}-(launch.launch-year}({launch.launch.success})
        </li>
       )
     )}
   </ol>
 </div>
 );

export default launch;
```

LaunchDetails

```
  if (error){
    return <div> There was an Eroor</div>
 }
    if (!data){
       return<div>please select a mission for itd detail</div>
 }
   return <LaunchDetails data={data}/>
};
export default Lunchdetails;
```

- lets create LunchDetails componnt

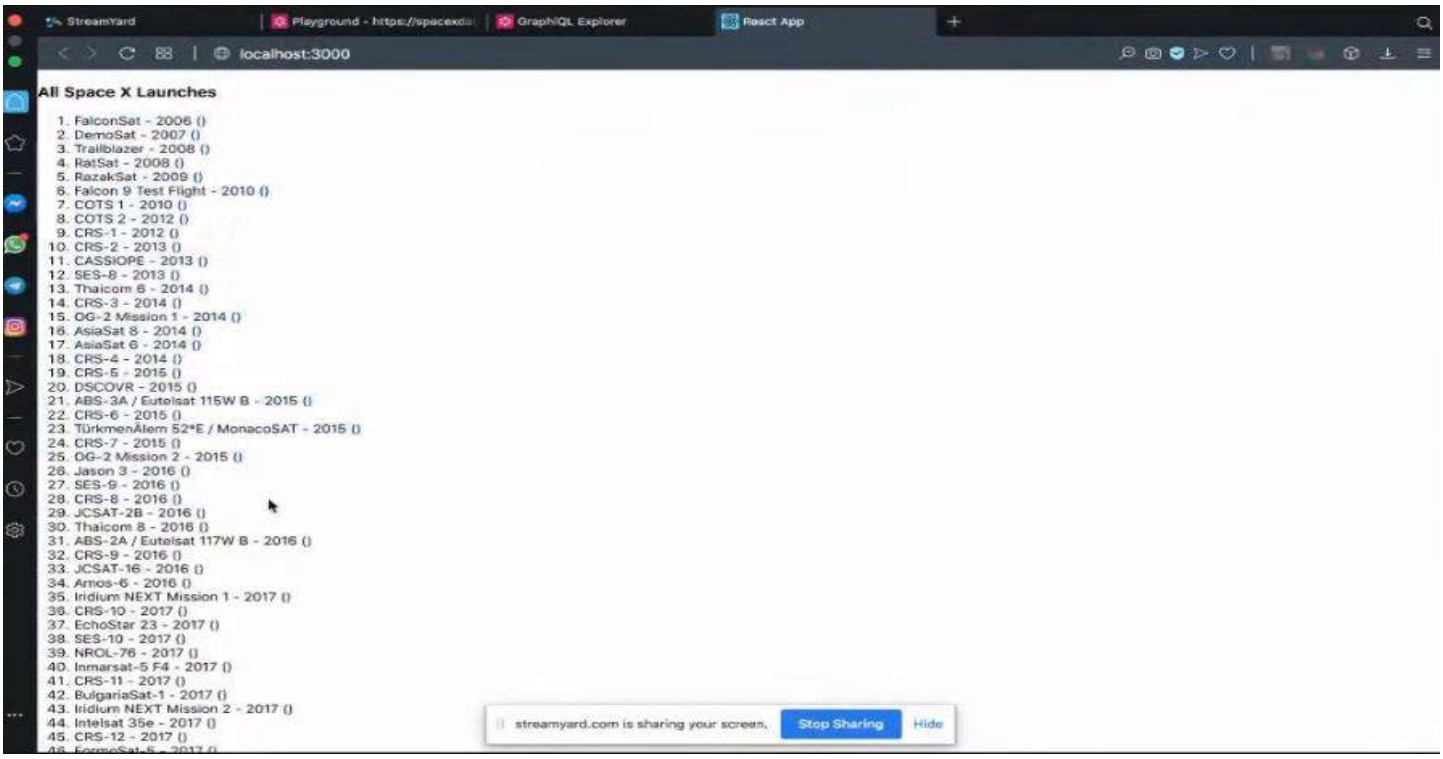
LaunchDetails.tsx

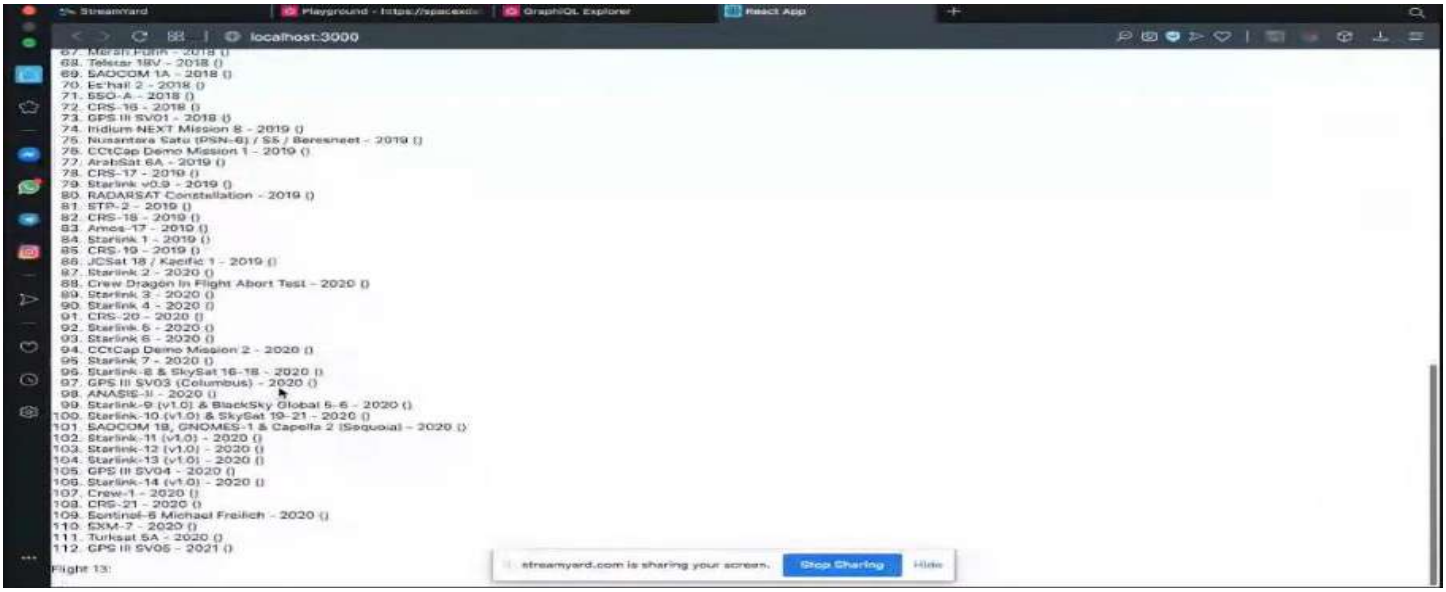
```
import React from 'react';
import[LaunchInfoQuery} from'../.../generated/gra[hql';
import './style.css';
 interface Props{
   data:LaunchInfoQuery
 }
 const LaunchDetails:React.FC<Props>=({data})=>
  if (!data-launch){
    return <div>Launch unavailable<div>
}
   return(
    <div className='LaunchDetails'>
     <div className='LaunchDetailsStatus '>
        <span>Flight {data.launch.flight-number}:</span>
    </div>
  )
};
export default LaunchDetails;
```

app.tsx

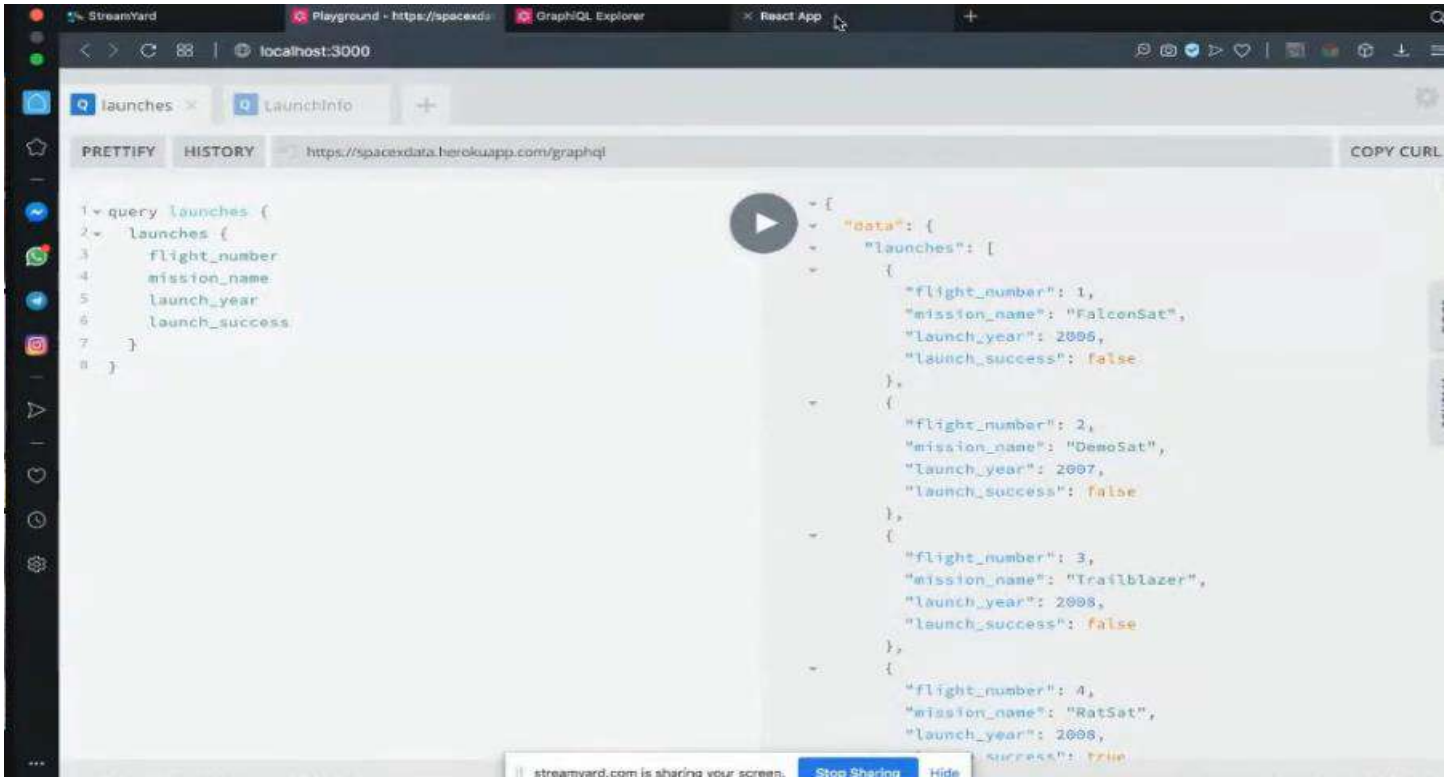
```
import React from'react';
import Launch from'./components/Launch';
import LaunchDetails from './components/LaunchDetails';
  const App=()=>{
   <div className='App'>
    <Launch/>
    <LaunchDetails/>
   </div>
```

- these are all space x launches

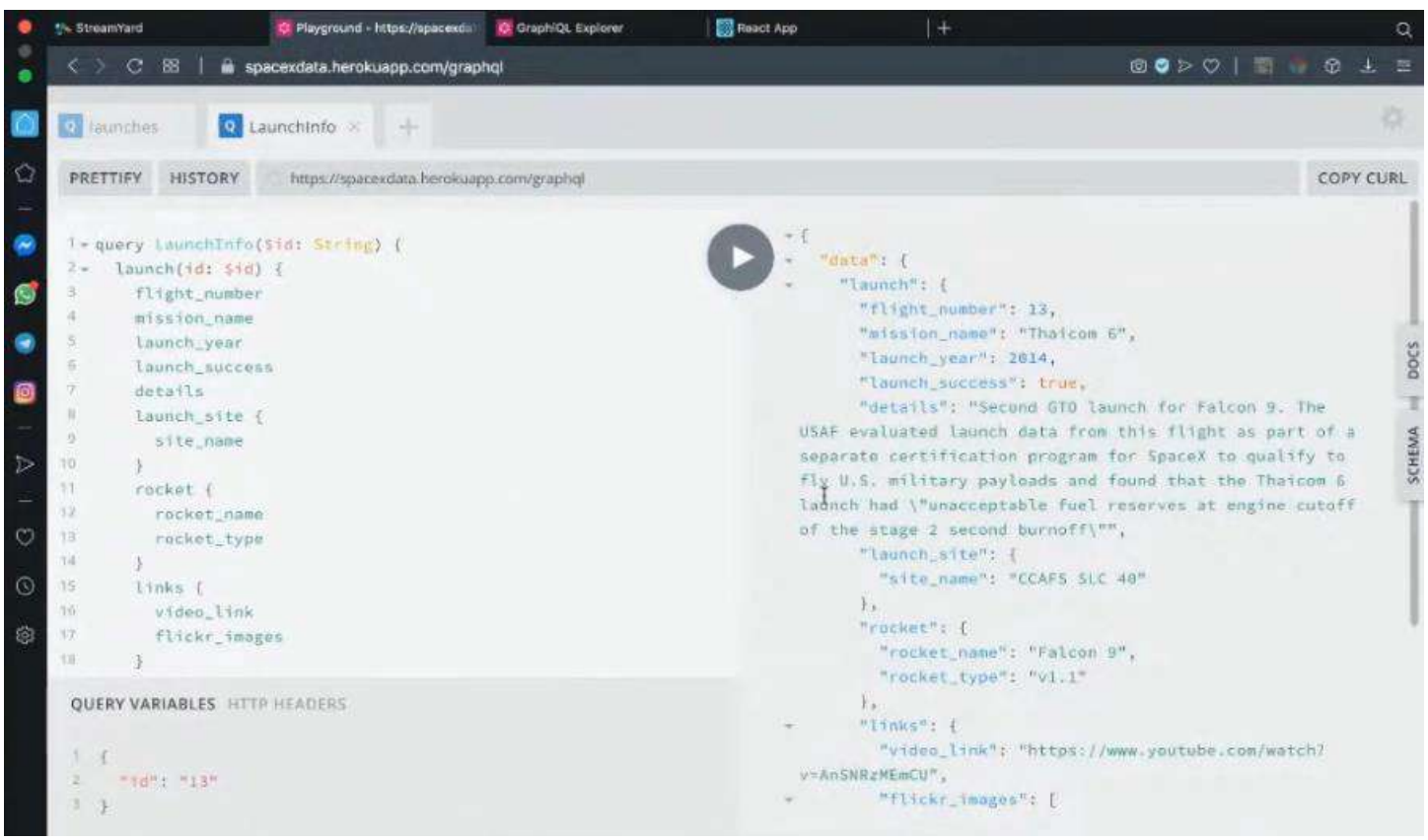




- This is our first query



- This is our 2nd query



- lets put some CSS styles so it looks better

EXPLORER

OPEN EDITORS

SPACEX

node_modules

public

src

components

Launch

LaunchDetails

generated

graphql.tsx

App.css

App.tsx

App.tsx | styles.css

Users > adil > Documents > GitHub > project8 > src > components > LaunchList > styles.css

```
.Launches {
    height: 100vh;
    overflow: hidden auto;
    background-color: #ecece;
    width: 300px;
    padding-left: 20px;
    padding-right: 20px;
}

.LaunchesOL {
    list-style: none;
    margin: 0;
    padding: 0;
```

localhost:3000

All Space X Launches

FalconSat - 2006 ()

DemoSat - 2007 ()

Trailblazer - 2008 ()

RatSat - 2008 ()

RazakSat - 2009 ()

Falcon 9 Test Flight - 2010 ()

COTS 1 - 2010 ()

COTS 2 - 2012 ()

CRS-1 - 2012 ()

CRS-2 - 2013 ()

CASSIOPE - 2013 ()

SES-8 - 2013 ()

Thaicom 6 - 2014 ()

CRS-3 - 2014 ()

streamyard.com is sharing your screen. Stop Sharing Hide

EXPLORER

OPEN EDITORS

SPACEX

node_modules

public

src

components

Launch

index.tsx

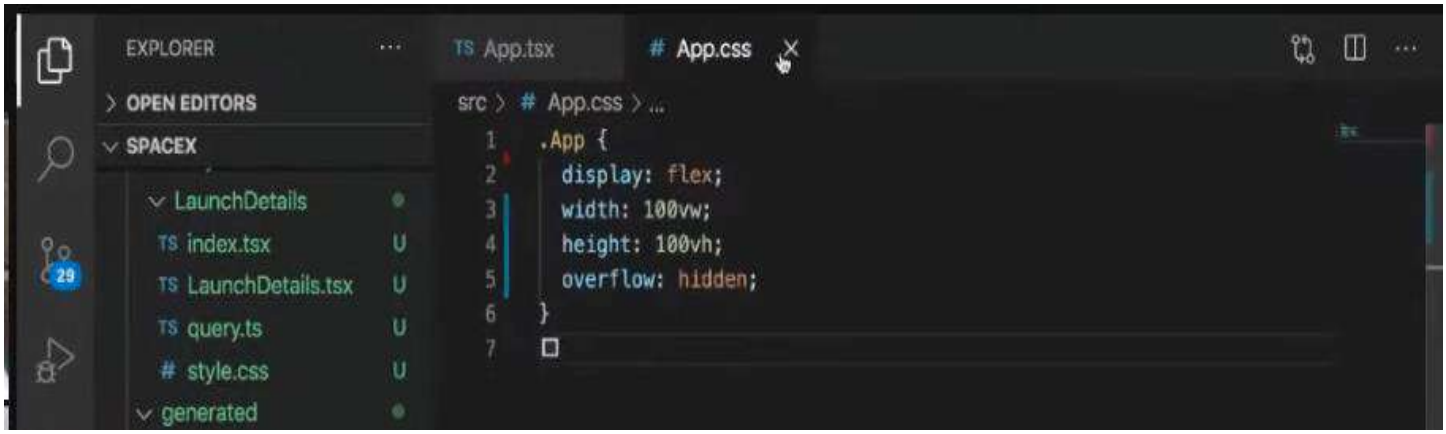
Launch.tsx

query.ts

App.tsx | style.css

src > components > LaunchDetails > style.css

```
.LaunchDetailsImage-list {
  display: grid;
  grid-gap: 20px;
  grid-template-columns: repeat(2, 1fr);
  margin-top: 40px;
  padding-bottom: 100px;
}

.LaunchDetailsImage {
  width: 100%;
}
```



- add flight numbers
- name of mission and type of rocket 'in the Launch details as

```
<h1>
 {data.launch.mission-name}-{data.launch.rocket?.rocket -name }
 </h1>
```

- add description

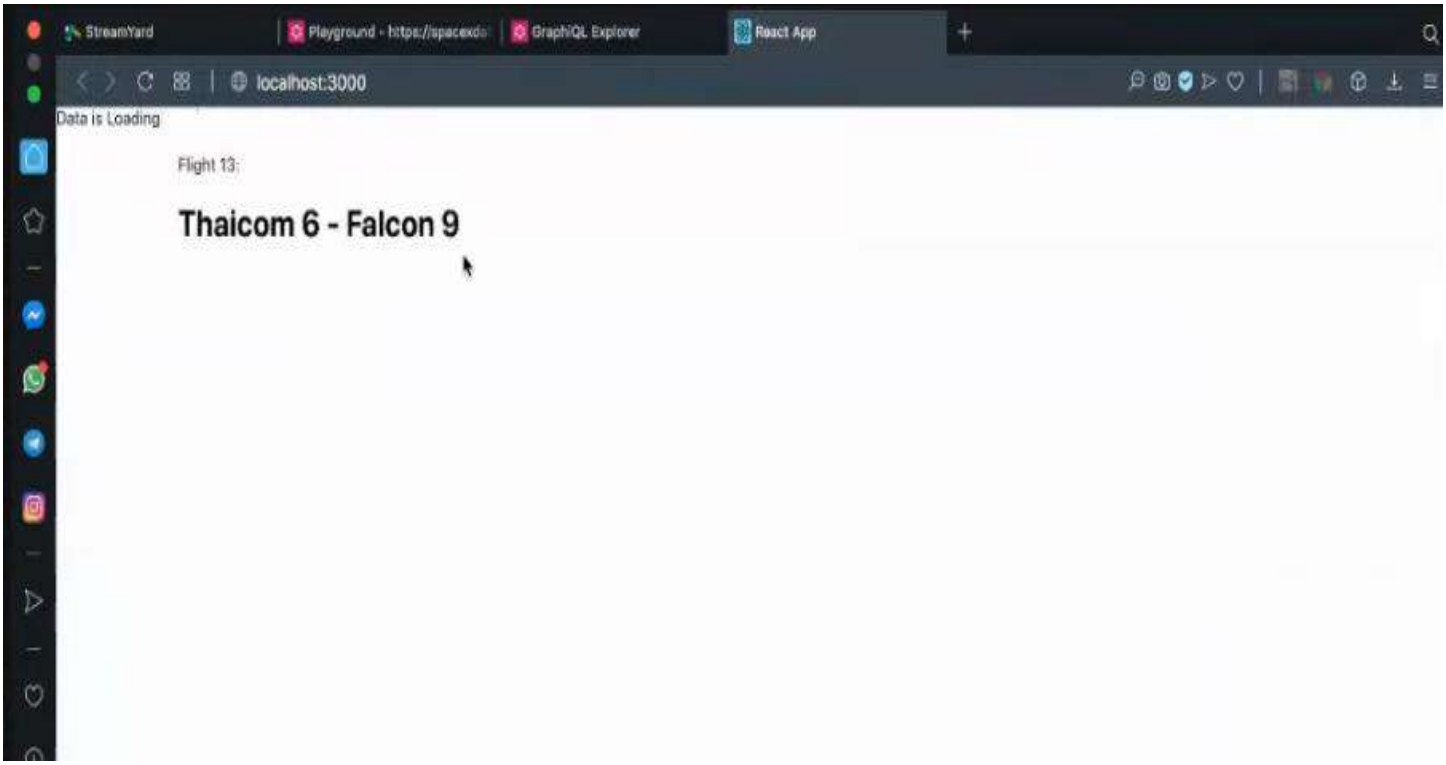
```
<p>{data.launch.details></p>
```

- Add site and year as well as add images of ot as

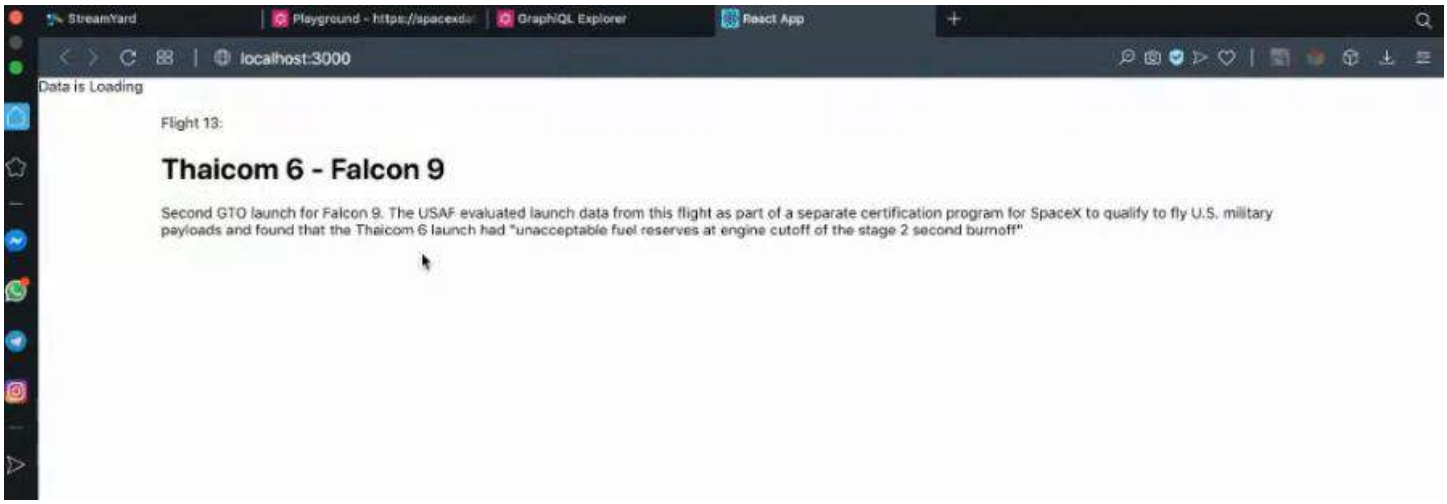
```
import React from 'react';
import[LaunchInfoQuery} from'../.../generated/gra['hql'];
import './style.css';
 interface Props{
   data:LaunchInfoQuery
 }
 const LaunchDetails:React.FC<Props>=({data})=>
  if (!data-launch){
    return <div>Launch unavailable<div>
}
   return(
    <div className='LaunchDetails'>
     <div className='LaunchDetailsStatus '>
        <span>Flight {data.launch.flight-number}:</span>
    </div>
            <h1>

           {data.launch.mission-name}-{data.launch.rocket?.rocket -name }

           </h1>

       <p>{data.launch.details></p>
       <p>launched from{data.launch-site?.site_name}in {data.launch.launch-year}</p>
       <div>!!data.launch.link&&!!data.launch.link.flickr_images&&(
          <div>
```

```
        {data.launch.link.flickrimages.map(image=>
          image?<img src={image}:null/>
        )}
      </div>
    )



  )
};
export default LaunchDetails;
```

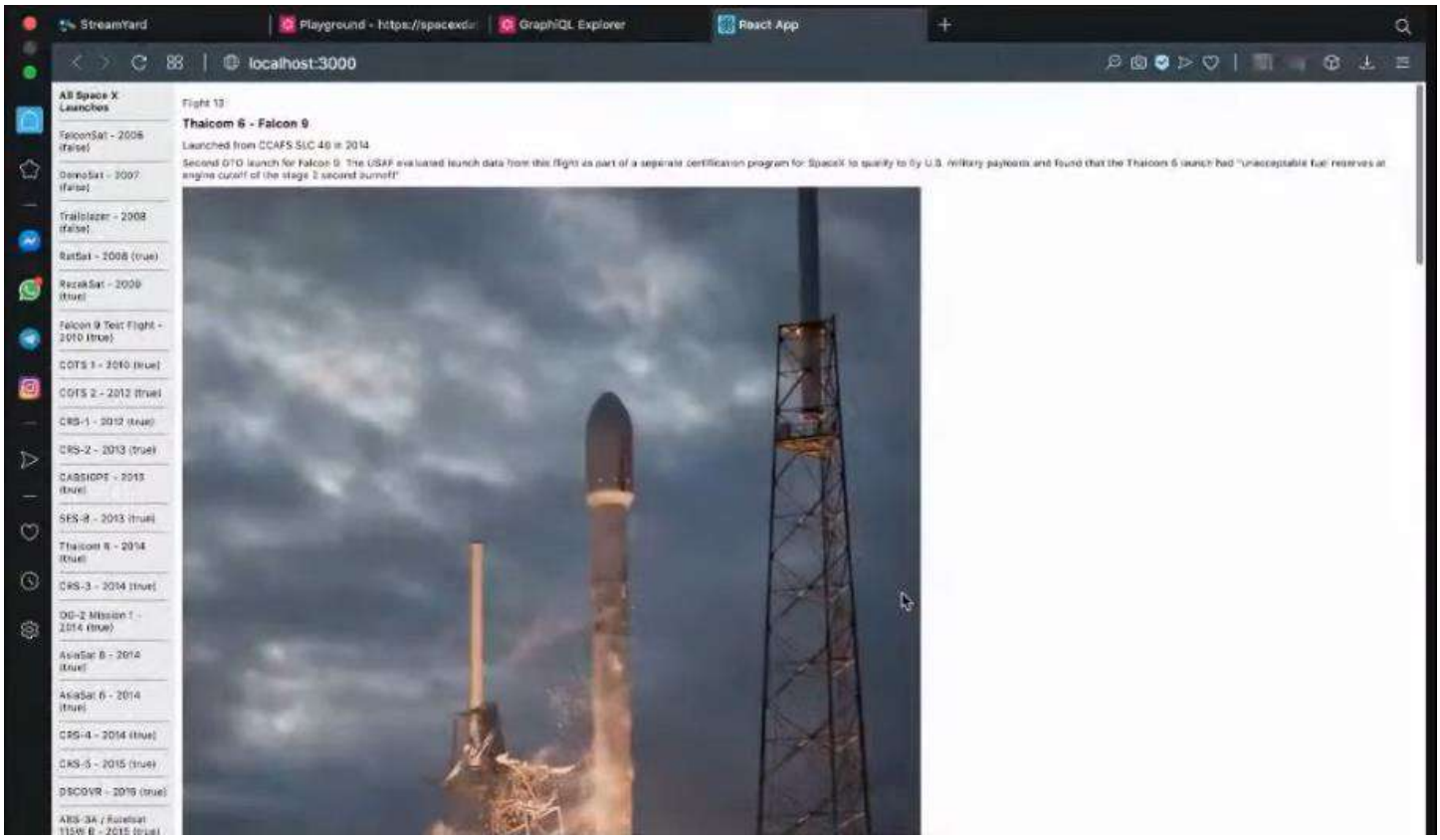
- now look into the browser te name, type of rocket, and description is given



-



- Here is the name, type, picture, and the year of launch as well as shown in the browser as



Session 18: Introduction to Redux

Redux Toolkit

The Redux Toolkit package is intended to be the standard way to write Redux logic. It was created to help address three common concerns about Redux:

- "Configuring a Redux store is too complicated"
- "I have to add a lot of packages to get Redux to do anything useful"
- "Redux requires too much boilerplate code"

We can't solve every use case, but in the spirit of create-react-app and apollo-boost, we can try to provide some tools that abstract over the setup process and handle the most common use cases, as well as include some useful utilities that will let the user simplify their application code.

Because of that, this package is deliberately limited in scope. It does not address concepts like "reusable encapsulated Redux modules", data caching, folder or file structures, managing entity relationships in the store, and so on.

That said, these tools should be beneficial to all Redux users. Whether you're a brand new Redux user setting up your first project, or an experienced user who wants to simplify an existing application, Redux Toolkit can help you make your Redux code better.

What's Included#

Redux Toolkit includes these APIs:

- configureStore(): wraps createStore to provide simplified configuration options and good defaults. It can automatically combine your slice reducers, adds whatever Redux middleware you supply, includes redux-thunk by default, and enables the use of the Redux DevTools Extension.
- createReducer(): that lets you supply a lookup table of action types to case reducer functions, rather than writing switch statements. Also, it automatically uses the immer library to let you write simpler immutable updates with normal mutative code, like state.todos[3].completed = true.
- createAction(): generates an action creator function for the given action type string. The function itself has toString() defined so that it can be used in place of the type constant.
- createSlice(): accepts an object of reducer functions, a slice name, and an initial state value, and automatically generates a slice reducer with corresponding action creators and action types.

- createAsyncThunk: accepts an action type string and a function that returns a promise, and generates a thunk that dispatches pending/fulfilled/rejected action types based on that promise
- createEntityAdapter: generates a set of reusable reducers and selectors to manage normalized data in the store

The createSelector utility from the Reselect library, re-exported for ease of use.

Using Redux with React Hooks

What are Hooks?

Hooks were added to React in 16.8 and allow you to access things like state, React lifecycle methods, and other goodies in function components that were previously only available in-class components.

For example, a React class component like this:

```
class Count extends React.Component {
  state = {
    count: 0
  };

  add = () => {
    this.setState({ count: this.state.count + 1 });
  }

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.add}>Add</button>
      </div>
    );
  }
}
```

Could be rewritten as a function component using Hooks like this:

```
const Count = () => {
  // state variable, initialized to 0
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
```

```
      <button onClick={() => setCount(count + 1)}>Add</button>
    </div>
  );
};
```

The code is more concise and allows teams to make more use of function components without having to convert them to class components as soon as they need state or access to the React lifecycle. This post isn't really about Hooks in general, so I'd check out the excellent Hooks documentation if you'd like to learn more.

How to use Redux with Hooks

React Redux now includes its useSelector and useDispatch Hooks that can be used instead of connecting.

useSelector is analogous to connect's mapStateToProps. You pass it a function that takes the Redux store state and returns the pieces of the state you're interested in.

useDispatch replaces connect's mapDispatchToProps but is lighter weight. All it does is return your store's dispatch method so you can manually dispatch actions. I like this change, as binding action creators can be a little confusing to newcomers to React-Redux.

---

1. Using Redux with React Hooks

- create component folders
- make a file of counter.js
- create onclick buttons
- create usestate to implement the value
- create increment, decrement, and increment by value button as shown
- create an input text
- use counter variable so, whenever you want to want to increase or decrease the value it is being shown in this

counter

counter.js

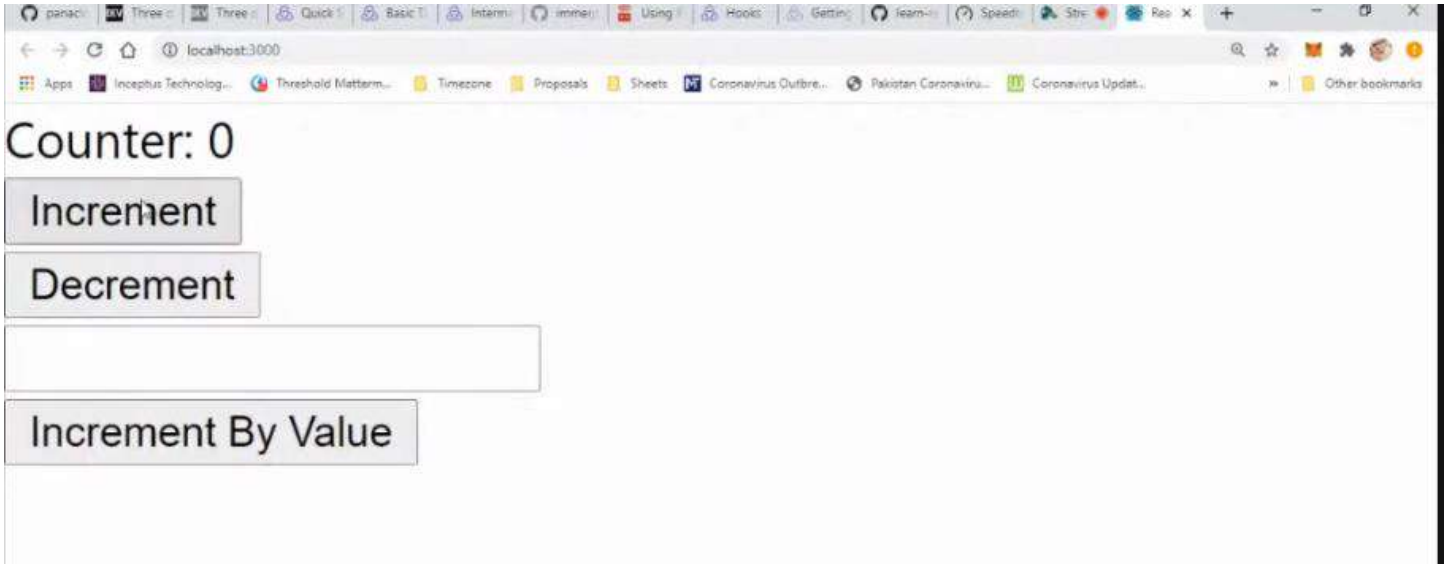
```
import React, { useState } from 'react';
import {useDispatch, useSelector} from 'react-redux';
import { increment, decrement, incrementByAmount } from '../store/action';

function Counter() {
    const [counter, setCounter] = useState(0);
    const [value,setValue]=useState(0);
    return (
   <div>
     <div>Counter:{counter}</div>
       <div>

           <button onClick={()=>{
               setCounter(counter+1);

           }}>Increment</button>
           <input type="text" onchnage
           <br/>
           <button onClick={()=>{
            setCounter(counter-1);

           }}>Decrement</button>
       </div>
       <div>
       <input type="text" onchange={(e)=>{
           setCounter(counter+number(value);
         }}>Increment By Value</button>


       <br/>
       <button onClick={()=>{
           setCounter(counter-1);
          }}>increment</button>
      </div>
    </div>
```

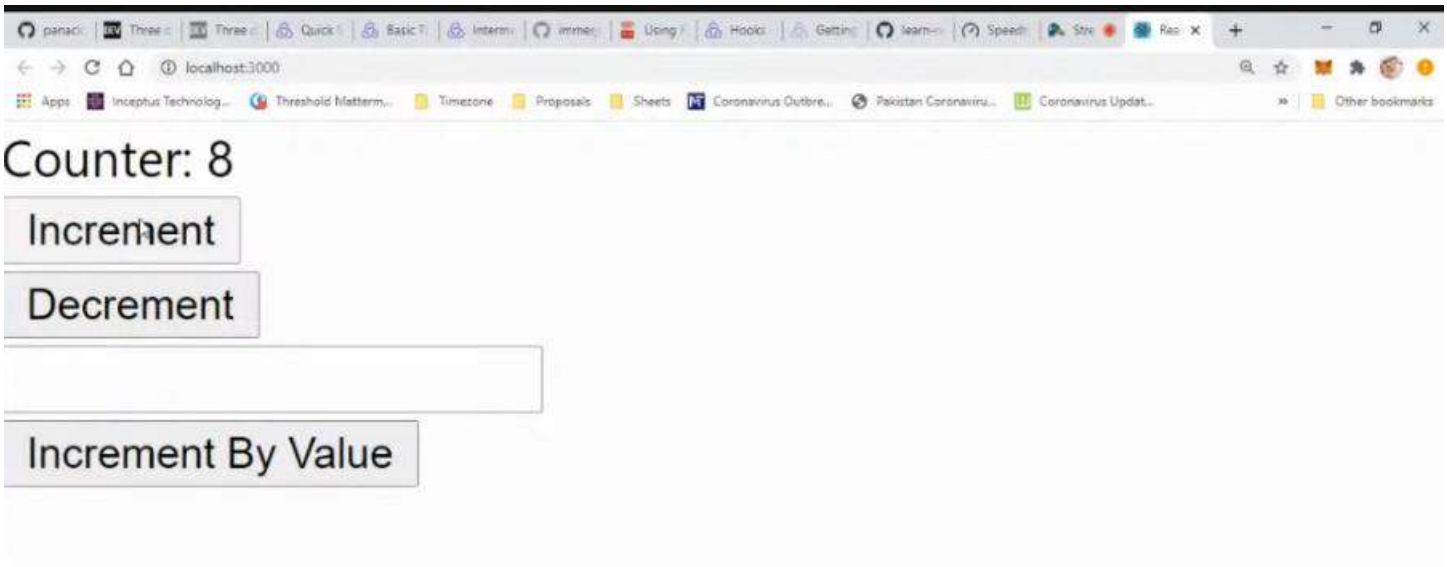
- Now create App.js file and import counter component in it

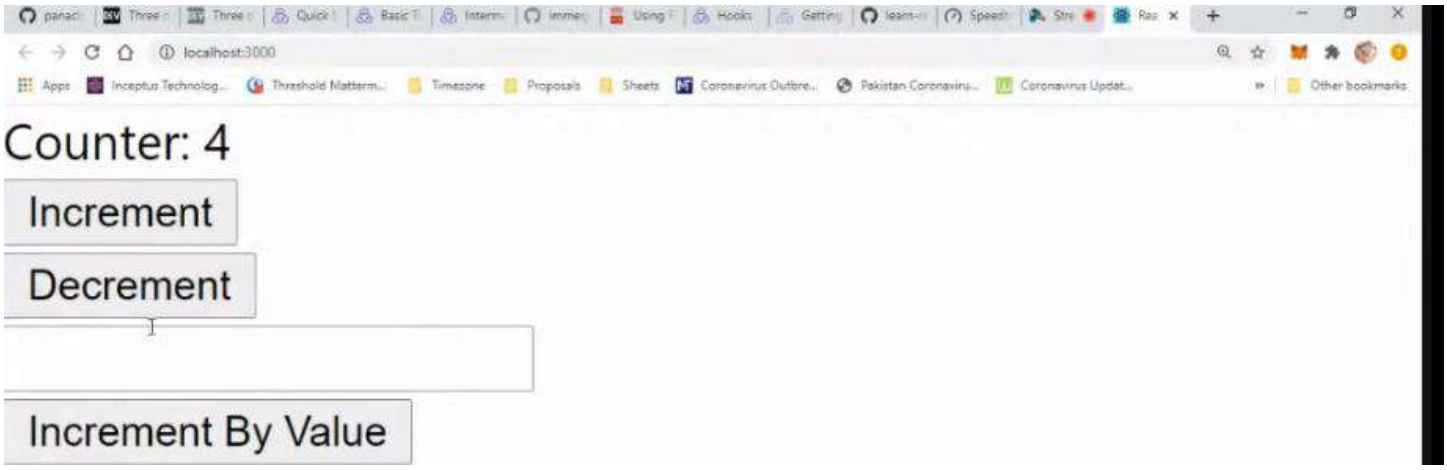
App.js

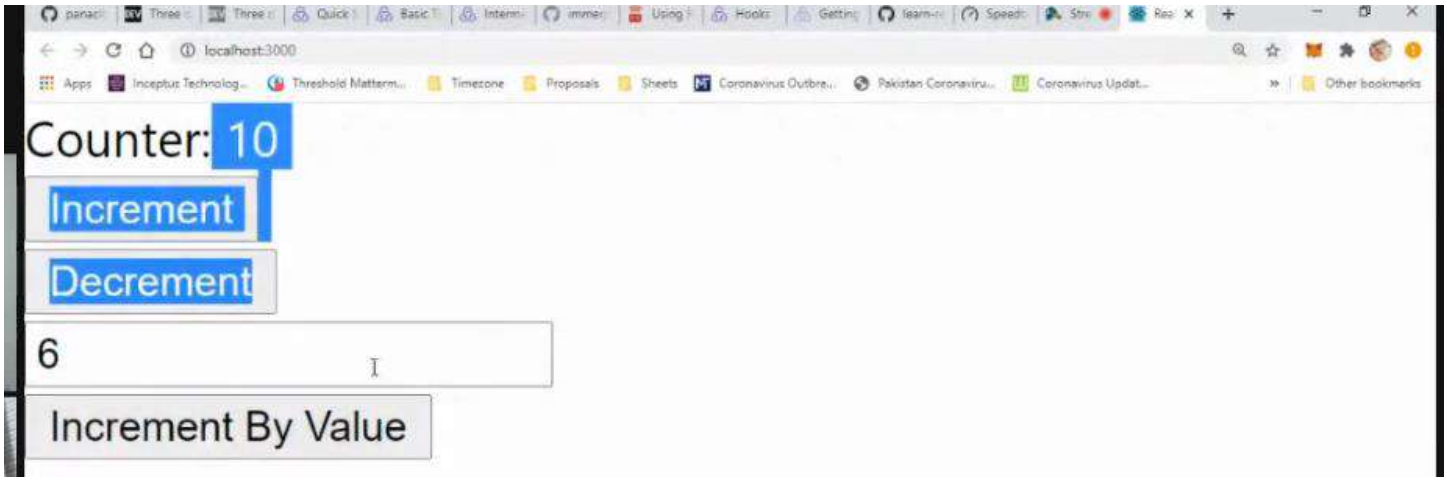
```
import React from 'react';
import './App.css';
import Counter from './component/Counter';

function App(){
  return(
  <div>
   <counter/>
   </div>
  );
}
```

- Now go to the browser and look into the output by pressing all three buttons









- create another component counterDispaly.js

```
counterDisplay.js
function counterDisplay(){
  return(
  <div>
  counter display
   </div>
  );
}
export default CounterDisplay;
```

- install a library of redux and redux react
- to create a folder of store.js

```
import {createstore} from "redux";

export const store=createStore(null)
```

- create a wrapper over the app in index.js as
- create a provider inside which wrap the index
- as well as provide store

```
index.js
import React from 'react';
import *as serviceWorker from './serviceWorker';
import {provider} from 'react-redux';


ReactDOM.render(
   <React.StricMode>
     <provider store{store}>
     <App/.
     </provider>
</React.StrictMode>,
document.getElementById('root')
);
```

- create a message by using usedispatch as a function that will pass the message to the redux store and importing it as shown in app.js

```
const dispatch=useDispatch();
```

- call this function and pass a message as an object

```
dispatch({type:'INCREMENT"});

dispatch{tpye:"DECREMENT"});
```

counter.js

```
import React, { useState } from 'react';
import {useDispatch, useSelector} from 'react-redux';
import { increment, decrement, incrementByAmount } from '../store/action';

function Counter() {
    const [counter, setCounter] = useState(0);
    const [value,setValue]=useState(0);
     const dispatch=useDispatch();
    return (
   <div>
     <div>Counter:{counter}</div>
        <div>

            <button onClick={()=>{
                setCounter(counter+1);
               dispatch({type:'INCREMENT"});

            }}>Increment</button>
            <input type="text" onchnage
            <br/>
            <button onClick={()=>{
             setCounter(counter-1);
           dispatch({type:"DECREMENT"});

            }}>Decrement</button>
        </div>
        <div>
        <input type="text" onchange={(e)=>{
            setCounter(counter+number(value);
          }}>Increment By Value</button>
<br/>
        <button onClick={()=>{
            setCounter(counter-1);
           }}>increment</button>
       </div>
    </div>
```

- create a file of reducer in the store folder
- create a function of a store
- it will receive two properties

- state
- action

- it is a pure function means it does not update or resist an external property means no side effect of it
- whenever reducer call a state I need generals to state it will be created by the redux
- make cases as shown for all three functions and dispatch the message in counter.js as number
- inside a payload send object
- the store will call the reducer

reducer.js

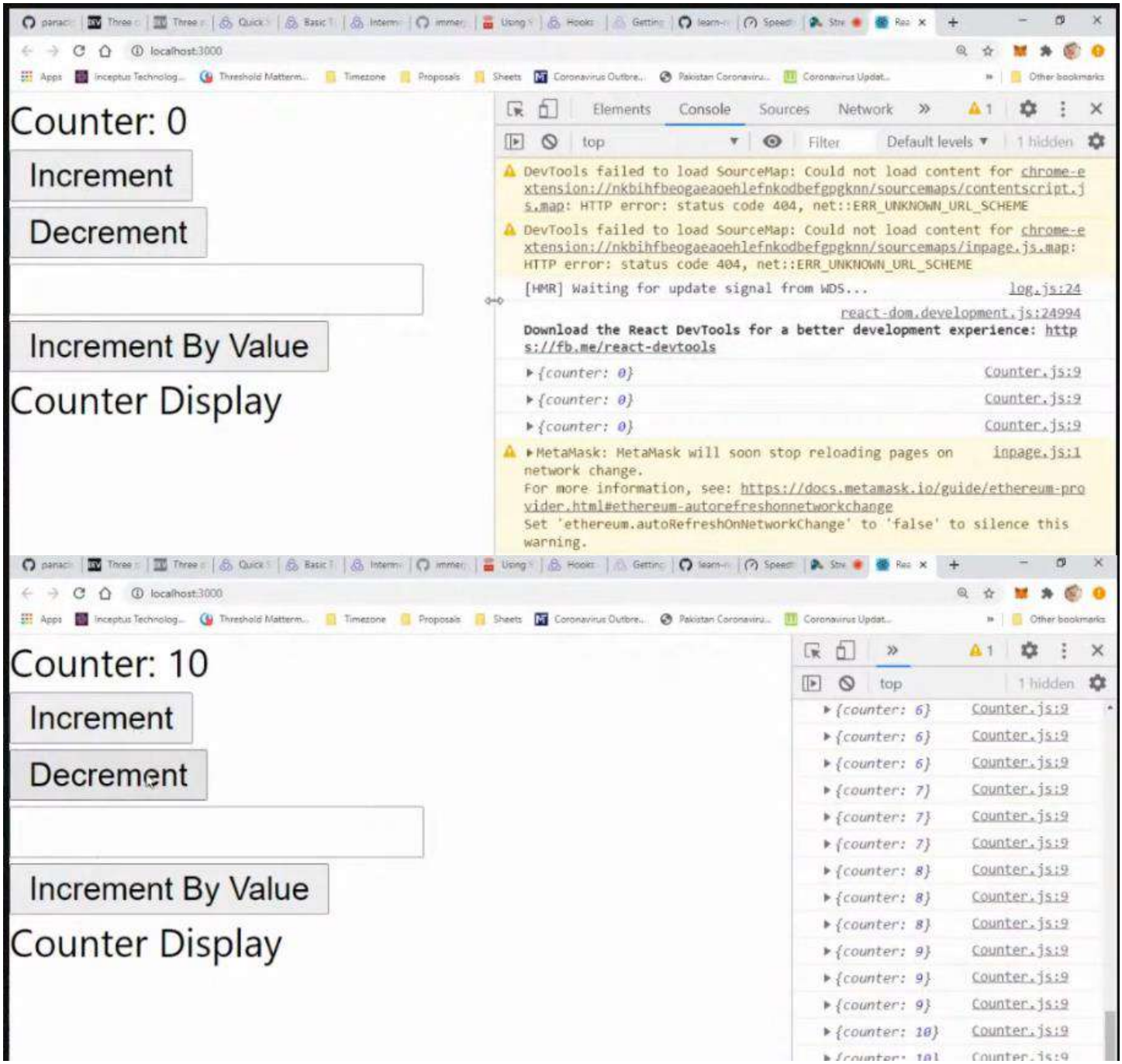
```
import { INCREMENT, DECREMENT, INCREMENT_BY_AMOUNT } from "./action";



 const initialState = {
}
export const counterReducer = (state=initialState, action)=>{
    switch(action.type) {
        case "INCREMENT": {
        case INCREMENT: {
            return {...state, counter: state.counter+1 }
        }
        case "DECREMENT": {
        case DECREMENT: {
            return {
                ...state, counter: state

        }
        case "INCREMENT_BY_AMOUNT": {
        case INCREMENT_BY_AMOUNT: {
            return {
                ...state, counter: state.counter+action.payload
            }
```

Counter.js

```
import React, { useState } from 'react';
import {useDispatch, useSelector} from 'react-redux';
import { increment, decrement, incrementByAmount } from '../store/action';

function Counter() {
   // const [counter, setCounter] = useState(0);
              const[value,setValue]=useState(0);
           const dispatch=useDispatch();
           const counter= useSelector((state)=>{
              return state.counter

              <div>
 });
return
  (<div>
      <div>Counter:{counter}</div>
<div>
                <button onClick={()=>{
            //   setCounter(counter+1);
               dispatch( {type:"INCREMENT"} );
                 dispatch( {type:"INCREMENT"} );

           }}>Increment</button>
           <br/>
           <button onClick={()=>{
                // setCounter(counter-1);
               dispatch( { type: "DECREMENT" } );
           }}>Decrement</button>
        </div>
        <div>
              function Counter() {
           <br/>
           <button onClick={()=>{
                setCounter(counter+ Number(value))
               dispatch( {type: "INCREMENT_BY_AMOUNT", payload:Number(value) } )
                 dispatch( {type: "INCREMENT_BY_AMOUNT", payload:Number(value) } )

           }}>Increment By Value</button>

        </div>

    </div>
```

Now import the reducer into the counter.js

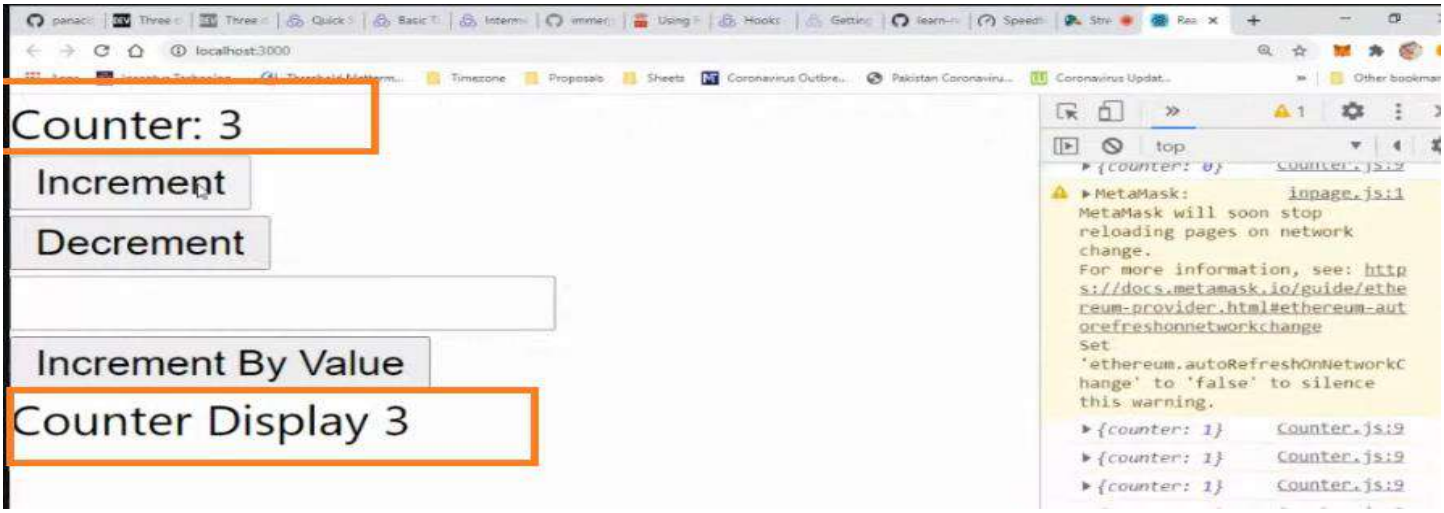
- Go to the browser and look out the output



- Now to work with counter display
- Import use selector from react-redux

- Use counter function =useSelector and provide sate inside the UseSelector and return it as;
- In the div section pass counter to the counter display
- Now check the display counter by incrementing or decrementing the value as

```
JS CounterDisplay.js ×   JS Counter.js   JS reducer.js   JS store.js   JS index.js
src > components > JS CounterDisplay.js > CounterDisplay
import React from 'react';
import {useSelector} from 'react-redux'


function CounterDisplay() {
    const counter = useSelector((state)=>{
        return state.counter
    })
  return (
    <div >
        Counter Display {counter}
    </div>
  );
}
```



create a file name action

- create variables as shown
- create functions for these three variables and return them
- replace dispatch function function in counter.js as

```
dispatch(increment());
```

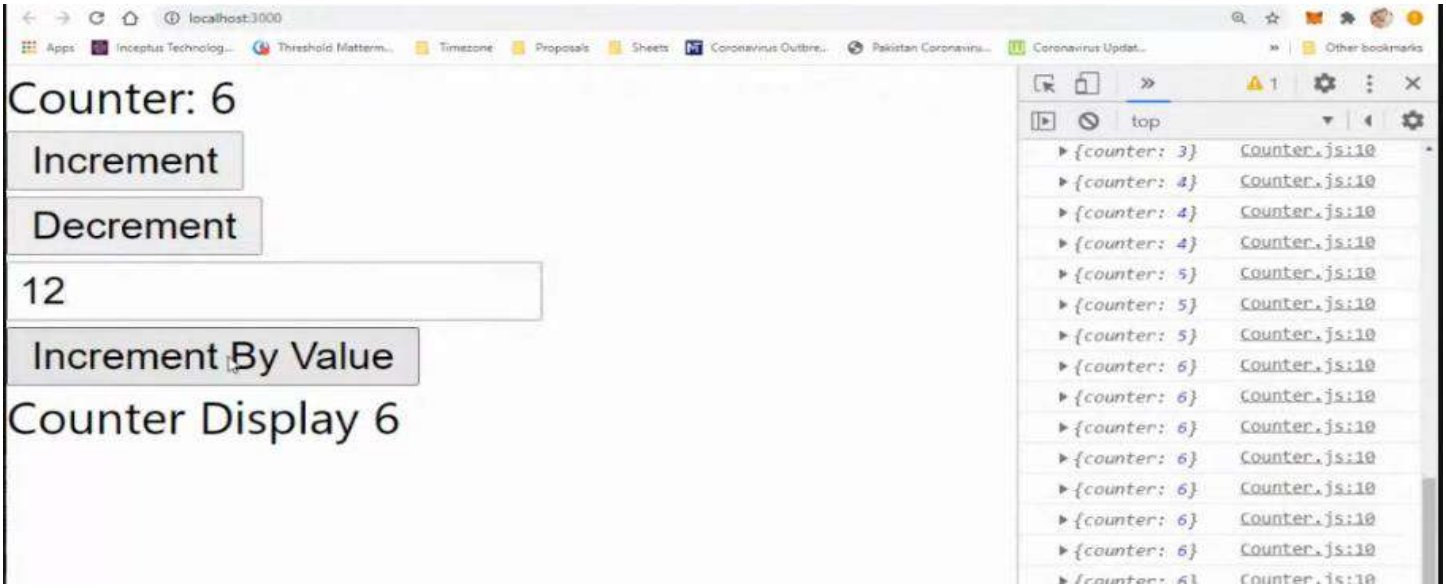
```
dispatch( decrement());

dispatch(incrementByAmount(Number(value)))
```

Action.js

```
export const INCREMENT = "INCREMENT";
export const DECREMENT = "DECREMENT";
export const INCREMENT_BY_AMOUNT = "INCREMENT_BY_AMOUNT";

export const increment = ()=>{
    return {
        type: INCREMENT
    }
}

export const decrement = ()=>{
    return {
        type: DECREMENT
    }
}

export const incrementByAmount = (number)=>{
    return {
        type: INCREMENT_BY_AMOUNT,
        payload: number
   }
}
```

Counter.js

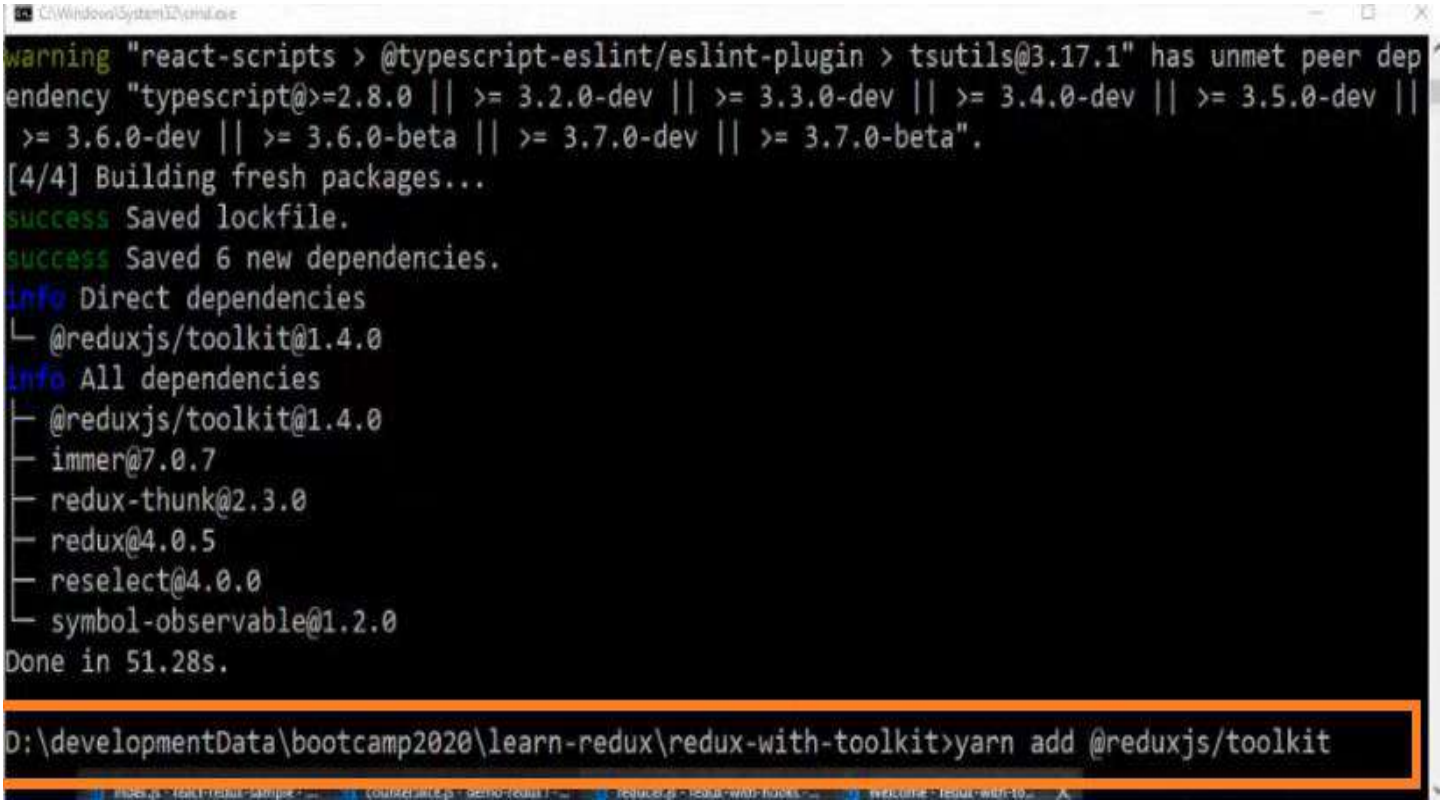
```
import React, { useState } from 'react';
import {useDispatch, useSelector} from 'react-redux'
import { increment, decrement, incrementByAmount } from '../store/action';

function Counter() {
   // const [counter, setCounter] = useState(0);
                const[value,setValue]=useState(0);
            const dispatch=useDispatch();
            const counter= useSelector((state)=>{
               return state.counter
               <div>
 });
return
  (<div>
      <div>Counter:{counter}</div>
<div>
                <button onClick={()=>{
             //   setCounter(counter+1);
              //  dispatch( {type:"INCREMENT"} );
                //  dispatch( {type:"INCREMENT"} );
                dispatch(increment())

            }}>Increment</button>
            <br/>
            <button onClick={()=>{
                 // setCounter(counter-1);
               // dispatch( { type: "DECREMENT" } );
              dispatch( decrement());

            }}>Decrement</button>
        </div>
        <div>
               function Counter() {
            <br/>
            <button onClick={()=>{
                 setCounter(counter+ Number(value))

                 // dispatch( {type: "INCREMENT_BY_AMOUNT", payload:Number(value) } )
                dispatch(incrementByAmount(Number(value)))

          }}>Increment By Value</button>
      </div>
   </div>
```



2. Redux Toolkit

Redux toolkit will help to get out much of coding if you are going use more requirements

- create a react app
- add toolkit separately
- add react-redux



app.js

```
function App(){
 return(
 <div>
   Hello App Compnent
</div>
 );
}
export default App;
```

- create a component folder and add counter.js and counterDisplay.js
- create a store.js

store.js

```
import {configureStore}from'@reduxjs/toolkit';
export const store=configureStore({
   reducer:{
      counter:counterReducers,
           }
})
```

- create provider in index.js

Index.js

```
import React from 'react';
import *as serviceWorker from './serviceWorker';
import {provider} from 'react-redux';


ReactDOM.render(
   <React.StricMode>
     <provider store{store}>
     <App/>
     </provider>
</React.StrictMode>,
document.getElementById('root')
);
```

- create counterSlice.js

in it, like redux react we are not needed to put state action initial state with each case it is a very simple code as seen

counterSlice.js

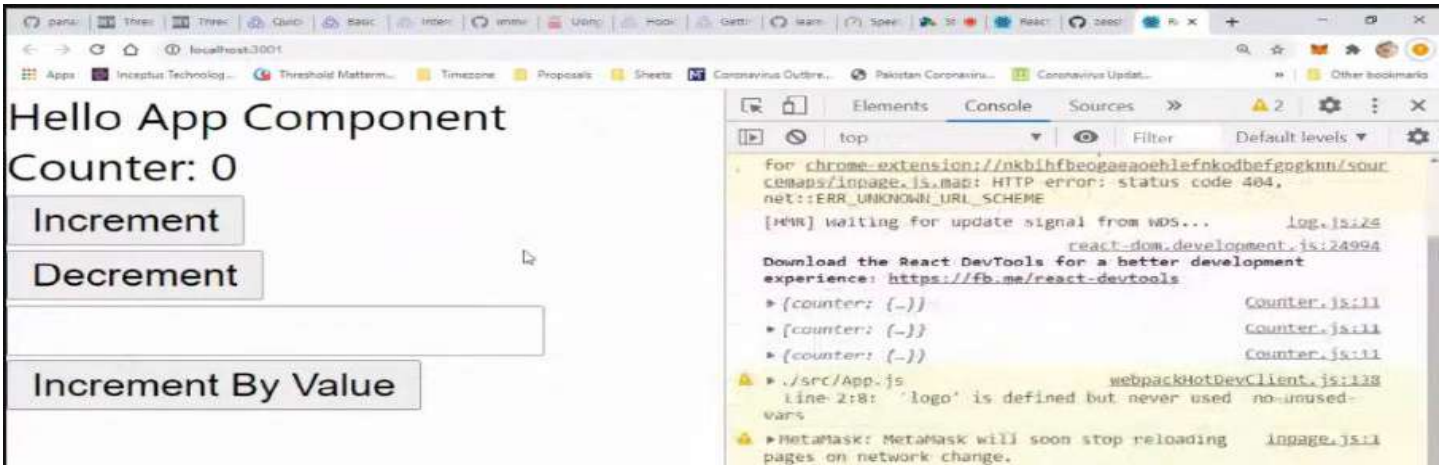
```
import {createSlice}from"@redux.js/toolkit";
export const counterSlice=createSlice({
name:"counter",
initialState:{
  count:0
 }
 reducers:{
   increment:(state)=>{
     state.count=state.count++
    }'
    decrement:state=>{
      sate.count--;
    },
    incrementByAmount:{state,action)=>{
     state.count+=action.payload
    }
  }
});
export const {increment,decrement,incrementByAmount}=counterSlice.action
export const {reducer}=counterSlice.reducer;
```

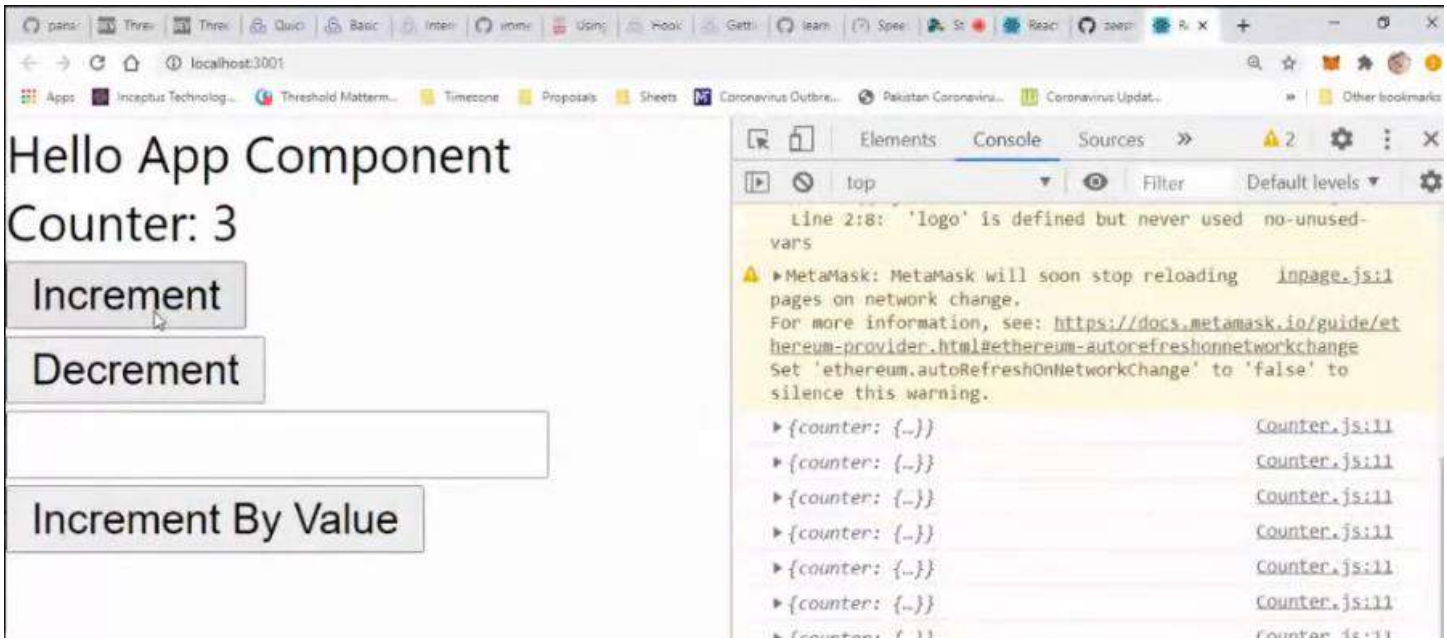
- in counter.js everything will remain the same as in redux react

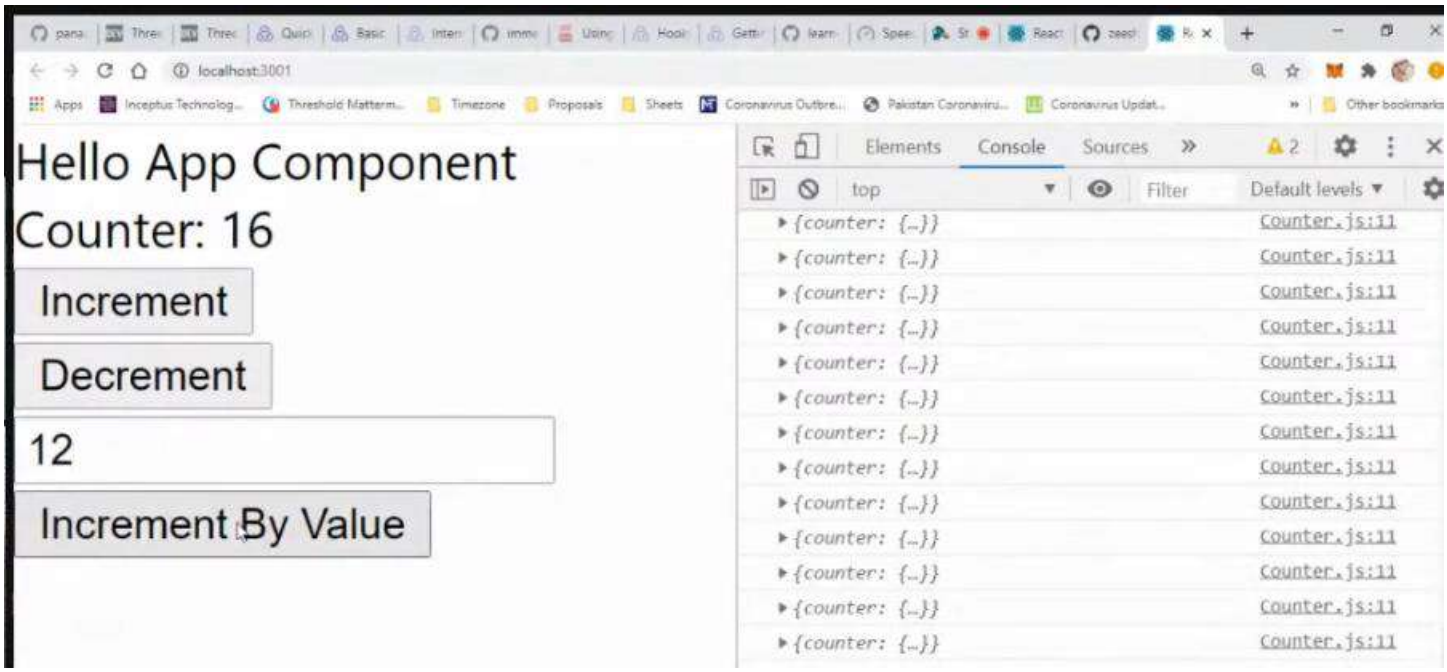
counter.js

```
import React, { useState } from 'react';
import {useDispatch, useSelector} from 'react-redux';
import { increment, decrement, incrementByAmount } from '../store/action';

function Counter() {
   // const [counter, setCounter] = useState(0);
    const[value,setValue]=useState(0);
const dispatch=useDispatch();
const counter= useSelector((state)=>{
            return state.counter.count

            <div>
 });
return
```

```
            }}>Increment</button>
            <br/>


        <button onClick={()=>{
                // setCounter(counter-1);
               // dispatch( { type: "DECREMENT" } );
              dispatch( decrement());

            }}>Decrement</button>
        </div>
        <div>
              function Counter() {
            <br/>
            <button onClick={()=>{
                setCounter(counter+ Number(value))

                // dispatch( {type: "INCREMENT_BY_AMOUNT", payload:Number(value) } )
                  dispatch(incrementByAmount(Number(value)))
            }}>Increment By Value</button>
        </div>
    </div>
```




localhost:3001
Hello App Component
Counter: 3
Increment
Decrement
Increment By Value
Elements
Console
Sources
top
Filter
Default levels
Line 2:8: 'logo' is defined but never used no-unused-vars
MetaMask: MetaMask will soon stop reloading pages on network change.
For more information, see: https://docs.metamask.io/guide/ethereum-provider.html#ethereum-autorefreshonnetworkchange
Set 'ethereum.autoRefreshOnNetworkChange' to 'false' to silence this warning.
inpage.js:1
{counter: {…}}
Counter.js:11


localhost:3001
Hello App Component
Counter: 16
Increment
Decrement
12
Increment By Value
Elements
Console
Sources
top
Filter
Default levels
{counter: {…}}
Counter.js:11

Session 18(A): Async_Redux_Toolkit

1. CreateAsyncThunk

A function that accepts a Redux action type string and a callback function that should return a promise. It generates promise lifecycle action types based on the action type prefix that you pass in and returns a thunk action creator that will run the promise callback and dispatch the lifecycle actions based on the returned promise.

This abstracts the standard recommended approach for handling async request lifecycles.

It does not generate any reducer functions, since it does not know what data you're fetching, how you want to track loading state, or how the data you return needs to be processed. You should write your reducer logic that handles these actions, with whatever loading state and processing logic is appropriate for your app.

Sample usage:

```
import { createAsyncThunk, createSlice } from '@reduxjs/toolkit';
import { userAPI } from './userAPI';

// First, create the thunk
const fetchUserById = createAsyncThunk(
  'users/fetchByIdStatus',
  async (userId, thunkAPI) => {
    const response = await userAPI.fetchById(userId)
    return response.data
  }
)

// Then, handle actions in your reducers:
const usersSlice = createSlice({
  name: 'users',
  initialState: { entities: [], loading: 'idle' },
  reducers: {
    // standard reducer logic, with auto-generated action types per reducer
  },
  extraReducers: {
    // Add reducers for additional action types here, and handle loading state as needed
    [fetchUserById.fulfilled]: (state, action) => {
      // Add user to the state array
      state.entities.push(action.payload)
    }
  }

})

// Later, dispatch the thunk as needed in the app

dispatch(fetchUserById(123))
```

Parameters#

createAsyncThunk accepts three parameters: a string action type value, a payloadCreator callback, and an options object.

type#

A string that will be used to generate additional Redux action type constants, representing the lifecycle of an async request:

For example, a type argument of 'users/requestStatus' will generate these action types:

pending: 'users/requestStatus/pending'

fulfilled: 'users/requestStatus/fulfilled'

rejected: 'users/requestStatus/rejected'

payloadCreator#

A callback function that should return a promise containing the result of some asynchronous logic. It may also return a value synchronously. If there is an error, it should either return a rejected promise containing an Error instance or a plain value such as a descriptive error message or otherwise a resolved promise with a RejectWithValue argument as returned by the thunkAPI.rejectWithValue function.

The payloadCreator function can contain whatever logic you need to calculate an appropriate result. This could include a standard AJAX data fetch request, multiple AJAX calls with the results combined into a final value, interactions with React Native AsyncStorage, and so on.

The payloadCreator function will be called with two arguments:

- arg: a single value, containing the first parameter that was passed to the thunk action creator when it was dispatched. This is useful for passing in values like item IDs that may be needed as part of the request. If you need to pass in multiple values, pass them together in an object when you dispatch the thunk, like dispatch(fetchUsers({status: 'active', sortBy: 'name'})).
- thunkAPI: an object containing all of the parameters that are normally passed to a Redux thunk function, as well as additional options:

    - dispatch: the Redux store dispatch method
    - getState: the Redux store getState method
    - extra: the "extra argument" given to the thunk middleware on setup, if available
    - requested: a unique string ID value that was automatically generated to identify this request sequence
    - signal: an AbortController.signal object that may be used to see if another part of the app logic has marked this request as needing cancelation.
    - rejectWithValue: rejectWithValue is a utility function that you can return in your action creator to return a rejected response with a defined payload. It will pass whatever value you give it and return it in the payload of the rejected action.

The logic in the payloadCreator function may use any of these values as needed to calculate the result.

Options#

An object with the following optional fields:

condition: a callback that can be used to skip execution of the payload creator and all action dispatches, if desired. See Canceling Before Execution for a complete description.

dispatchConditionRejection: if condition() returns false, the default behavior is that no actions will be dispatched at all. If you still want a "rejected" action to be dispatched when the thunk was canceled, set this flag to true.

Return Value#

createAsyncThunk returns a standard Redux thunk action creator. The thunk action creator function will have plain action creators for the pending, fulfilled, and rejected cases attached as nested fields.

- Using the fetchUserById example above, createAsyncThunk will generate four functions:

    - fetchUserById, the thunk action creator that kicks off the async payload callback you wrote
    - fetchUserById.pending, an action creator that dispatches an 'users/fetchByIdStatus/pending' action
    - fetchUserById.fulfilled, an action creator that dispatches an 'users/fetchByIdStatus/fulfilled' action
    - fetchUserById.rejected, an action creator that dispatches an 'users/fetchByIdStatus/rejected' action

When dispatched, the thunk will:

- dispatch the pending action
- call the payloadCreator callback and wait for the returned promise to settle
- when the promise settles:
- if the promise is resolved successfully, dispatch the fulfilled action with the promised value as action.payload
- if the promise is resolved with a rejectWithValue(value) return value, dispatch the rejected action with the value passed into action.payload and 'Rejected' as action.error.message
- if the promise failed and was not handled with rejectWithValue, dispatch the rejected action with a serialized version of the error value as action.error
- Return a fulfilled promise containing the final dispatched action (either the fulfilled or rejected action object)

Promise Lifecycle Actions#

createAsyncThunk will generate three Redux action creators using createAction: pending, fulfilled, and rejected. Each lifecycle action creator will be attached to the returned thunk action creator so that your reducer logic can reference the action types and respond to the actions when dispatched. Each action object will contain the current unique requested and args values under action.meta.

The action creators will have these signatures:

```
interface SerializedError {
  name?: string
  the message?: string
  code?: string
  stack?: string
}

interface PendingAction<ThunkArg> {
  type: string
  payload: undefined
  meta: {
    requestId: string
    arg: ThunkArg
  }
}

interface FulfilledAction<ThunkArg, PromiseResult> {
  type: string
  payload: PromiseResult
  meta: {
    requestId: string
```

```
                    arg: ThunkArg
                  }
                }

                interface RejectedAction<ThunkArg> {
                  type: string
   aborted: b

         interface RejectedWithValueAction<ThunkArg, RejectedValue> {
           type: string
           payload: RejectedValue
  error: { message: 'Rejected' }
  meta: {
    requestId: string
    arg: ThunkArg
    aborted: boolean
  }
}

type Pending = <ThunkArg>(
  requestId: string,
  arg: ThunkArg
) => PendingAction<ThunkArg>

type Fulfilled = <ThunkArg, PromiseResult>(
  payload: PromiseResult,
  requestId: string,
  arg: ThunkArg
) => FulfilledAction<ThunkArg, PromiseResult>

type Rejected = <ThunkArg>(
  requestId: string,




  arg: ThnkArg
) => RejectedAction<ThunkArg>

type RejectedWithValue = <ThunkArg, RejectedValue>(
  requestId: string,
  arg: ThunkArg
) => RejectedWithValueAction<ThunkArg, RejectedValue>
```

To handle these actions in your reducers, reference the action creators in createReducer or createSlice using either the object key notation or the "builder callback" notation. (Note that if you use TypeScript, you should use the "builder callback" notation to ensure the types are inferred correctly):

Let's Take A Look Into the Previous Session

- we created a provider
- provider require a store so we created a store
- redux toolkit have aconfigureStore function which we will call .this to require a reducer so we need to pass a counterReducer
- for counterReducer, we have created counterSlice because it becomes more difficult to create an action separately, reducer and then managing them separately when you are creating a lot of code as in reduct react that is why we are using this toolkit of redux for saving code complications
- so call createSlide and provide its properties as name, initial state, reducers for increment, decrement, increment by the amount of async. since reducer is a pure function so we need to clone any update for it.
- when we create counterSlice it create an action "counterSlice.action" that will return {increment,decrement,incremenByAmount} as well as return reducers
- in-store we will call this reducer
- in counter.js we have two button increment, decrements, and the type text box that will call for the set value
- we are calling dispatch that will return the dispatch function i.e increment dispatch it will request to store for it and go on.....

App.js

store.js

```
import {configureStore}from'@reduxjs/toolkit';
export const store=configureStore({
   reducer:{
      counter:counterReducers,
                    }
            })
```

counterSlice.js

```
import {createSlice}from"@redux.js/to
export const counterSlice=createSlice
name:"counter",
initialState:{
  count:0
 }
 reducers:{
   increment:(state)=>{
     state.count=state.count++
    }'
decrement:state=>{
     sate.count--;
   },
   incrementByAmount:{state,action)=>{
    state.count+=action.payload
   }
  }
});
export const
{increment,decrement,incrementByAmount}=co
unterSlice.action
export const {reducer}=counterSlice.reducer;
```

Counter.js

Counter.js

```
import React, { useState } from 'react';
import {useDispatch, useSelector} from './react-redu
import { increment, decrement, incrementByAmount
'../store/action';

function Counter() {
   const[value,setValue]=useState(0);
const dispatch=useDispatch();
const counter= useSelector((state)=>{
            return state.counter.count

            <div>
 });
return
  (<div>
     <div>Counter:{counter}</div>
<div>
              <button onClick={()=>{
          }}>Increment</button>
          <br/>

<button onClick={()=>{
            dispatch( decrement());

          }}>Decrement</button>
       </div>
       <div>
            function Counter() {
          <br/>
          <button onClick={()=>{
              setCounter(counter+ Number(value))


dispatch(incrementByAmount(Number(value)))
          }}>Increment By Value</button>
       </div>
    </div>
```

- In case async a way we need only to add counterUpdate=createAsyncThunk function
- Add extra reducers for functionalities
- create asyncthunk which require an async property whatever data we will pass in these properties that will be store in action. Payload

CounterSlice.js

```
import {createSlice,createAsyncThunk}from"@redux.js/toolkit";
export const counterUpdate=createAsyncThunk(
  "counter/counterupdate",
  async(value,thunkAPI)=>{
    const response=fetch ("http://localhost:3000/api/updaetcounter)
    const data =await response.jason();
   return data
}
 export const counterSlice=createSlice({
name:"counter",
initialState:{
  count:0,
 isloading:false,
 error:" ",
 }
 reducers:{
   increment:(state)=>{
     state.count=state.count++
    }'
    decrement:state=>{
      sate.count--;
    },
    incrementByAmount:{state,action)=>{
     state.count+=action.payload;
      state.isLoading=false;
    }
  },
  extraReducer:{
   [counterUpdate.fulfilled]:(state,action)=>{
      state.count+=action.payload
   },
 [counterUpdate.pending]:(state)=>{
      state.isLoading=true;
   },
[counterUpdate.rejected]:(state)=>{
      state.isLoading=false;
      sate.error="Error in update counter"
   },
 }

});
export const {increment,decrement,incrementByAmount}=counterSlice.action
export const {reducer}=counterSlice.reducer
```

- add another button in an increment by value "increment from server" in counter.js
- in dispatch call counterUpdate
-

counter.js

```
React, { useState } from 'react';
import {useDispatch, useSelector} from './react-redux';
import { increment, decrement, incrementByAmount } from '../store/action';

function Counter() {
   // const [counter, setCounter] = useState(0);
    const[value,setValue]=useState(0);
const dispatch=useDispatch();
const counter= useSelector((state)=>{
              return state.counter.count

              <div>
 });
return
  (<div>
      <div>Counter:{counter}</div>
<div>
                <button onClick={()=>{
            //   setCounter(counter+1);
             //  dispatch( {type:"INCREMENT"} );
               //  dispatch( {type:"INCREMENT"} );
               dispatch(increment());
            }}>Increment</button>
            <br/>

<button onClick={()=>{
                // setCounter(counter-1);
               // dispatch( { type: "DECREMENT" } );
             dispatch( decrement());

            }}>Decrement</button>
        </div>
        <div>
              function Counter() {
            <br/>
            <button onClick={()=>{
                setCounter(counter+ Number(value))

                // dispatch( {type: "INCREMENT_BY_AMOUNT", payload:Number(value) } )
                  dispatch(counterUpdate())
            }}>Increment from server</button>
```

2. **<u>Building API Mocking Servers with Mirage</u>**

- **install mirage**

- # Using npm
  npm install --save-dev miragejs

- # Using Yarn
  yarn add --dev mirage

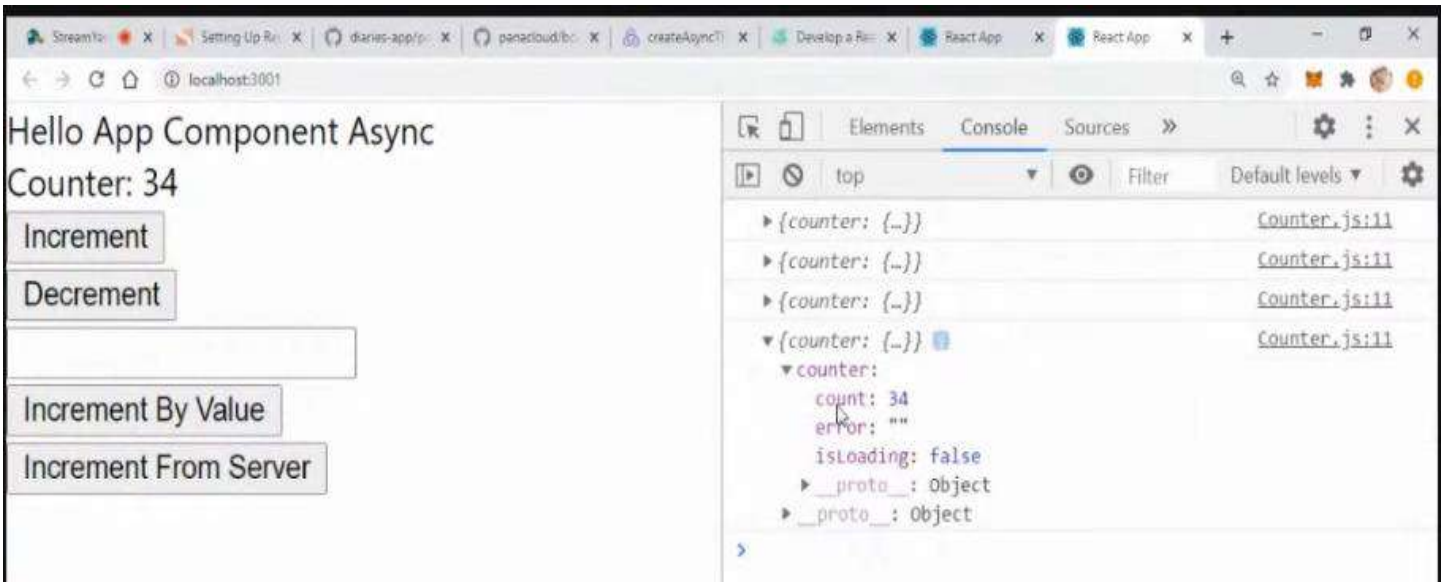
```
src/server.js:
import { createServer, Model } from "miragejs";

export function makeServer({ environment = "test" } = {}) {
  let server = newServer({
     environment,
     route(){
       this.namespace="api"
        this.get("updatecounter",(=>{
            return 34;
      };
  });
```

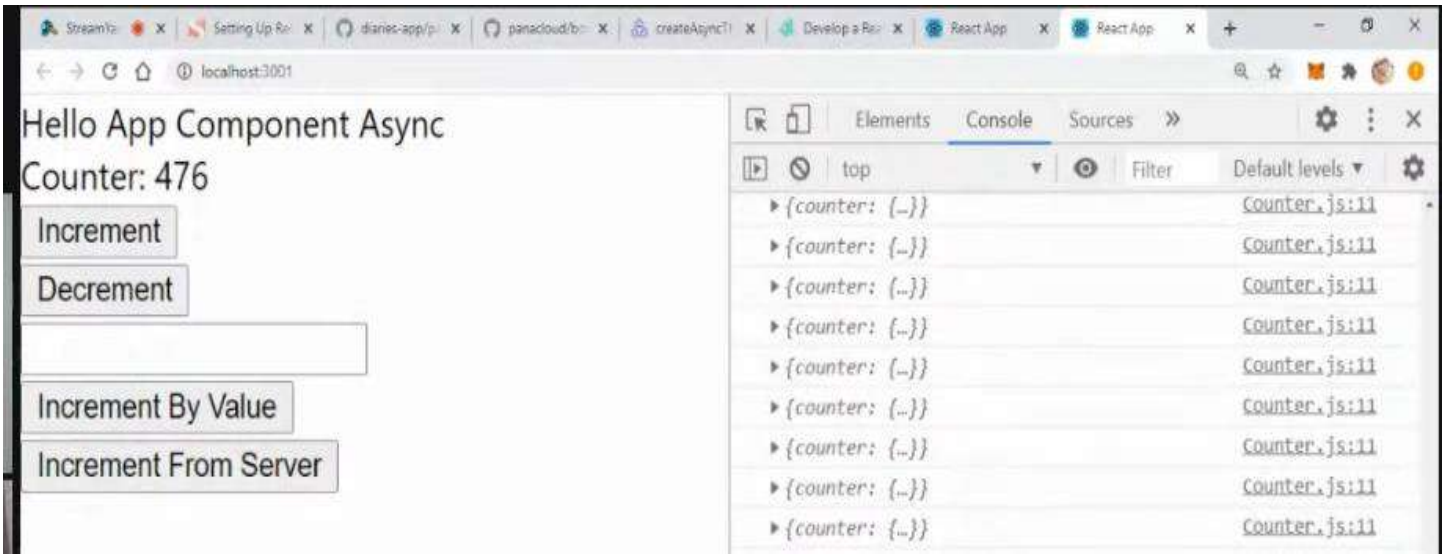
- in index.js call this server

```
makeServer();
```

```
index.js
import React from 'react';
import *as serviceWorker from './serviceWorker';
import {provider} from 'react-redux';

  makeServer();
ReactDOM.render(
   <React.StricMode>
     <provider store{store}>
     <App/.
   </provider>
</React.StrictMode>,
document.getElementById('root')
```

- Now go to the browser and check it
- so we are receiving value from the server counting 34
- when we will update it will be updated as shown





- when request isin progress on server to show uploading , it must dipslay loader
- in counter.js provide the function of loader and in condition when requestis processising it must display loading

```
const {counter,isLoading}= useSelector((state)=>{
        return{
       counter: state.counter.count,
       isLoading:state.counter.isLoading
      }
 });
```

counter.js

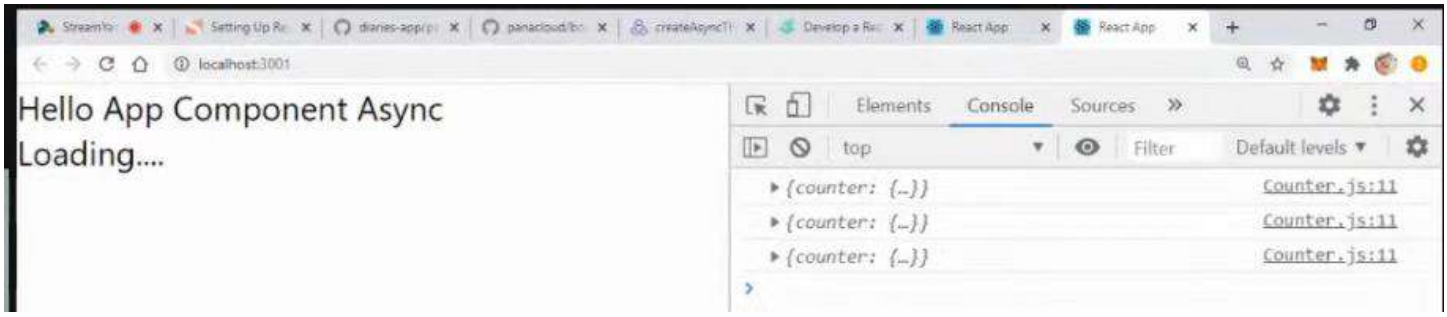
```
function Counter() {
   // const [counter, setCounter] = useState(0);
    const[value,setValue]=useState(0);
const dispatch=useDispatch();
const {counter,isLoading}= useSelector((state)=>{
        return{
       counter: state.counter.count,
       isLoading:state.counter.isLoading
      }


 });
 if(isLoading)return<div>Loading</div>
return
  (<div>
      <div>Counter:{counter}</div>
<div>
                <button onClick={()=>{
            //   setCounter(counter+1);
             //  dispatch( {type:"INCREMENT"} );
               //  dispatch( {type:"INCREMENT"} );
              dispatch(increment());
           }}>Increment</button>
           <br/>


<button onClick={()=>{
                // setCounter(counter-1);
              // dispatch( { type: "DECREMENT" } );
             dispatch( decrement());

           }}>Decrement</button>
        </div>
        <div>
              function Counter() {
           <br/>
           <butto
```

```
                setCounter(counter+ Number(value))

                // dispatch( {type: "INCREMENT_BY_AMOUNT", payload:Number(value) } )
                  dispatch(counterUpdate())
           }}>Increment from server</button>
        </div>
    </div>
```

- in service.js provide the property of timing so that "Loading..." will stay for one second when the request is in progress

```
service.js

import { createServer, Model } from "miragejs";

export function makeServer({ environment = "test" } = {}) {
  let server = newServer({
    environment,
    route(){
      this.namespace="api"
       this.get("updatecounter",(=>{
          return 34;
     },{
        timing:1000

  });
```

- Whenever you will click to any of button of increment, decrement, increment byAmount , meanwhile of this request progressing a loading option will be displayed over the screen



this is how we implement async functionality in-app with redux…….

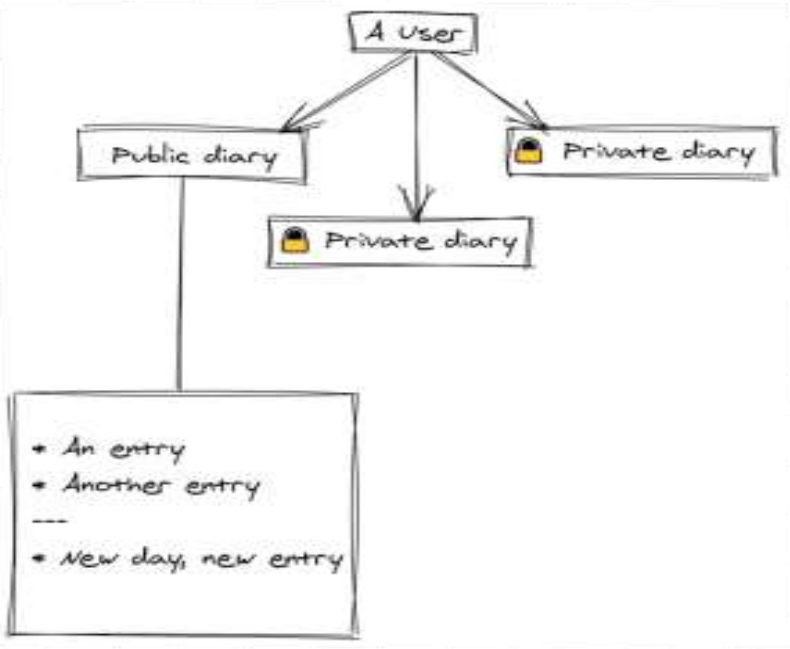
3. Setting Up Redux For Use In A Real-World Application

DESCRIBING THE APPLICATION'S INITIAL STATE

Let's go over our application's requirements in detail. The application will allow authenticated users to create or modify existing diaries. Diaries are private by default, but they can be made public. Finally, diary entries will be sorted by their last modified date.

This relationship should look something like this:

An Overview of the Application's Data Model



- Armed with this information, we can now model our application's state. First, we will create an interface for each of the following resources: User, Diary, and DiaryEntry. Interfaces in Typescript describe the shape of an object.

- Go ahead and create a new directory named interfaces in your app's src sub-directory:

  cd src && mkdir interfaces

- Next, run the following commands in the directory you just created:

  - ✓ touch entry.interface.ts
  - ✓ touch diary.interface.ts
  - ✓ touch user.interface.ts

- create user.interface.ts
- give the following prorerty of users as
  - id
  - username
  - email
  - password
  - diaryIds

```
src/interfaces/user.interface.ts
export interface User {
    id?: string;
    username: string;
    email: string;
    password?: string;
    diaryIds: string[] | null;
}
```

- now create diary.interface.ts and provide the entities you want

```
src/interfaces/diary.interface.ts
export interface Diary {
    id?: string;
    title: string;
    type: 'private' | 'public';
    createdAt?: string;
    updatedAt?: string;
    userId?: string;
    entryIds: string[] | null;
  }
```

- create entry.interface.ts that will have diaryId along with other entities as shown

```
src/interfaces/entry.interface.ts
export interface Entry {
    id?: string;
    title: string;
    content: string;
    createdAt?: string;
    updatedAt?: string;
    diaryId?: string;
  }
```

- add mirage
- install server
- create server.ts
- it contains function handle error
- setupserver that includes models that's the mirage shows up the actual configuration this contains one to many and many to many relationships
- factories provide the initial data available by defaults
- in route use any uri you want to use
- here we are using only two APIs
  - login
  - signup

src/services/mirage/server.ts

```
import { Server, Model, Factory, belongsTo, hasMany, Response } from 'miragejs';
import user from './routes/user';

export const handleErrors = (error: any, message = 'An error ocurred') => {
  return new Response(400, undefined, {
    data: {
      message,
      isError: true,
    },
  });
};

export const setupServer = (env?: string): Server => {
  return new Server({
    environment: env ?? 'development',

    models: {
      entry: Model.extend({
        diary: belongsTo(),
      }),
      diary: Model.extend({
        entry: hasMany(),
        user: belongsTo(),
      }),
      user: Model.extend({
        diary: hasMany(),
      }),
    },

    factories: {
      user: Factory.extend({
        username: 'test',
        password: 'password',
       email: 'test@email.com',
   s: (server): any => {
    server.create('user');
  },

  routes(): void {
    this.urlPrefix = 'https://diaries.app';

    this.post('/auth/login', user.login);
    this.post('/auth/signup', user.signup);
  },
```

- here we are using a dummy server that will use interface function with tokens

- we are using the login function as well

src/services/mirage/routes/user.ts

```
import { Response, Request } from 'miragejs';
import { handleErrors } from '../server';
import { User } from '../../../interfaces/user.interface';
import { randomBytes } from 'crypto';

const generateToken = () => randomBytes(8).toString('hex');

export interface AuthResponse {
  token: string;
  user: User;
}

const login = (schema: any, req: Request): AuthResponse | Response => {
  const { username, password } = JSON.parse(req.requestBody);
  const user = schema.users.findBy({ username });
  if (!user) {
    return handleErrors(null, 'No user with that username exists');
  }
  if (password !== user.password) {
    return handleErrors(null, 'Password is incorrect');
  }
  const token = generateToken();
  return {
    user: user.attrs as User,
    token,
```

```
index.tsx
if (process.env.NODE_ENV === 'development') {
  setupServer();
}
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import App from './app/App';
import * as serviceWorker from './serviceWorker';
import { setupServer } from './services/mirage/server';
import { Provider } from 'react-redux';
import store from './store';

if (process.env.NODE_ENV === 'development') {
  setupServer();
}
```

```
ReactDOM.render(
 <React.StrictMode>
   <App />
   <Provider store={store}>
     <App />
   </Provider>
 </React.StrictMode>,
 document.getElementById('root')
)
```

- Create a file named api.ts under src/services and add the following code to it:

```
import Swal, { SweetAlertIcon } from 'sweetalert2';

export const showAlert = (titleText = 'Something happened.', alertType?: SweetAlertIcon): void
=> {
  Swal.fire({
    titleText,
    position: 'top-end',
    timer: 3000,
    timerProgressBar: true,
    toast: true,
    showConfirmButton: false,
    showCancelButton: true,
    cancelButtonText: 'Dismiss',
    icon: alertType,
    showClass: {
      popup: 'swal2-noanimation',
      backdrop: 'swal2-noanimation',
    },
    hideClass: {
      popup: '',
      backdrop: '',
    },
  });
```

util.ts

n this file, we are exporting an Axios instance modified to include our app's API url, https://diaries.app. We have configured an interceptor to handle success and error responses, and we display error messages using a sweetalert toast which we will configure in the next step.

- Create a file named util.ts in your src directory

util.ts

```
import Swal, { SweetAlertIcon } from 'sweetalert2';
export const showAlert = (titleText = 'Something happened.', alertType?: SweetAlertIcon): void
=> {
  Swal.fire({
titleText,
```

```
    position: 'top-end',
      timer: 3000,
      timerProgressBar: true,
      toast: true,
      showConfirmButton: false,
      showCancelButton: true,
      cancelButtonText: 'Dismiss',
      icon: alertType,
      showClass: {
        popup: 'swal2-noanimation',
        backdrop: 'swal2-noanimation',
      },
      hideClass: {
        popup: '',
        backdrop: '',
      },
    });
  };
```

This file exports a function that displays a toast whenever it is invoked. The function accepts parameters to allow you set the toast message and type. For example, we are showing an error toast in the Axios response error interceptor like this:

showAlert(response.data?.data?.message, 'error');

- create authSlice

```
src/features/auth/userSlice.ts
import { createSlice, PayloadAction } from '@reduxjs/toolkit';
import { User } from '../../interfaces/user.interface';

const user = createSlice({
  name: 'user',
  initialState: null as User | null,
  reducers: {
    setUser(state, { payload }: PayloadAction<User | null>) {
      return state = (payload != null) ? payload : null;
    },
  },
});

export const { setUser } = user.actions;
export default user.reducer
```

In this file, we're creating a slice for the auth property of our app's state using the createSlice() function introduced earlier. The reducers property holds a map of reducer functions for updating values in the auth slice. The returned object contains automatically generated action creators and a

single slice reducer. We would need to use these in other files so, following the “ducks pattern”, we do name exports of the action creators, and a default export of the reducer function.

- create a file named userSlice.ts in the auth directory and add the following code to it:

userSlice.ts

```
import { createSlice, PayloadAction } from '@reduxjs/toolkit';
import { User } from '../../interfaces/user.interface';

const user = createSlice({
  name: 'user',
  initialState: null as User | null,
  reducers: {
    setUser(state, { payload }: PayloadAction<User | null>) {
      return state = (payload != null) ? payload : null;
    },
  },
});
```

util.ts

```
export const { setUser } = user.actions;
export default user.reducer;
```

This creates a slice reducer for the user property in our the application’s store. The setUser reducer function accepts a payload containing user data and updates the state with it. When no data is passed, we set the state’s user property to null.

- To put it all together, create a file named rootReducer.ts in the src directory with the following content

rootReducer.ts

```
import { combineReducers } from '@reduxjs/toolkit';
import authReducer from './features/auth/authSlice';
import userReducer from './features/auth/userSlice';
import diariesReducer from './features/diary/diariesSlice';
import entriesReducer from './features/entry/entriesSlice';
import editorReducer from './features/entry/editorSlice';

const rootReducer = combineReducers({
  auth: authReducer,
  diaries: diariesReducer,
  entries: entriesReducer,
  user: userReducer,
  editor: editorReducer,
});

export type RootState = ReturnType<typeof rootReducer>;
export default rootReducer;
```

In this file, we've combined our slice reducers into a single root reducer with the combineReducers() function. We've also exported the RootState type, which will be useful later when we're selecting values from the store. We can now use the root reducer (the default export of this file) to set up our store.

- Create a file named store.ts with the following contents:

store.ts

```
import { configureStore } from '@reduxjs/toolkit';
import rootReducer from './rootReducer';
import { useDispatch } from 'react-redux';

const store = configureStore({
  reducer: rootReducer,
});

type AppDispatch = typeof store.dispatch;
export const useAppDispatch = () => useDispatch<AppDispatch>();
export default store;
```

With this, we've created a store using the configureStore() export from Redux toolkit. We've also exported an hook called useAppDispatch() which merely returns a typed useDispatch() hook.

util.ts

Next, update the imports in index.tsx to look like the following:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './app/App';
import * as serviceWorker from './serviceWorker';
import { setupServer } from './services/mirage/server';
import { Provider } from 'react-redux';
import store from './store';
// ...
```

Session 18(B): Redux Toolkit and Mirage

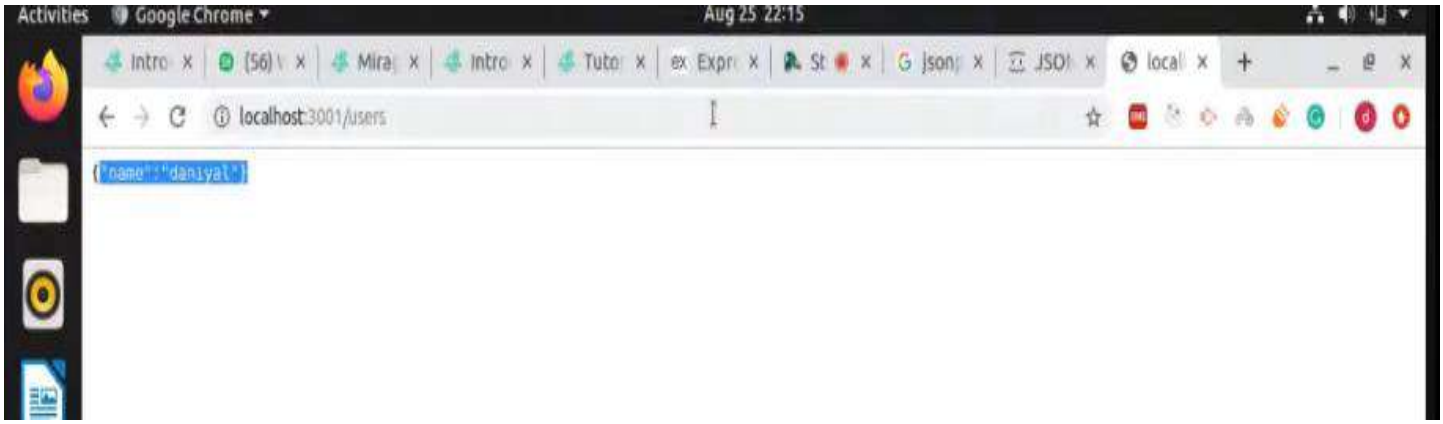
1. **<u>How We Can Create Our Server On Front End Is Introduce By Mirages</u>**

- hence we can create a fake server on the client site to get the real environment of that application as a result of which we are not needed backend process or request

- create index.js express
- here we two API on the front end
- if we don't have a backend and we have req to built front end application this is the point f mirage that we can create our API on the front end

```
 const port =3001

app.get('user'(req,res=>{
  res.send('Hello World')
})
```

```
const express =require('express')
const app=express()
const port =3001

app.get('user'(req,res=>{
  res.send('Hello World')
 })
app.get('/users,(re.res)=>{
  res.send({name:"daniyal"})
})

app.listen(port, ()=>{
  console.log('Example app Listenning at htpp
```



2. **<u>Build Complete frontend Features Even If Your API Doesn't Exist With the Helps Miragejs</u>**

- create project mirageapi
- create an app on the terminal as shown by the following commands

```
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop$ mkdir mirageapi
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop$
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop$ cd mirageapi/
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/mirageapi$ sudo npx crea
te-react mirageapiapp
[sudo] password for daniyal:
[...................] / rollbackFailedOptional: verb npm-session f72deb01af6ca94
```

- install dependencies miragejs . you can install it via yarn or npm

  nmp install –sve-dev miragejs

- install server to create API
- call create a server that can be imported from miragejs

```
# Using npm
npm install --save-dev miragejs

# Using Yarn
yarn add --dev miragejs
```

- create a folder of mirage
- inside this folder create an index.js file where you will have to create APIs
- we will use these for the method according to our requirement on the back end as
  - this.get()
  - this.post()
  - this.put()
  - this.delete()
- now we are using the get method
- create the array and return the data
- simply import createServer from miragejs
- create a function and pass array in it
- create route

this.get (to retrieve the data).inside it getTodos and

index.js

```
import {createServer} from "miragejs";
export default function(){text:"./fakeapi";
    let arr={{id:"1"
   crreateServer({
     routes(){
   this.namespace="fakeapi"
   this..get("/getTodos",{todos:arr)

     }
    })
  }
```

- create a component folder
- create Todo.js in it

Todo.js

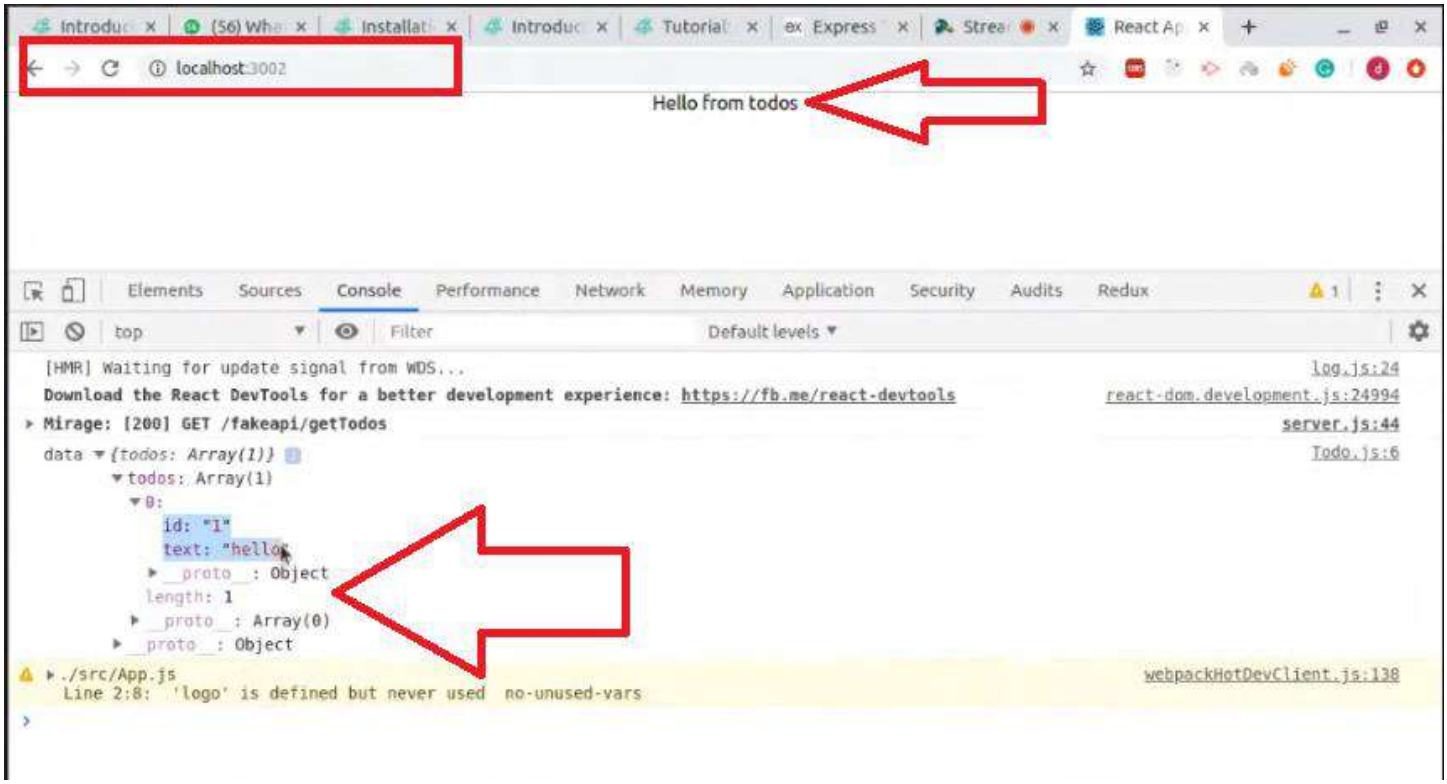
```
import React,{useEffect} from 'react';
let baseUrl="fakeapi"

 export const Todos=()=>{
 useEffect(()=>{
  fetch(`/$baseUrl}/getTodos`)
  .then(res=>ress.json())
  .then(data=>console.log('data',data))
  return(
  <div>
    Hello from todos
  </di>
  )
```

- Create App.js
- import components from todos in app.js

App.js

```
import React from'react';
import logo from './logo.svg';
import MirageServer from './miragejs
import './App.css';
import {} from "./components/Todo';
function App(){
  return(
 <div className="App">
    <Todos/>
   );
}

export default App;
```

- run this application now we have all same data as we retrieve I Github
- hence from the server we are getting the data



- now using miragejs use the function of the post and push the array

- console.log the request
- whatever the data we will send we will get in this request

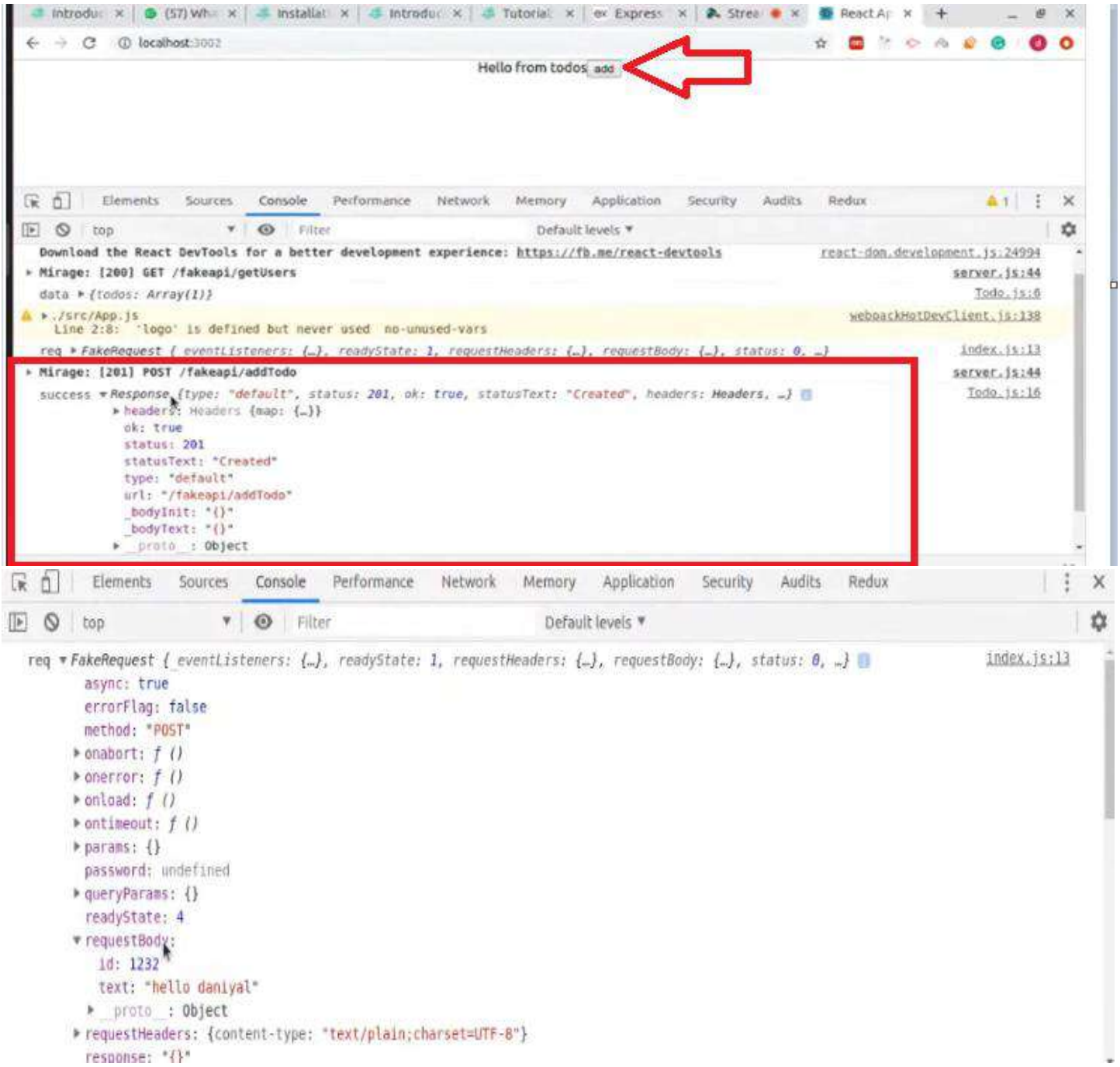
index.js

```
import {createServer} from "miragejs";
export default function(){, text:"/fakeapi";
    let arr={{id:"1",text:"hello"}]
    let users=[{name:"dniyal",age:24}]
   crreateServer({
     routes(){
   this.namespace="fakeapi"
   this.get("/getTodos",{todos:arr)

   this.get("/getTodos",{todos:arr)
   this.post("/addTodo",(_,req)=>{
    console.log('req',req)
   arr.push(req.requestBody)
```

- create a TodoHandler
- create add button

```
                <button onClick=>{addTodoHandler"} >add</button>
import React,{useEffect} from 'react';
let baseUrl="fakeapi"

 export const Todos=()=>{
 useEffect(()=>{
  fetch(`/$baseUrl}/getTodos`)
  .then(res=>ress.json())
  .then(data=>console.log('data',data))
   const AddHandler=()=>
     fetch(`/${baseUrl`}/addTodos`,{
         method:"POST",
         body:{id:1232,text:"text"}
        }).then=>{
          console.log("success",res)
       },catch(error=>
          console.log("error addtodo", error)  })
}
  return(
  <div>
    Hello from todos
    <button onClick=>{addTodoHandler"} >add</button>
  </div>
```

- Now go to the browser and click to add button you will get the all add data of fakeapi request as shown

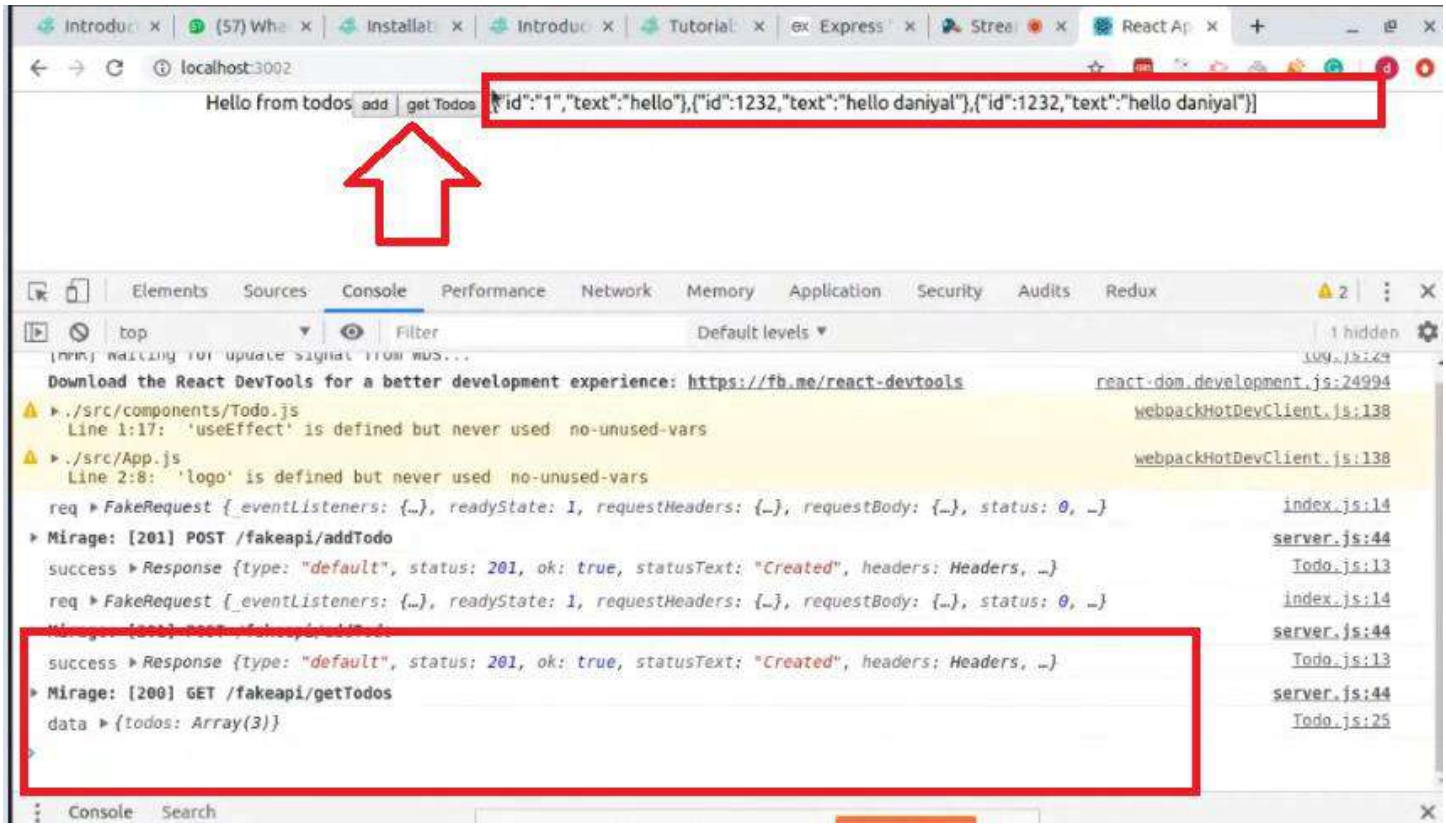


- Now remove useffect hook
- create the handler to get the Todos
- whenever we click add button then we push data into the array
- create getTodo button whenever we click on this button we get the data of getTodos

- add JSoN.stringify(data)
- then run the application and look out the output on your browser

Todos.js

```
import React,{useEffect,useState} from 'react';
let baseUrl="fakeapi"

 export const Todos=()=>{
 let[data,setData]=useStae
  fetch(`/$baseUrl}/getTodos`)
  .then(res=>ress.json())
  .then(data=>console.log('data',data))
   const AddTodoHandler=()=>
     fetch(`/${baseUrl`}/addTodos`,{
         method:"POST",
         body:{id:1232,text:"text"}
        }).then=>{
          console.log("success",res)
       }),catch(error=>
          console.log("error addtodo", error)
      })
  const getDataHandler=()=>{
   fetch(`/$baseUrl}/getTodos`)
  .then(res=>ress.json())
  .then(data=>{
     setData(data)
  console.log('data',data.todos)})
  }
  return(
  <div>
    Hello from todos
    <button onClick=>{addTodoHandler"} >add</button>
   <button onClick=>{getdataHandler"} >add</button>
   {
      JSoN.stringify(data)
  </div>
  )
```


localhost:3002
Hello from todos
add
get Todos
[{"id":"1","text":"hello"},{"id":1232,"text":"hello daniyal"},{"id":1232,"text":"hello daniyal"}]
Elements
Sources
Console
Performance
Network
Memory
Application
Security
Audits
Redux
top
Filter
Default levels
Download the React DevTools for a better development experience: https://fb.me/react-devtools
react-dom.development.js:24994
./src/components/Todo.js
Line 1:17: 'useEffect' is defined but never used no-unused-vars
webpackHotDevClient.js:138
./src/App.js
Line 2:8: 'logo' is defined but never used no-unused-vars
webpackHotDevClient.js:138
req ▸ FakeRequest {_eventListeners: {…}, readyState: 1, requestHeaders: {…}, requestBody: {…}, status: 0, …}
index.js:14
Mirage: [201] POST /fakeapi/addTodo
server.js:44
success ▸ Response {type: "default", status: 201, ok: true, statusText: "Created", headers: Headers, …}
Todo.js:13
req ▸ FakeRequest {_eventListeners: {…}, readyState: 1, requestHeaders: {…}, requestBody: {…}, status: 0, …}
index.js:14
server.js:44
success ▸ Response {type: "default", status: 201, ok: true, statusText: "Created", headers: Headers, …}
Todo.js:13
Mirage: [200] GET /fakeapi/getTodos
server.js:44
data ▸ {todos: Array(3)}
Todo.js:25
Console
Search

Session 19: Forms_with_Formik_and_Yup

1. Working With Forms

working with forms is not as easy as most of the students consider it requires all these steps to be passed through for that purpose

- ✓ when you are developing a form you need to manage the input box
- ✓ once you create the correct input you need to validate
- ✓ if validation is correct you need to collect out the error and hide them
- ✓ then you have collected this information and send to the server

- ➢ open up your terminal
- ➢ create userForm.js

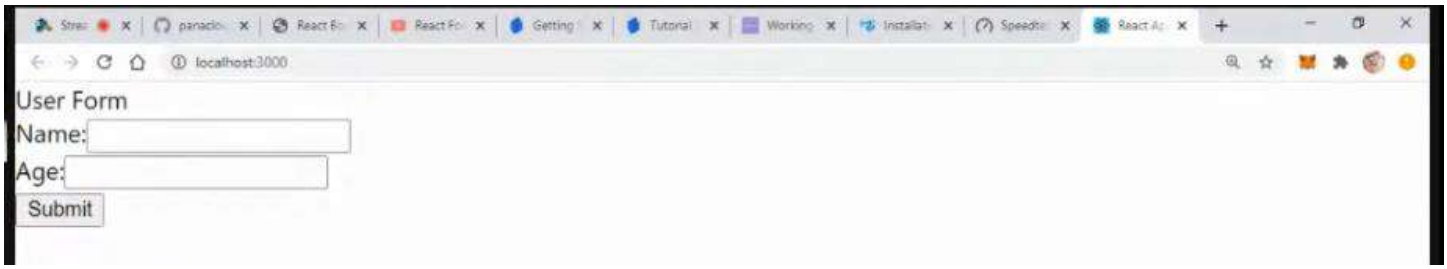
userForm.js

```
import React from'react';
function UserForm(){
  return(
   <div>
     <div> User Form</div>
     <div>
        <label htmlFor="name">Name:</label>
        <input id="name" type="text"/>
     </div>
     <div>
        <label htmlFor="age"Age:</label>
        <input id type="number/>
    </div>
  </div>
        < button type="submit">Submit</button>
  </div
 </div>
 )}
export default UserForm;
```

App.js

```
import React from 'react';
import './App.css';
function App(){
  return(
    <div>
      <UserForm/>
    </div>
  );
 }
export default App;
```

- now go to the browser and check out the output

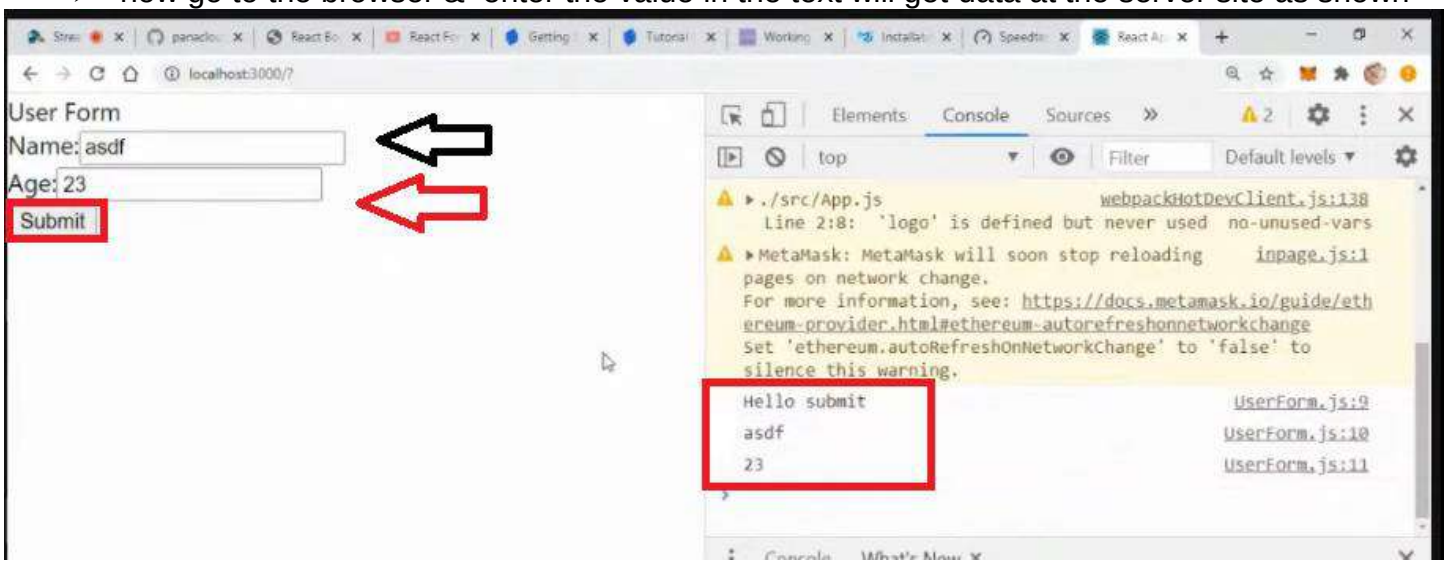


- collect all information by using the usref function as so that if you change functionality it will b updated automatically
- create forms
- use input buttons
- create an input text field
- implement submit button

userForm.js

```
import React from'react';
function UserForm(){
    const name=useRef();
    const age=useRef();
    const handleSubmit=(e)={
      e.preventDefault();
      console.log("Helo Submit");
      console.log(name.current.value);
     console.log(name.current.value);
     console.log(age.current.value);

   return(
    <div>
      <div> User Form</div>
    <form onSubmit={handleSubmit}>
      <div>
        <label htmlFor="name">Name:</label>
       < input id="name" ref={name}type="text"/>
      </div>
      <div>
        <label htmlFor="age"Age:</label>
        <input id="age" ref={age} type="number"/>
     </div>
   </div>
         < button type="submit">Submit</button>
   </div
  </div>
 );
}
export default UserForm
```

- now go to the browser & enter the value in the text will get data at the server site as shown



- for any validation create

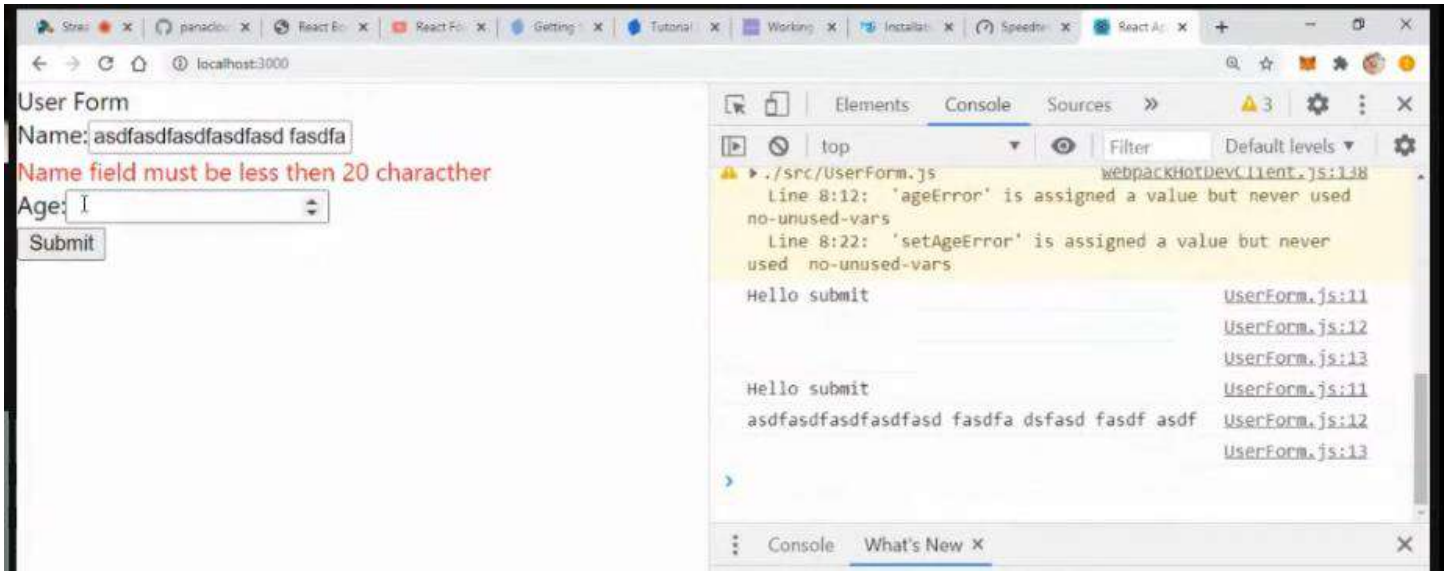
```
const[nameError,setNameError]=useState("");

const[ageError,setAgeError]=useState("");
```

- set condition of name error if it is less than of 20 characters or any t incorrect entry

```
import React from'react';
function UserForm(){
    const name=useRef();
    const age=useRef();
   const[nameError,setNameError]=useState("");
   const[ageError,setAgeError]=useState("");
    const handleSubmit=(e)={
       e.preventDefault();
       console.log("Helo Submit");
       console.log(name.current.value);
      console.log(name.current.value);
      console.log(age.current.value);

if (name.current,value{
   setNameError("name field is required");
}
   else if(name.current.value.length>20){
     setNameError("name field must be less than 20 cahracter");

   return(
    <div>
      <div> User Form</div>
    <form onSubmit={handleSubmit}>
      <div>
```

```
<label htmlFor="name">Name:</label>
        < input id="name" ref={name}type="text"/>
        <div style={{color:"red"}}>NameError</div>
      </div>
      <div>
         <label htmlFor="age"Age:</label>
         <input id="age" ref={age} type="number"/>
     </div>
   </div>
          < button type="submit">Submit</button>
   </div
 </div>
 );
}
export default UserForm
```

- now go to the browser and enter more than 20 characters it will display an error as shown



- let's use formiks for validation of the message

2. Overview: What is Formik?

Formik is a small group of React components and hooks for building forms to React and React Native. It helps with the three most annoying parts:

- Getting values in and out of form state
- Validation and error messages
- Handling form submission

By colocating all of the above in one place, Formik keeps things organized--making testing, refactoring, and reasoning about your forms a breeze.

- create a file of UserFromwithFormik.js
- all things will be the same as in previous but
- add formikhooks
- provide initial values and submit

User Foom with Formik.js

```
import React from 'react';
    import { useFormik } from 'formik';

function UserForm(){
  const formik=userFormik({
     initilValues:{
       name:"",
       age:0
}
    onSubmit: values => {
    console.log(values);
   }
});
      <div> User Form</div>
    <form onSubmit={formik.handleSubmit}>
      <div>
<label htmlFor="name">Name:</label>
        < input id="name" type="text"/>
        onChange={formiks.handleChange}
        value={formik.value.name}
       />
       <div style={{color:"red"}}</div>

      </div>
      <div>
         <label htmlFor="age"Age:</label>
         <input id="age" type="number"onChange(handleChange}value={formik.values.age/>
     </div>
   </div>
          < button type="submit">Submit</button>
   </div
 </div>
 );
}
export default UserForm;
```

- formik provide us with validation

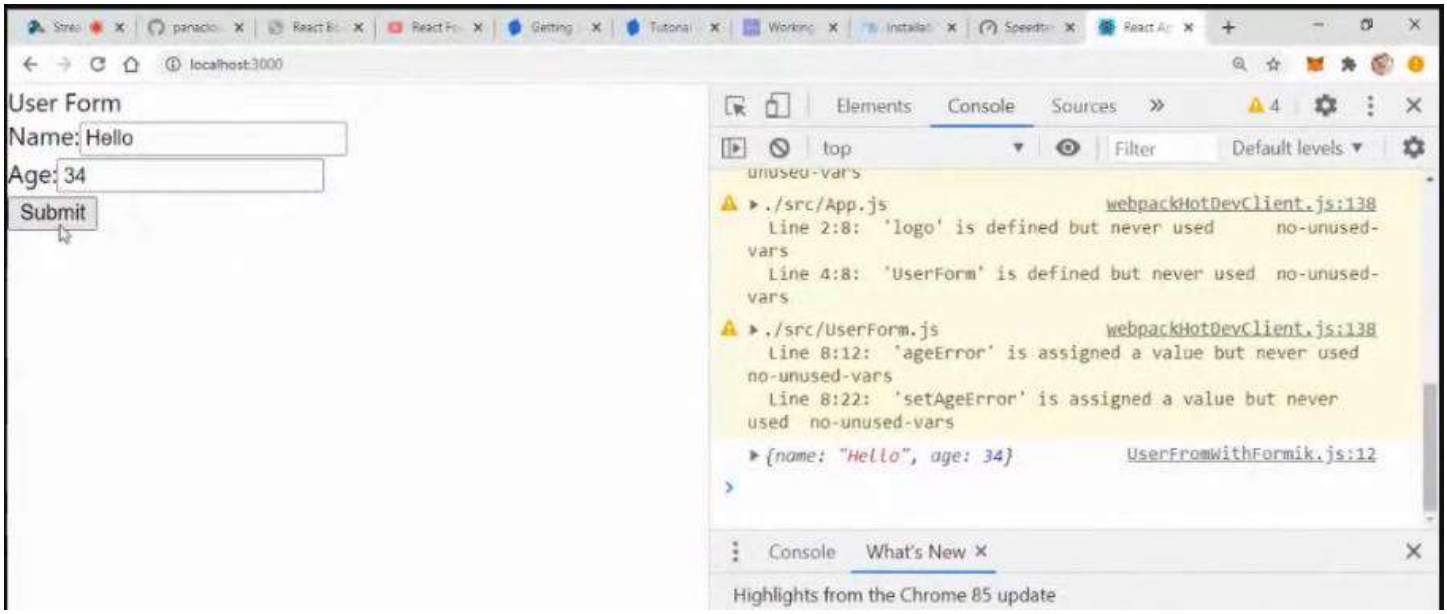
submit the button if it not validate then it should display an error

User Form with Formik.js

```
import React from 'react';


  import { useFormik } from 'formik';

  function UserForm(){
    const formik=userFormik({
       initilValues:{
         name:"",
         age:0
  }
      onSubmit: values => {
      console.log(values);
     },
     validate:(values)=>{
      const error={};
        if (!Values.name){
             error.name="Name field is required";
       }
      else if (Value.name.length>20{
          error.name="Name field must be less than 20 characters";
       }
          return error;
     }

  });
        <div> User Form</div>
      <form onSubmit={formik.handleSubmit}>
        <div>
  <label htmlFor="name">Name:</label>
          < input id="name" type="text"/>
          onChange={formiks.handleChange}
          value={formik.value.name}
         />


            <div style={{color:"red"}}>{formik.error.name}</div>
               null
        </div>
        <div>
           <label htmlFor="age"Age:</label>
           <input id="age" type="number"onChange(handleChange}value={formik.values.age
       </div>
     </div>
           < button type="submit">Submit</button>
     </div
   </div>
```

App.js

```
import React from 'react';
import './App.css';
import UserFromWithFormik from './UserFromWithFormik
function App(){
  return(
    <div>
      <UserFormWithFormik/>
    </div>
  );
 }
export default App;
```

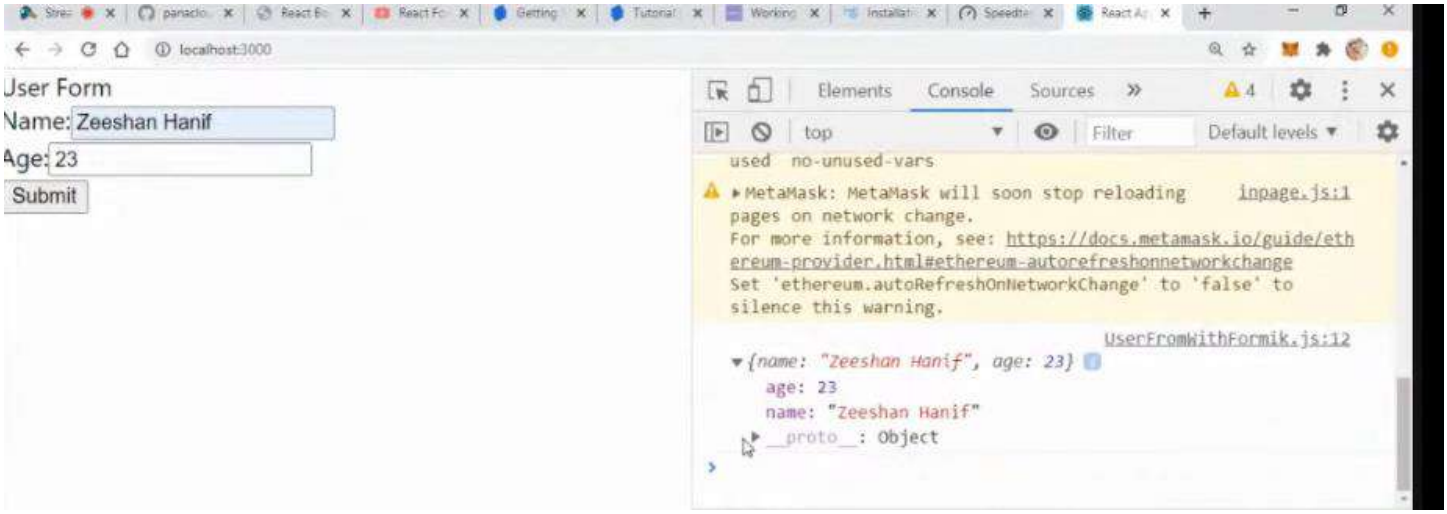


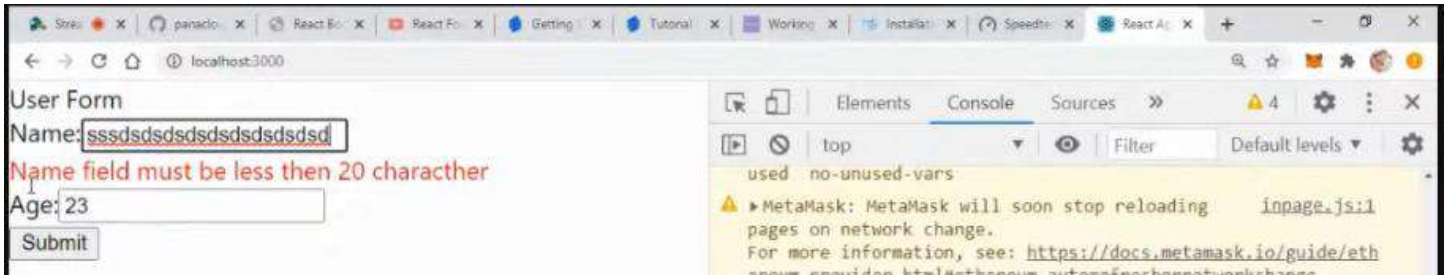
- formik provide us with validation
- submit the button if it not validate then it should display an error
- initialize value, create submit, provide a condition for validation of name as shown

User Form with Formik.js

```
import React from 'react';
import { useFormik } from 'formik';

function UserForm(){
  const formik=userFormik({
     initilValues:{
       name:"",
       age:0
}
    onSubmit: values => {
    console.log(values);
   },
   validate:(values)=>{
    const error={};
      if (!Values.name){
          error.name="Name field is required";
     }
    else if (Value.name.length>20{
        error.name="Name field must be less than 20 characters";
     }
        return error;
   }

});
      <div> User Form</div>
    <form onSubmit={formik.handleSubmit}>
      <div>
<label htmlFor="name">Name:</label>
        < input id="name" type="text"/>
        onChange={formiks.handleChange}
        value={formik.value.name}
       />
        formik.error.name?

          <div style={{color:"red"}}>{formik.error.name}</div>
             null
      </div>
      <div>
         <label htmlFor="age"Age:</label>
         <input id="age" type="number"onChange(handleChange}value={formik.values.age/>
     </div>
   </div>
          < button type="submit">Submit</button>
   </div
 </div>
 );
```

- Now go to the browser and enter the name character in the text box if these exceed more than 20 then it will display an error as shown from the red statement





3. <u>Schema Validation with Yup</u>

As you can see above, validation is left up to you. Feel free to write your validators or use a 3rd-party helper library. Formik's authors/a large portion of its users use Jason Quense's library Yup for object schema validation. Yup has an API that's similar to Joi / React PropTypes but is also small enough for the browser and fast enough for runtime usage. You can try it out here with this REPL.

Since Formik authors/users love Yup so much, Formik has a special configuration option/prop for Yup called validation schema which will automatically transform Yup's validation error messages into a pretty object whose keys match values/initial values/touched (just like any custom validation function would have to). Anyways, you can install Yup from NPM/yarn like so...

```
npm install yup --save
```

or via yarn

```
yarn add yup
```

- to get started with schemaValidation create a file name"UserFromWithFormikValidation.js"
- create schema validation for name and age, provide the condition and error as well
- create function **UserFromWithFormikValidation**
- **create** schema validation object with name and age property as

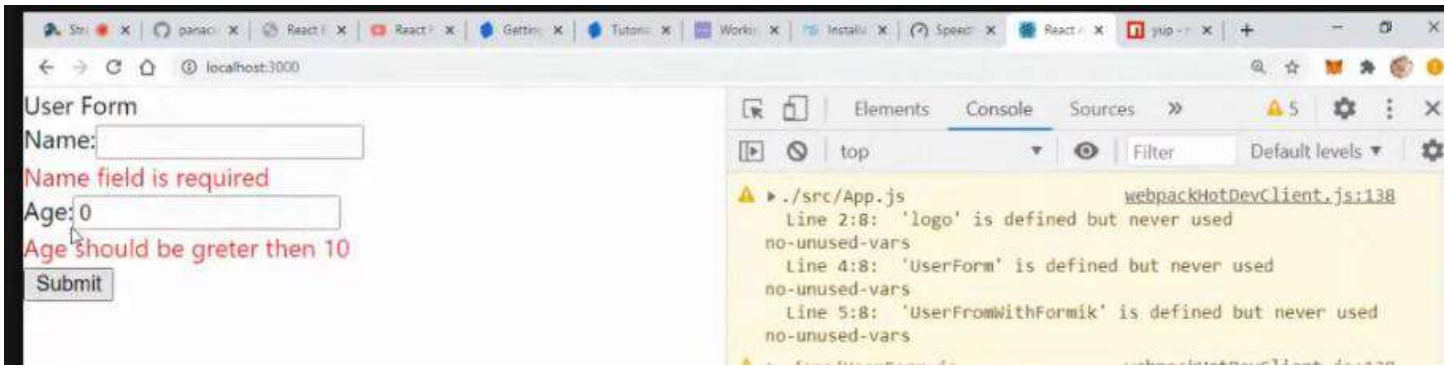
```
            validationSchema:Yup.object(
        {
          name.Yyup.string()
              .required("Nmae field is required")
              .max(15,"Name field should be less then 15 characters"),
         age.Yup.strig()
          .max(60,"age should be less than 60")
          .min(10,"Age should be greater than 10")

    });
```

User Form With FormikValidation.js"

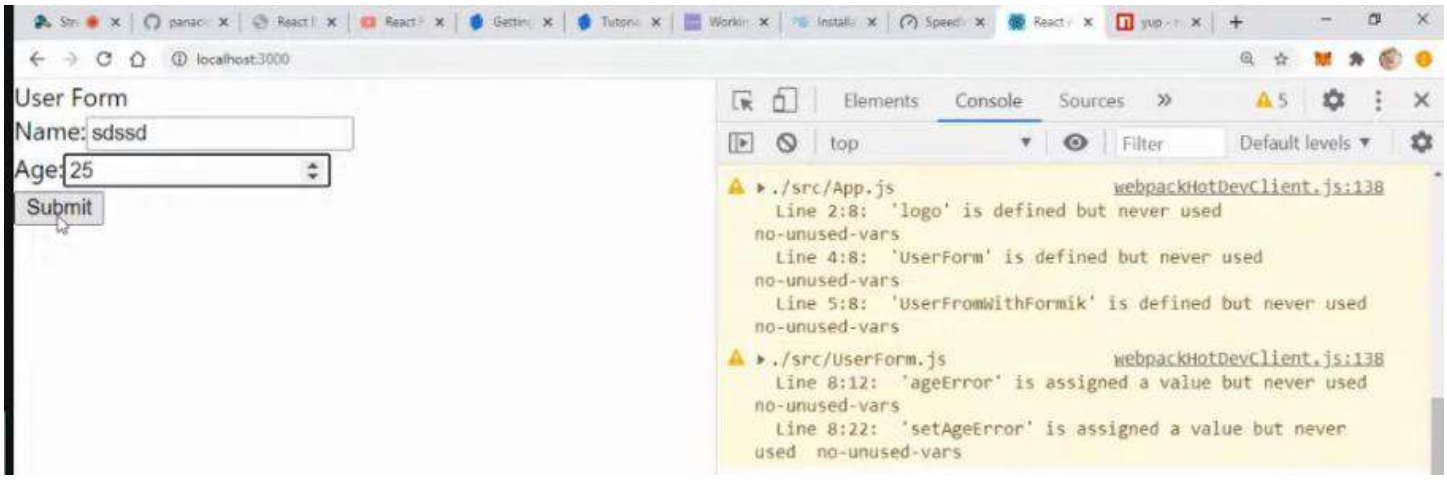
```
        import React from 'react';
    import { useFormik } from 'formik';
    import yup from ,./Yup';

    function UserFromWithFormikValidation(){
      const formik=userFormik({
         initilValues:{
           name:"",
           age:0
    }
        onSubmit: values => {
        console.log(values);
       },
       validationSchema:Yup.object(
         {
           name.Yyup.string()
               .required("Nmae field is required")
               .max(15,"Name field should be less then 15 characters"),
          age.Yup.strig()
           .max(60,"age should be less than 60")
           .min(10,"Age should be greater than 10")

    });
          <div> User Form</div>
        <form onSubmit={formik.handleSubmit}>
          <div>
    <label htmlFor="name">Name:</label>
           < input id="name" type="text"/>
           onChange={formiks.handleChange}


    value={formik.value.name}
           />
           formik.error.name?

   <div style={{color:"red"}}>{formik.error.name}</div>
      null
```

```
    </div>
    <div>
      <label htmlFor="age"Age:</label>
      <input id="age" type="number"onChange(handleChange}value={formik.values.age/>
     {
       formik.error.age?

       <div style={{color:"red"}}>{formik.error.age}</div>
         null
     }
   </div>
       < button type="submit">Submit</button>
  </div
</div>
);}
```

- Now go to the browser and click to submit button without adding any value to the text box
- It will display errors as shown



➢ no error as requirements are fulfilled



➢ We can use this formik getfield if we have several values to prevent the error code writing again and again

```
{...formik.getFeildProps("age")}/>
```

Leveraging React Contex

To save you even more time, Formik comes with React Context-powered API/component make life easier and less verbose: <Formik />, <Form />, <Field />, and <ErrorMessage />. More explicitly, they use React Context implicitly to connect to the parent <Formik /> state/methods.

- create UserFromWithFormikContext.js

User Form With FormikContext.js

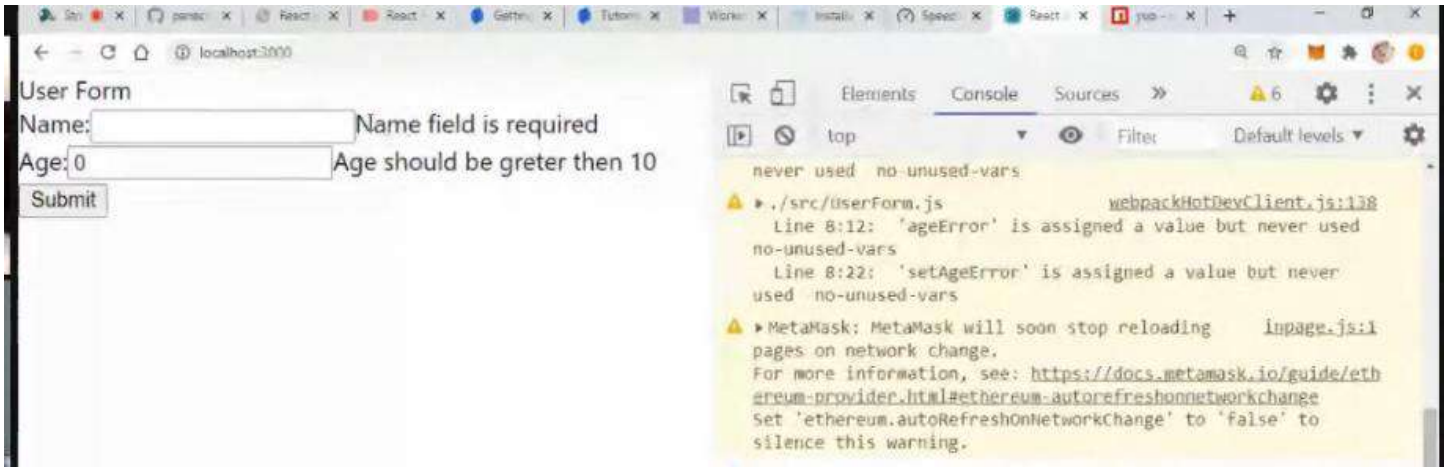
```
import React from 'react';
import { useFormik,Formik } from 'formik';
import yup from ,./Yup';

function UserFromWithFormikContext(){

 return (
    <div>
      <div> User Form</div>
          <Formik initialValues={{
          name:"",
          age:0
       }}
      onSubmit={(values)}=>{
        console.log(values);
        validationSchema={
         Yup.object(
        {
        name.Yup.string()
            .required("Nmae field is required")
            .max(15,"Name field should be less then 15 characters"),
      age.Yup.string()
        .max(60,"age should be less than 60")
        .min(10,"Age should be greater than 10")
{
    (formik)=>(


    <form onSubmit={formik.handleSubmit}>
        <div>
   <label htmlFor="name">Name:</label>
 <Field type="text"name="name"id="name"/>
<ErrorMessage name="name" render={(msg)}=>(
    <spam style={{color:"red"}}>{msg}</span>
   }}/>
```

```
        <div>
       <div>

    <labelhtmlFor="age">Age:</label>


     <Field type="number"name="age"id="name"/>
       <ErrorMessage name="name/>

  </div>
</div>
      < button type="submit">Submit</button>
</div>
```

App.js

```
import React from 'react';
import './App.css';
import UserFromWithFormik from './UserFromWithFormikContext';
function App(){
   return(
     <div>
       <UserFormWithFormikContext/>
     </div>
   );
  }
export default App;
```

Go to the browser and click on the submit button it will show the following statement :



- create UserFromWithFormikMaterialUI.js
- create textField and provide material ui styles
- provide variant in the field property

User Form With FormikMaterialUI.js

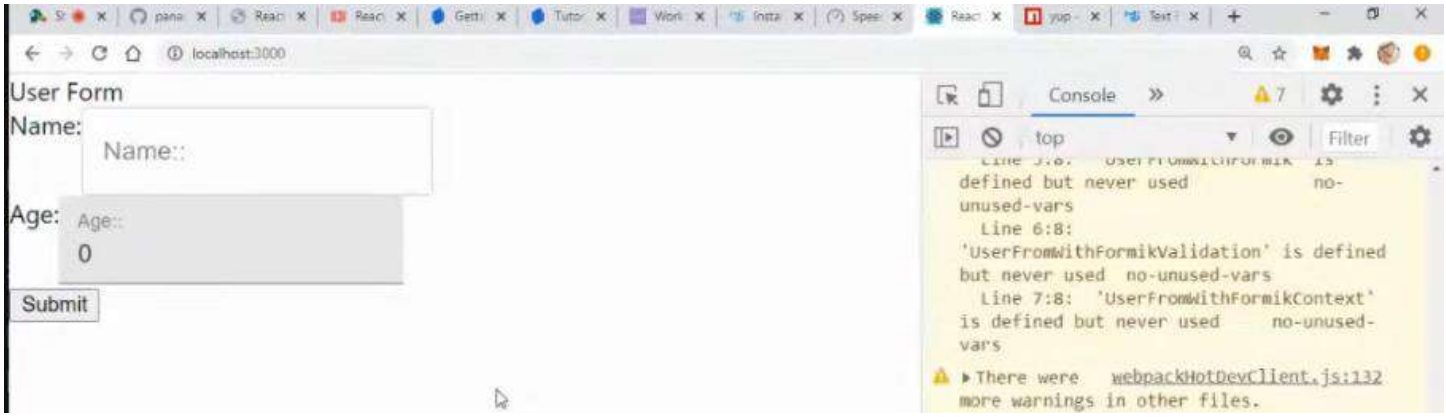
```
import React from 'react';
import { useFormik,Formik } from 'formik';
import yup from ,./Yup';
import TextField from 'material-ui/core/TextField';

function UserFromWithFormikMaterialUI(){

 return (
   <div>
     <div> User Form</div>
    <Formik initialValues={{
         name:"",
         age:0
      }}
     onSubmit={(values)}=>{
       console.log(values);
       validationSchema={
        Yup.object(
       {
       name.Yup.string()
           .required("Nmae field is required")
           .max(15,"Name field should be less then 15 characters"),
     age.Yup.string()
       .max(60,"age should be less than 60")
       .min(10,"Age should be greater than 10")
{
   (formik)=>(


   <form onSubmit={formik.handleSubmit}>
       <div>
  <label htmlFor="name">Name:</label>



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

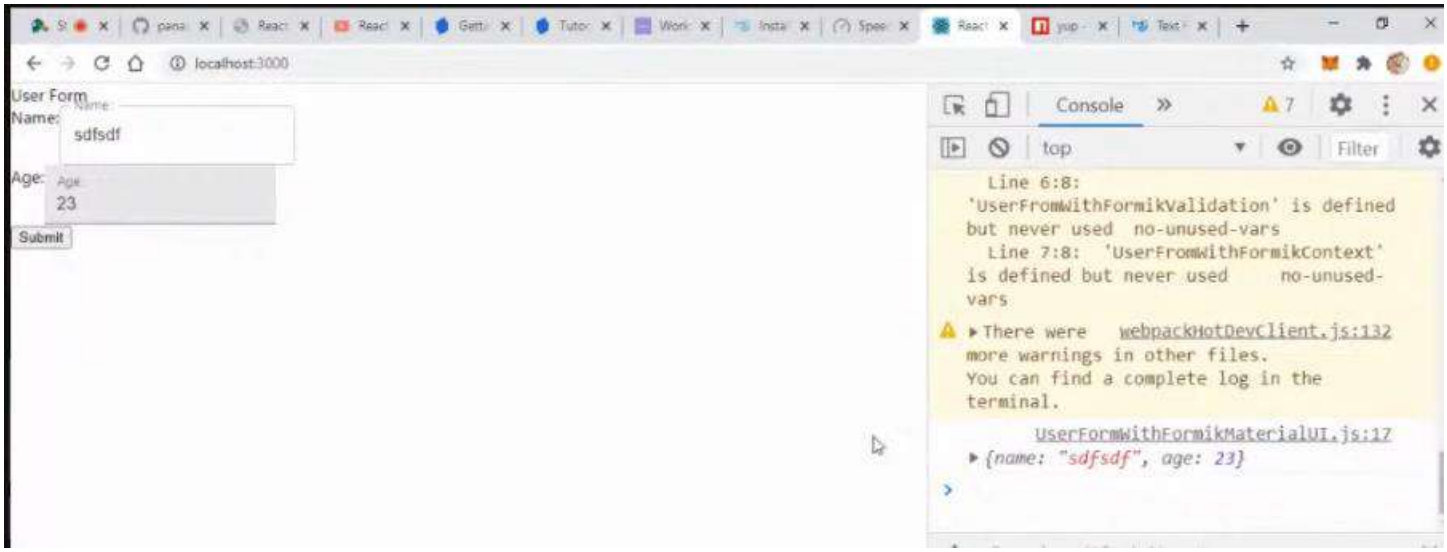
```
<ErrorMessage name="name" render={(msg)}=>(

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

<div>
    <labelhtmlFor="age">Age:</label>


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


     </div>
   </div>
          < button type="submit">Submit</button>
   </div
</Formik>
 </div>
 );
```

App.js

```
import React from 'react';
import './App.css';
import UserFromWithFormik from './UserFromWithFormikMaterialUI';
function App(){
   return(
     <div>
       <UserFormWithFormikMaterialUI/>
     </div>
   );
  }
          export default App;
```

- this is how you can use material UI in the context field
- now go to the browser and enter the value in the context field as shown:





This is how formik provides multiple features

Session 19(A): Multi-Page_Forms_using_Formik

MultiSteps Forms: Creating Linear -Alternative Level

- create folder
- create react app multi-steps using npx

```
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC: ~/Desktop/reactmultiistepform
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop$ mkdir reactmultiistepfo
rm
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop$
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop$ cd reactmultiistepform/
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/reactmultiistepform$ ls
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/reactmultiistepform$ sud
o npx create-react-app multistepform
[sudo] password for daniyal:
```

- Now after the project is being created install all dependencies formik ,yup,material UI
- create a folder of stepper inside it create index.js file and follow this coding

index.js

```
import React from 'react';
import { makeStyles } from '@material-ui/core/styles';
import Stepper from '@material-ui/core/Stepper';
import Step from '@material-ui/core/Step';
import StepLabel from '@material-ui/core/StepLabel';
import Button from '@material-ui/core/Button';
import Typography from '@material-ui/core/Typography';

const useStyles = makeStyles((theme) => ({
  root: {
    width: '100%',
  },
  backButton: {
    marginRight: theme.spacing(1),
  },
  instructions: {
    marginTop: theme.spacing(1),
    marginBottom: theme.spacing(1),
  },
}));

function getSteps() {
  return ['Select master blaster campaign settings', 'Create an ad group', 'Create an ad'];
}
```

```
      case 0:
        return 'Select campaign settings...';
      case 1:
        return 'What is an ad group anyways?';
      case 2:
        return 'This is the bit I really care about!';
      default:
        return 'Unknown stepIndex';
    }
  }

  export default function HorizontalLabelPositionBelowStepper() {
    const classes = useStyles();
    const [activeStep, setActiveStep] = React.useState(0);
    const steps = getSteps();

    const handleNext = () => {
      setActiveStep((prevActiveStep) => prevActiveStep + 1);
    };

    const handleBack = () => {
      setActiveStep((prevActiveStep) => prevActiveStep - 1);
    };

    const handleReset = () => {
      setActiveStep(0);
    };

    return (
      <div className={classes.root}>
        <Stepper activeStep={activeStep} alternativeLabel>
          {steps.map((label) => (
            <Step key={label}>
              <StepLabel>{label}</StepLabel>
            </Step>
          ))}
        </Stepper>
        <div>
          {activeStep === steps.length ? (
            <div>
              <Typography className={classes.instructions}>All steps completed</Typography>
              <Button onClick={handleReset}>Reset</Button>
            </div>
          ); (
            <div>
              <Typography
  className={classes.instructions}>{getStepContent(activeStep)}</Typography>
```

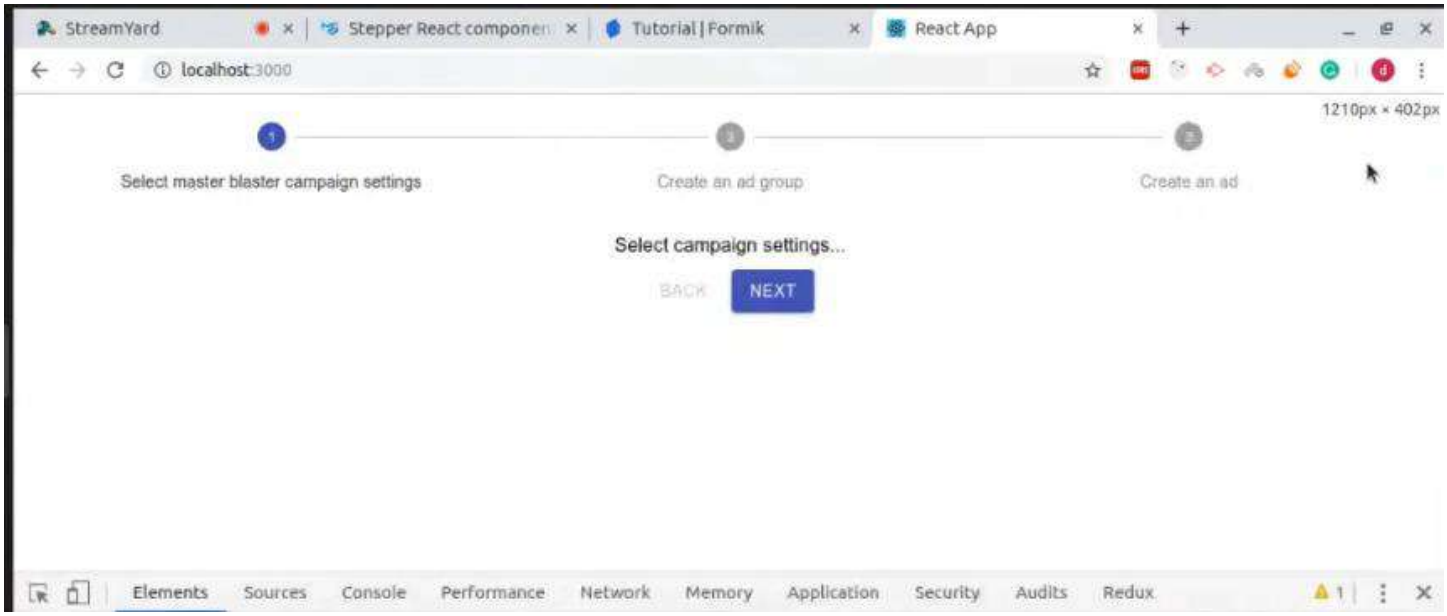
```
                onClick={handleBack}
                className={classes.backButton}
              >
                Back
              </Button>
              <Button variant="contained" color="primary" onClick={handleNext}>
                {activeStep === steps.length - 1 ? 'Finish' : 'Next'}
              </Button>
            </div>
          </div>
        )}
      </div>
    </div>
  );
}
```

Now import stepper in the App.js

App.js

```
import React from 'react':
import './App.css';
import Stepper from "./stepper";
function App(){
  return (
   <div className="App">
    <Stepper/>
  );
 }
export default App;
```

save all coding, go to the browser and run it



- follow this coding to create a basic stepper of address, the viewer

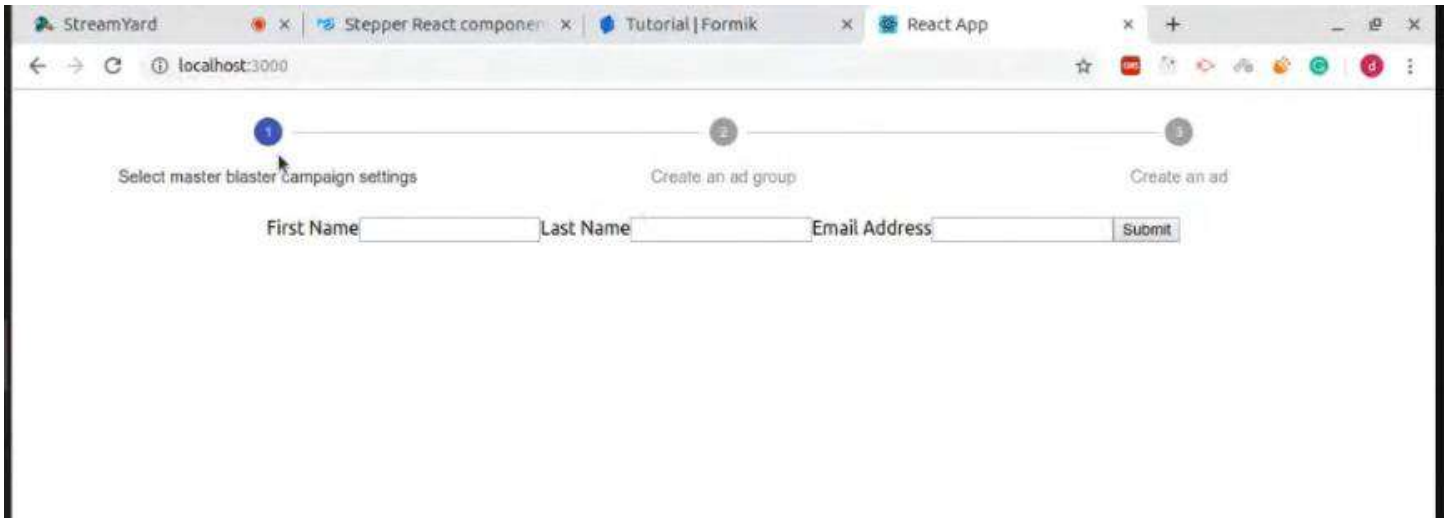
```
import React from 'react';
import { makeStyles } from '@material-ui/core/styles';
import Stepper from '@material-ui/core/Stepper';
import Step from '@material-ui/core/Step';
import StepLabel from '@material-ui/core/StepLabel';
import Button from '@material-ui/core/Button';
import Typography from '@material-ui/core/Typography';

const useStyles = makeStyles((theme) => ({
  root: {
    width: '100%',
  },
  backButton: {
    marginRight: theme.spacing(1),
  },
  instructions: {
    marginTop: theme.spacing(1),
    marginBottom: theme.spacing(1),
  },
}));

function getSteps() {
  return ['Select master blaster campaign settings', 'Create an ad group', 'Create an ad'];
}
```

```
            return <Address/>
          case 1:
            return 'What is an ad group anyways?';
          case 2:
            return 'This is the bit I really care about!';
          default:
            return 'Unknown stepIndex';
        }
      }

      export default function HorizontalLabelPositionBelowStepper() {
        const classes = useStyles();
        const [activeStep, setActiveStep] = React.useState(0);
        const steps = getSteps();



        return (
          <div className={classes.root}>
            <Stepper activeStep={activeStep} alternativeLabel>
              {steps.map((

  <Step key={label}>
        <StepLabel>{label}</StepLabel>
      </Step>
    ))}
   </Stepper>
  {
  getstepContent(activeStep)
 }

   </div>
 );
}
```

- create forms
- create a folder of forms component
- create address form and index.jss in it
- create the component of payementDetails and create the logic index.jsx in it
- create the third component of review ith index.js file in it

```
import React from 'react';
 import { Formik, Field, Form, ErrorMessage } from 'formik';
 import * as Yup from 'yup';

 const Address= () => {
   return (
     <Formik
       initialValues={{ firstName: '', lastName: '', email: '' }}
       validationSchema={Yup.object({
         firstName: Yup.string()
           .max(15, 'Must be 15 characters or less')
           .required('Required'),
         lastName: Yup.string()
           .max(20, 'Must be 20 characters or less')
           .required('Required'),
         email: Yup.string().email('Invalid email address').required('Required'),
       })}
       onSubmit={(values, { setSubmitting }) => {
         setTimeout(() => {
           alert(JSON.stringify(values, null, 2));
           setSubmitting(false);
         }, 400);
       }}
     >
       <Form>
         <label htmlFor="firstName">First Name</label>
         <Field name="firstName" type="text" />
         <ErrorMessage name="firstName" />
         <label htmlFor="lastName">Last Name</label>
         <Field name="lastName" type="text" />
         <ErrorMessage name="lastName" />
         <label htmlFor="email">Email Address</label>

 <Field name="email" type="email" /
         <ErrorMessage name="email" />
         <button type="submit">Submit</button>
       </Form>
     </Formik>
   );
 };
```

The <Field> component by default will render a <input> component that given a name prop will implicitly grab the respective onChange, onBlur, value props, and pass them to the element as well as any props you pass to it. However, since not everything is input, <Field> also accepts a few other props to let you render whatever you want



- now let us create stepper for address, payment, and Review as follows by creating component of these three as shown

stepper/index.js

```
import React from 'react';
import { makeStyles } from '@material-ui/core/styles';
import Stepper from '@material-ui/core/Stepper';
import Step from '@material-ui/core/Step';
import StepLabel from '@material-ui/core/StepLabel';
import Button from '@material-ui/core/Button';
import Typography from '@material-ui/core/Typography';

const useStyles = makeStyles((theme) => ({
  root: {
    width: '100%',
  },
  backButton: {
    marginRight: theme.spacing(1),
  },
  instructions: {
    marginTop: theme.spacing(1),
    marginBottom: theme.spacing(1),
  },
}));

function getSteps() {
  return [return ['add Address', 'add Payment', 'Add Review'];
}

function getStepContent(stepIndex,setStep ,setFormValues,formvalues) {
  switch (stepIndex) {
    case 0:
      return <Address submit={setSteps} setFormValues ={ setFormValues }/>
    case 1:
```

```
      return <Payment submit={setSteps} setFormValues ={ setFormValues />
    case 2:
      return <Review submit={setSteps} values={formValues}/>
    default:
      return 'Unknown stepIndex';
  }
}

export default function HorizontalLabelPositionBelowStepper() {
  const classes = useStyles();
  const [activeStep, setActiveStep] = React.useState(0);
const{formValues,setFormValues]=React.useState({})
  const steps = getSteps();



  return (
    <div className={classes.root}>
      <Stepper activeStep={activeStep} alternativeLabel>
        {steps.map((label) => (
          <Step key={label}>
            <StepLabel>{label}</StepLabel>
          </Step>
        ))}
      </Stepper>
   {
   getstepContent(activeStep, setActiveSteps,setFormValues,formvalues)
  }

    </div>
  );
}
```

index.js/Address

```
import React from 'react';
 import { Formik, Field, Form, ErrorMessage } from 'formik';
 import * as Yup from 'yup';

export const Address= ({submit,setFormValues}) => {
   return (
     <Formik
       initialValues={{ area:'',city:'',email }}
       validationSchema={Yup.object({
         area: Yup.string()
           .max(15, 'Must be 15 characters or less')
```

```
          .required('Required'),
        city: Yup.string()
          .max(20, 'Must be 20 characters or less')
          .required('Required'),

      })}
      onSubmit={(values)=>
          submit(1)
    setFormValues{{…Values}}
      }}
    >
      <Form>
        <label htmlFor="area">Area Name</label>
        <Field name="area" type="text" />
        <ErrorMessage name="area" />
    <br>
        <label htmlFor="city">City Name</label>
        <Field name="city" type="text" />
        <ErrorMessage name="city" />
   <br>
        <label htmlFor="email">Email Address</label>
        <Field name="email" type="email" />
        <ErrorMessage name="email" />
        <button type="submit">Submit</button>
      </Form>
    </Formik>
  );
};
```

index.js/payment

```
import React from 'react';
 import { Formik, Field, Form, ErrorMessage } from 'formik';
 import * as Yup from 'yup';

export const payment= ({submit}) => {
   return (
     <Formik
       initialValues={{ cardName ' ', cardNumber: ' '}}
       validationSchema={Yup.object({
         cardName: Yup.string()
           .max(15, 'Must be 15 characters or less')
           .required('Required'),
         cardNumber: Yup.string()
           .max(20, 'Must be 20 characters or less'
            .required('Required'),

       onSubmit={(values, ) => {
         submit(2)
       }}
```

```
            <Form>
              <label htmlFor="cardName">Card Name</label>
              <Field name="cardName" type="text" />
              <ErrorMessage name="cardName" />
              <label htmlFor="cardNumber">Card Number</label>
              <Field name="cardNumber" type="text" />
              <ErrorMessage name="cardNumber" />
              <button type="submit">Submit</button>
                    <button onClick={()=>submit(1)}>Back</button>
          </div>

            </Form>
           </Formik>
          );
```

index.js/Review

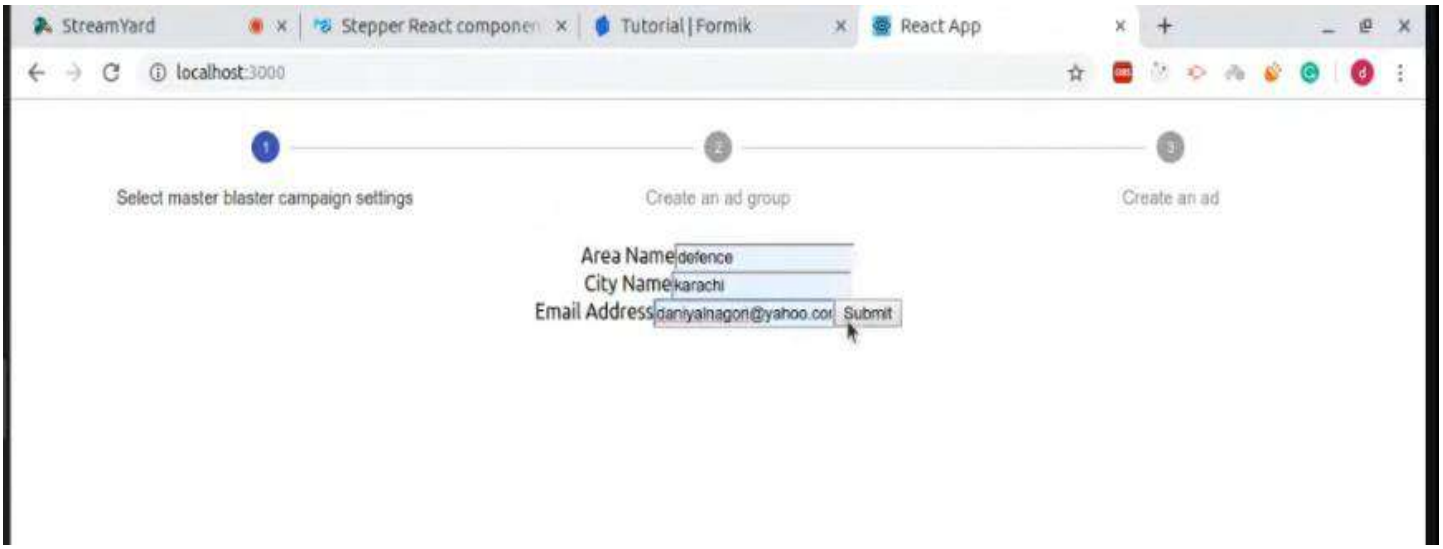
index.js/Review

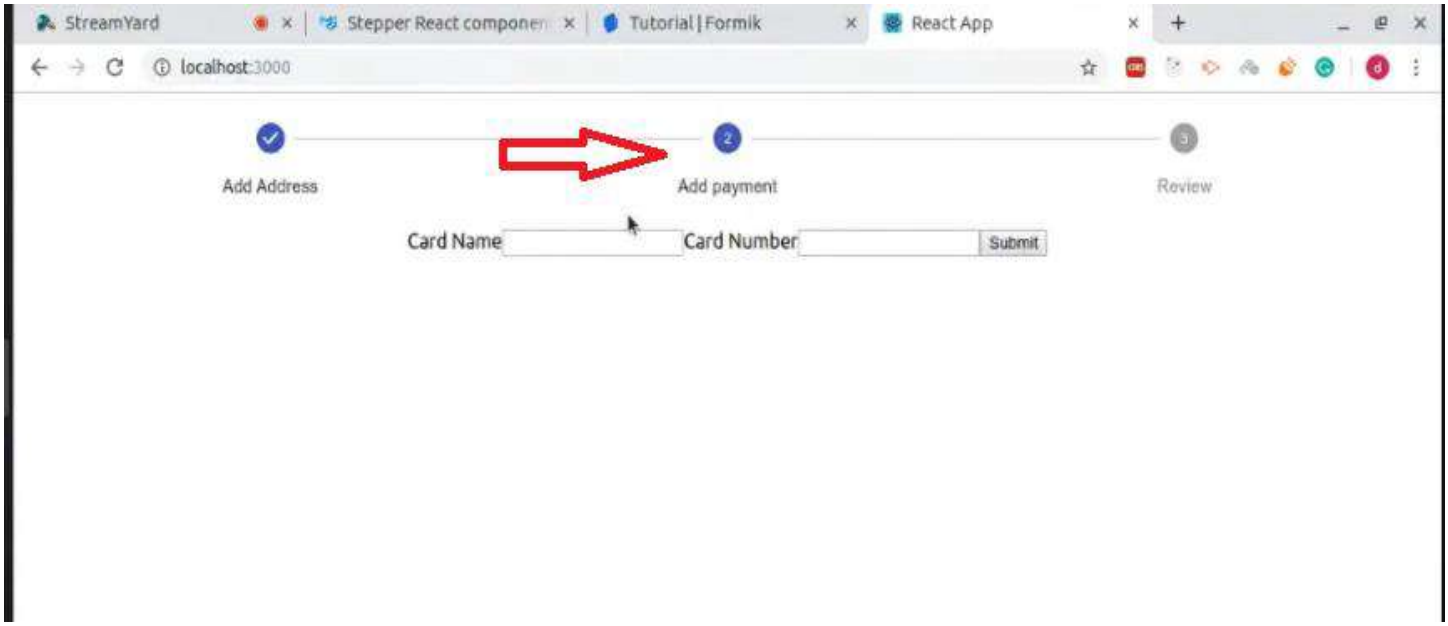
```
import React from 'react';
 import { Formik, Field, Form, ErrorMessage } from 'formik';
 import * as Yup from 'yup';

export const Review= ({values}) => {
   return (
     <div>
       <p>Area:{values.area</P>
        <p>City:{values.city</P>
         <p>Card Name:{values.cardName</P>
           <p>Card Number:{values.cardNumver</p>
         <p>Email:{values.email}</P>
            <p>$total:20$</P>
         <button onClick={()=>submit(1)}>Back</button>
  </div>

  );
};
```

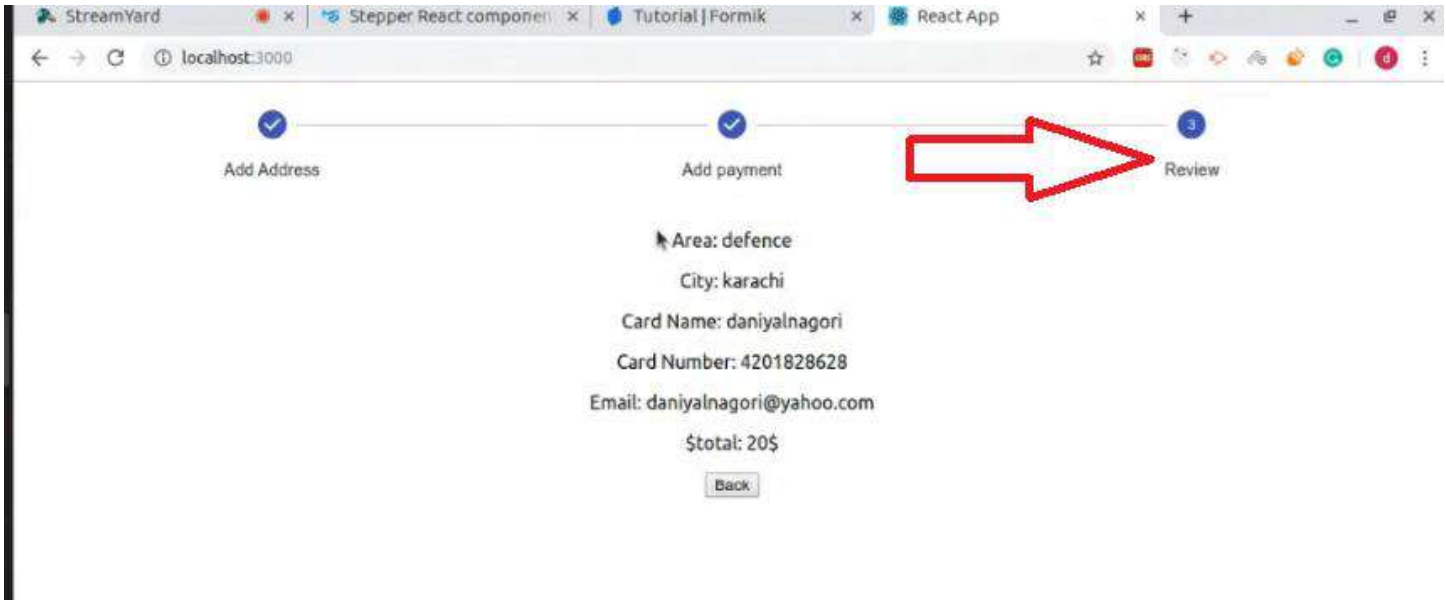
- Now go to the browser and enter the required statements ii them
  - Add Address

- Add payment
- Add Review

➢ Look at the output as shown






StreamYard
Stepper React componen
Tutorial | Formik
React App
localhost:3000
Add Address
Add payment
Review
Area: defence
City: karachi
Card Name: daniyalnagori
Card Number: 4201828628
Email: daniyalnagori@yahoo.com
$total: 20$
Back

Session 20: Component Driven Development (CDD)

Introduction To Component Driven Develoment

Component Driven Development

Component Driven Development means designing software applications by building loosely coupled independent components.

Component Driven Development

CDD is an all-encompassing concept which applies globally to design systems, pattern libraries and modern JavaScript frameworks, most notably React.

Component Driven Development

Like decoupled micro-services enable scale and efficiency for back-end development, the concept of component-driven development can enable similar advantages for teams building modern web applications.

Advantages of CDD

Organizations can achieve strategic goals that make their web development faster, more scalable and efficient

Autonomous Teams | Simple, Decoupled Codebases
Incremental Upgrades | Rapid Releases

Advantages of CDD

Autonomous Teams that own well defined features or products

Simple, Decoupled Codebases - teams can develop their own components without being delayed and complicated by others

Incremental Upgrades - Independent components for fast and modular upgrades

Rapid Releases - Deliver components independently and continuously to upgrade products or make fixes without complications with integrations

Your disk is almost full
Save space by optimizing storage.
Close
Manage...

Advantages of CDD for the Software

Faster Development - By modularizing development into components, you can build modular parts and narrow-focused APIs. This leads to faster build and testing times for each component.

Your disk is almost full
Save space by optimizing storage.
Close
Manage...

Advantages of CDD for the Software

Simpler Maintenance - Modifying or updating a part of an application is made easier by extending or updating the component instead of having to refactor larger parts of the application.

Your disk is almost full
Save space by optimizing storage.
Close
Manage...

Advantages of CDD for the Software

Better TDD - Writing unit tests to validate focused functionality for each component is easier when the component is in an independent environment.

Advantages of CDD for the Software

Shorter Learning Curves - Instead of having to learn the entire application, a new team member can focus on an independent code and get to work right away.

Advantages of CDD for the Software

Better Modeling of the System - Because the system is made up of smaller, modular components, it's easier to design, understand and work on.

What is CDD?

A website is often an interface made up of repeating components and patterns.

Repetition and Reusability benefits all stakeholders of a system. Global rules to systems enable better use of an interface and improves the design and development processes.

What is CDD?

CDD means designing software apps by building loosely-coupled independent components.

Instead of looking at all the components as a whole, CDD is an approach to software development that focuses on precision, attention-to-detail, and modularity for components, one at a time.

What is a Component?

A component is a well-defined, independent piece of user interface. A component could include things like buttons, headings, calls to action, hero banner, tables, navigation, header, footer, etc.

What is a Component?

In modern JS frameworks like React, we can style components based on prop attributes that are fed to the components.

```
<Button theme="primary">
<Button theme="secondary">
<Button theme="primary" size="large">
```

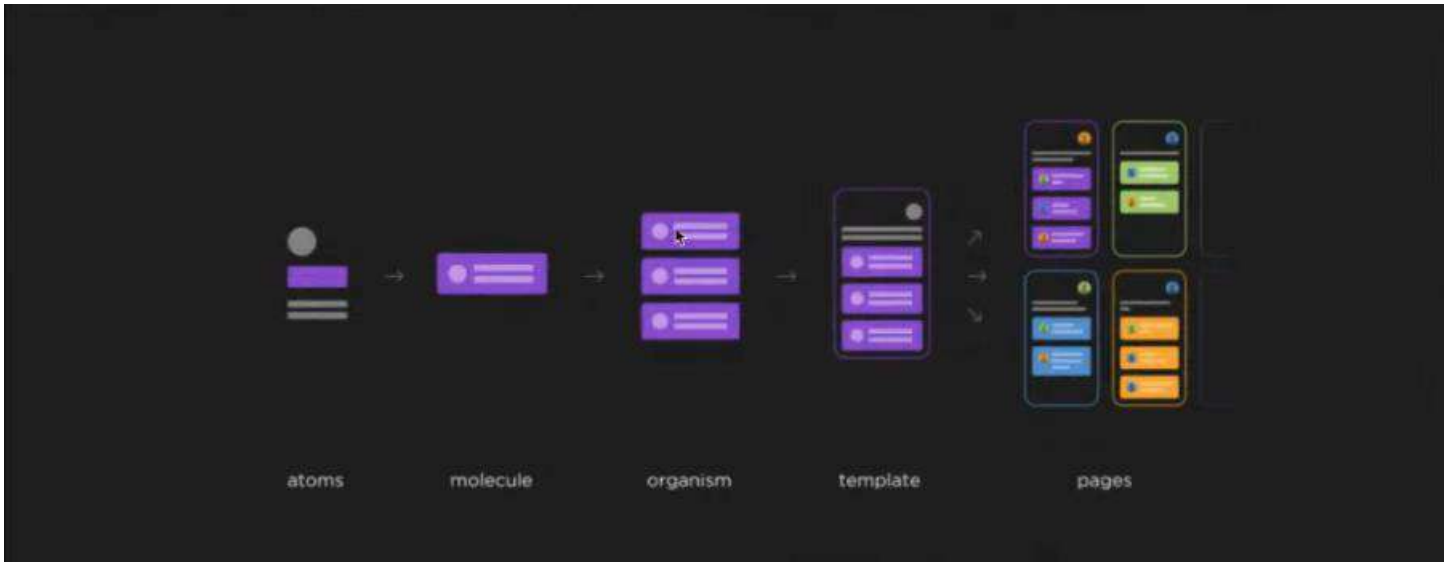
Atomic Design

Focus on looking at interfaces as a whole and then dissecting them into components of various size to make up the interface.

Atomic Design

Atomic Design introduces a standardized language for discussing components and labelling structures.

1. Atoms
2. Molecules
3. Organisms
4. Templates
5. Pages



CDD with Atomic Design

With Atomic Design, we can use the universal language shared with developers and designers who have used the Atomic Design methodology.

With CDD, we can get the precision, focus on the performance of each component that allows to assign responsibility to team members over functionality, layout and presentation in the final delivery of components

CDD with React

Develop components with placeholder text in a pattern library before importing the components to a live environment where the data can be pulled from a content management system.

Tools for CDD
Bit - The Component-Driven Platform

Bit provides increased speed, scale, and efficiency for the development of modern applications by facilitating component-driven development.

Bit includes two key parts:
A set of open source tools & a cloud platform

Tools for CDD
Bit - The Component-Driven Platform

A Cloud Platform

All published components are shared, documented, made discoverable, and installable to different applications. Teams can get and release updates to specific components, and learn how every change impacts dependent components in different apps.

Key Advantages of Bit
Acceleration of Web Development

Tools to quickly create, compose, upgrade, and manage components in apps and reduce time to market.

Helps to expose and integrate components with each other to build faster as teams

Brings component reuse up to 100% to save overlapping work.

Key Advantages of Bit Standardization & Governance

Helps to standardize the development process of components.

Tools such as customizable component development environments and automated workflows to define and regulate the way components are developed, built, tested, and documented by all the different teams working across different applications.

Key Advantages of Bit Autonomous Teams

Allows for a flexible team structure that gives teams true autonomous ownership over their features or products, from development to production.

Teams can develop components independently in their own codebases, constantly release innovation, and smoothly expose and integrate components with each other.

Key Advantages of Bit
Scalable, Decoupled Codebase

Because each team works in independent codebases, they are decoupled from other teams. This allows for simpler build, test, and maintenance of components.

This separation gives organizations an improved ability to scale web development in the same way microservices enable backend scalability.

Key Advantages of Bit
More Releases to Production

Because each team is free from obstacles created by other teams, they are able to deliver more releases to production.

Each team can constantly release upgrades, changes, or fixes multiple times a day without breaking anything else.

Up to **30x more releases** to production by product teams.

Key Advantages of Bit Incremental Upgrades

Independent Components allow for incremental release of modular upgrades to production.

Bit does not build changes to an entire monolith. It runs on a component dependency-graph across all applications.

Key Advantages of Bit Structuring Innovation

A small team is empowered to make decisions and relentlessly drive toward their goals delivers results and insights much more quickly than a larger group.

Empowers product teams to rapidly and independently release features and upgrades to production.

Key Advantages of Bit
100% Component Reuse and Consistency

Every component in the organization becomes reusable.

Developers have tools to easily share their components with others, and reuse components shared by others into their own apps.

Key Advantages of Bit
Eliminating Bugs and Glitches from Production

Tools to ensure that all components go thru a standardized and automated pipeline for common tasks like build and test

If a component doesn't meet the acceptance criteria, it will not make it to production.

Key Advantages of Bit
Reduced Cost, Resources, and Effort

Saves a lot of development time, resources and costs for web development.

Reduces the amount of duplicated code in the organization, speeds up delivery of new features, reduces maintenance costs for fixes and changes, and eliminates a lot of tools and resources wasted on legacy issues.

Storybook

A rapid development environment for UI components that allows to browse a component library, view the different states of each component, and interactively develop and test components.

Develop components in isolation from your application.

React Styleguidist

React Styleguidist is another a component development environment with hot reloaded dev server and a living style guide that lists component propTyles and shows editable usage examples.

It works with Create React App out of the box and supports ES6, Flow and TypeScript

Storybook vs React Styleguidist

Storybook is based on stories written in JS files
Styleguidist is based on examples in Markdown files

Storybook shows one variation of a component at a time, Styleguidist can show multiple variations of different components.

Storybook vs React Styleguidist

Storybook is good for showing a components states, Styleguidist is useful for documentation and demos of different components.

Storybook with React

Storybook allows to create React components.

You can import a react component into a storybook file. The storybook environment can exist as it's own git repository or it can be directly inside of a react project repository

Benefits of CDD for Dev Teams

- Faster Development Times
- More Code Reuse
- Better Test Driven Development
- A Unified & Consistent Design System
- Access Pre-built Components
- Collaboration on Changes to Components rather than re-writing components

Benefits of CDD for Dev Teams

- Less Code to Maintain
- More Focus on Building New Things
- Standardized and Consistent Development Methodology

Starting Out:StoryBook

Let's start by building the basic example mentioned above. This example will get us familiar with how to create stories and how the interface of the stories look like. You'll start by creating the React application and installing the Storybook in it.

- From your terminal, run the command below:

```
# Scaffold a new application.
npx create-react-app table-component

# Navigate into the newly created folder.
cd table-component

# Initialise storybook.
npx -p @storybook/cli sb init
```

After that, check that the installation was successful by running the following commands:

- In one terminal:

```
yarn start
```

- and in the other:

```
yarn storybook
```

You will be greeted by two different screens: the React application and the storybook explorer.

With storybook installed in our applications, you'll go on to remove the default stories located in the src/stories folder

- create a React app of name storybook
- after that install the dependencies of version 5.3.19 storybook

package.json

```
{
  "name": "storybook-react-example",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^4.2.4",
    "@testing-library/react": "^9.5.0",
    "@testing-library/user-event": "^7.2.1",
    "react": "^16.13.1",
    "react-dom": "^16.13.1",
    "react-scripts": "3.4.1"
  },
   scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject",
    "storybook": "start-storybook -p 6006 -s public",
    "build-storybook": "build-storybook -s public"
  },
  "eslintConfig": {
    "extends": "react-app"
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  },
  "devDependencies": {
    "@storybook/addon-actions": "^5.3.19",
    "@storybook/addon-knobs": "^5.3.19",
    "@storybook/addon-links": "^5.3.19",
    "@storybook/addons": "^5.3.19",
    "@storybook/preset-create-react-app": "^3.1.4",
    "@storybook/react": "^5.3.19"
  }
}
```

- Now create a folder of STORYBOOK after
- this creates a component of main.js in it

- create component folder add counter.js in it

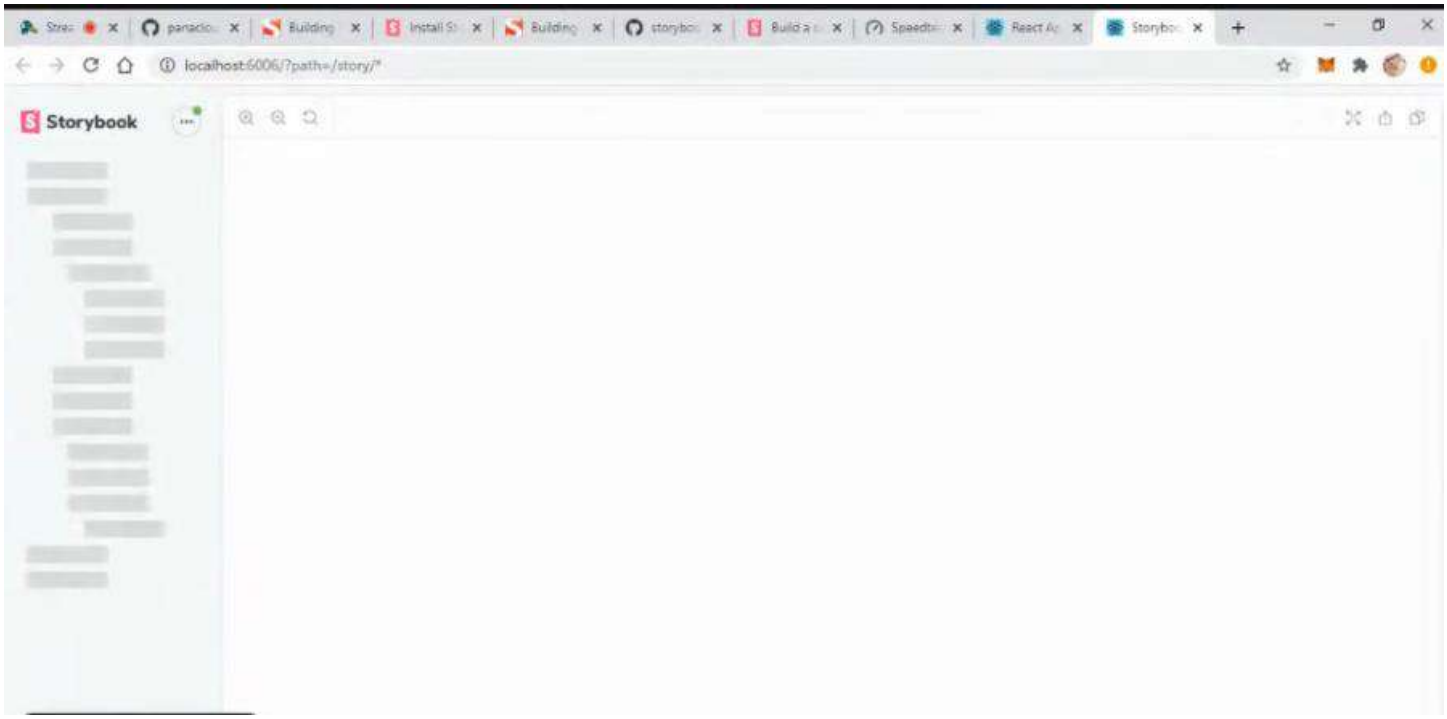
counter.js

```
import React from 'react';
function Counter(){
 return(
  <div>
      Hello Counter
  </div>
 );
}
export default counter;
```

App.js

```
import React from 'react';

function App(){
 return(
  <div>
      <Counter/>
  </div>
 );

}
export default App;
```

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.18363.1016]
(c) 2019 Microsoft Corporation. All rights reserved.

D:\developmentData\bootcamp2020\storybook\react-storybook>yarn storybook
```

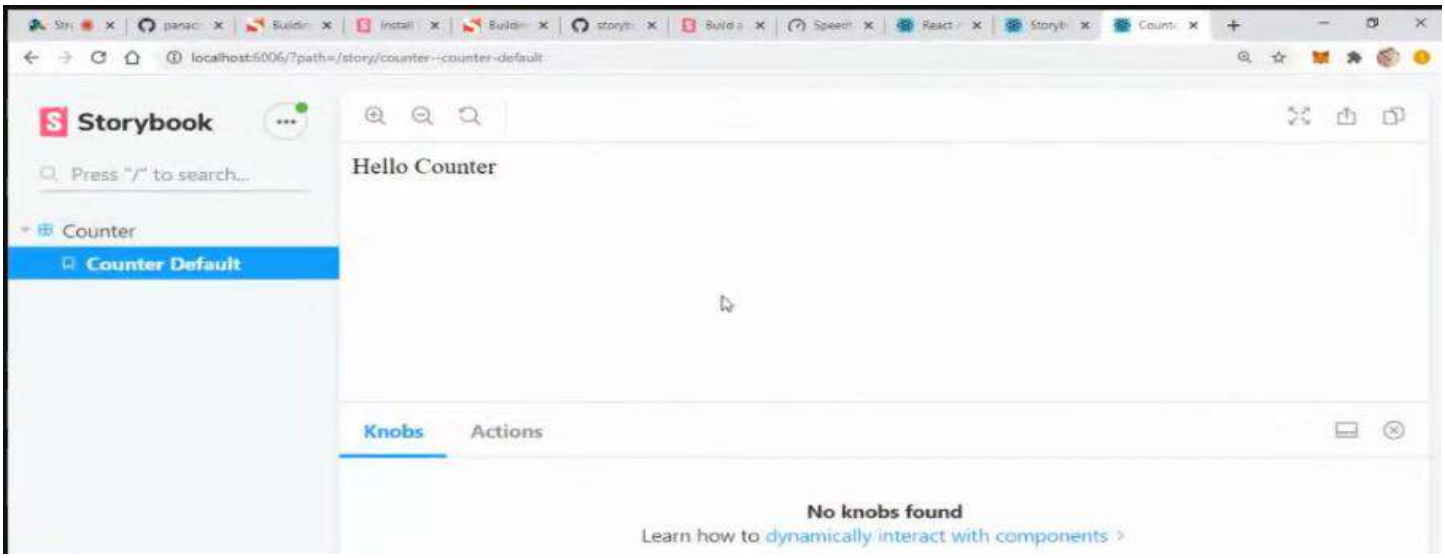


- Now lets create a story with title “counter” as
- create story in main.js

```
.storybook/main.js
module.export={
  stories:['../src/**/*.stories.js']
  addons:[
    '@storybook/preset-create-react-app',
    '@storybook/addon-actions',
    '@storybook/addon-links',
    '@storybook/addon-knobs'
  ],
};
```

- creat folder name stories
- create a file in it counter.stories.js

```
counter.stories.js
import Counter from "../component/Counter;
import React from './react';

export default {
  title:"counter"
  component:counter

}
export function CounterDefault(){
  return(
    <counter/>
  );
}
```



Lets create multiple option in storybook and with red background

Create three different default functions in storybook.js of

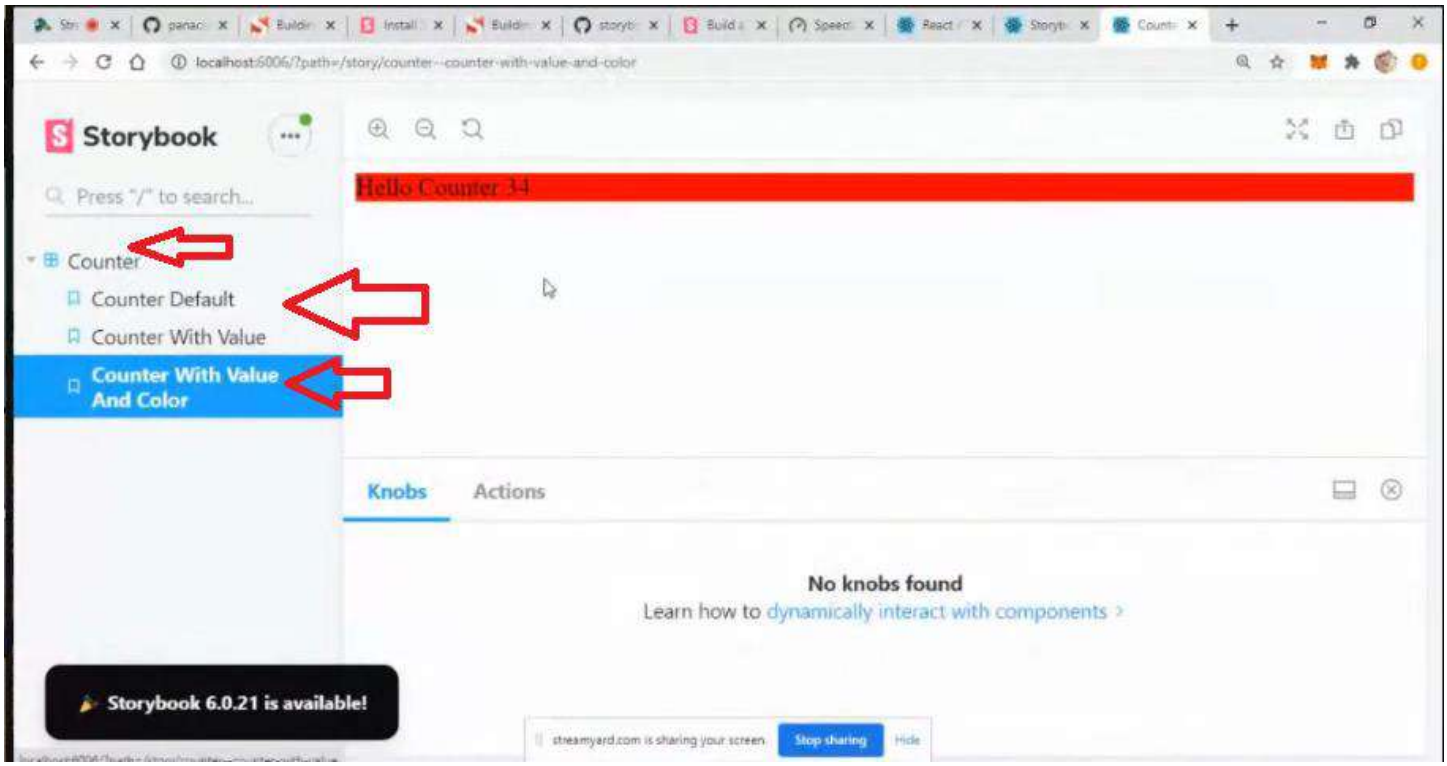
- CounterDefault(){
- CounterWithValue(){
- CounterWithValueAndColor(){

counter.stories.js

```
import Counter from "../component/Counter;
import React from './react';
export default {
  title:"counter"
  component:counter
}
export function CounterDefault(){
  return(
    <counter/>
  );
}
export function CounterWithValue(){
  return(
    <counter initialValue={34}/>
  );
}
export function CounterWithValueAndColor(){
  return(
    <counter initialValue={34} backgroundColor="red"/>
  );
}
```

counter.js

```
import React from 'react';

function Counter(props){
 return(
  <div style={{backgroundColor:props.backgroundColor}}>
      Hello Counter{props.initialValue}
  </div>
 );
}
export default counter;
```



- create a component button of increment and decrement
- and create a simple onclick button for showing an alert message

Button.js

```
import React from 'react';

function Button(props){
 return(
  <button onClick>={()=>
    alert ("Hello")

}}<{props.title}</button>

  </div>
 );
}
export default Button;
```

counter.js

```
import React from 'react';

function Counter(props){
 return(
  <div style={{backgroundColor:props.backgroundColor}}>
      Hello Counter{props.initialValue}
      <br/>
      <Button title="Increment"/>
       <Button title="decrement"/>
  </div>
 );
}
```

go to the browser and check out the output

- now to create separate stories create a file Button.stories.js
- as well as create large and small button as ;

Button.stories.js

```
import React from './react';

export default {
  title:"Button"
  component:Button
}
export function DefaultButton(){
  return(
    <Button title="Hello"/>
  );
}
export function smalltButton(){
  return(
    <Button title="Hello" size="small"/>
  );
}
export function largeButton(){
  return(
    <Button title="Hello" size="large"/>
  );
}
```

- add Button.css file and import button in button.js

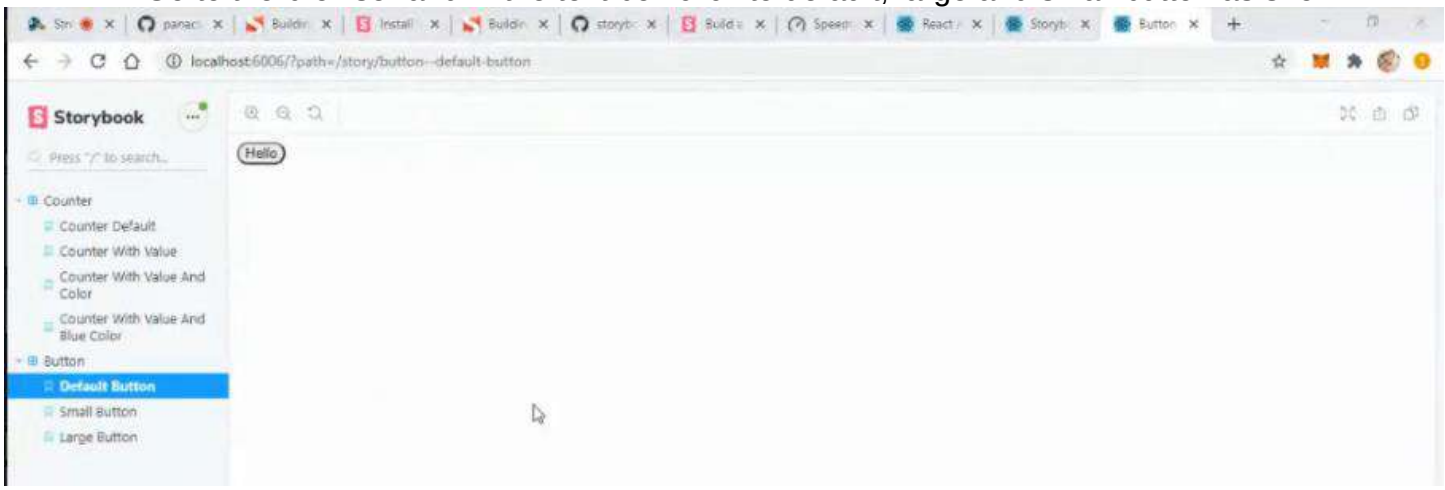
button.js

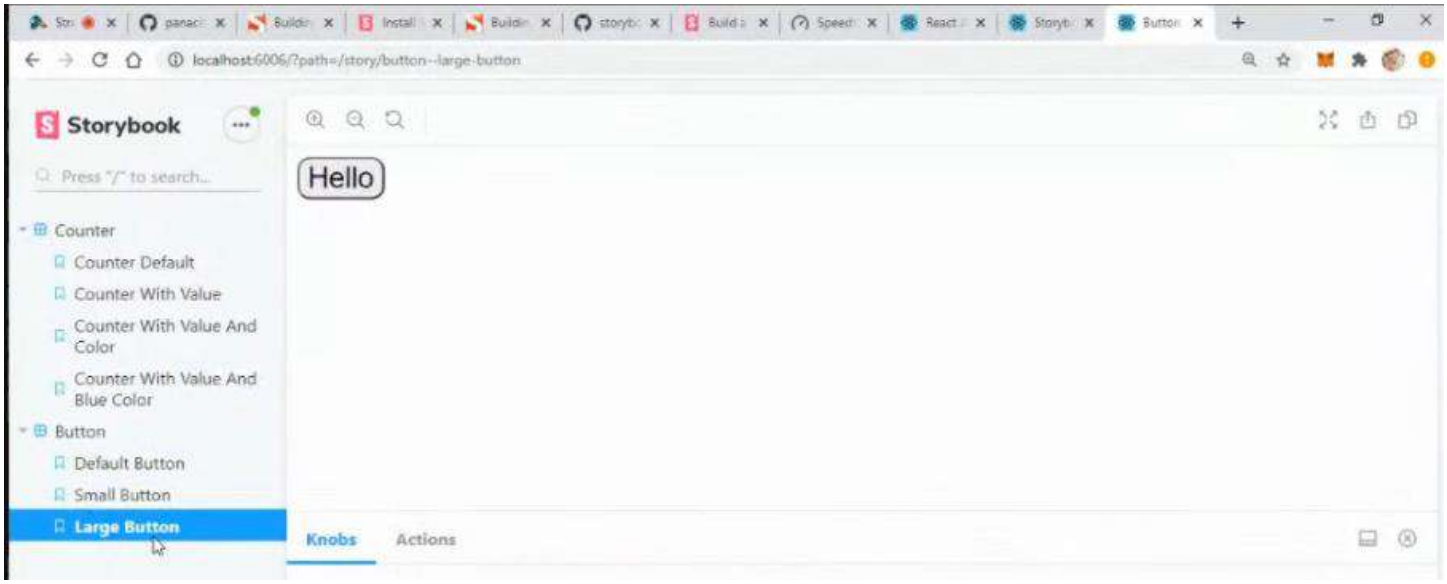
```
import React from 'react';
import './butoon.css';
function Button(props){
 const size=props.size;
 return(
  <button className={["buttonDefault",size.join(" ")}onClick>={()=>
    alert ("Hello")

}}<{props.title}</button>

  </div>
 );
}
export default Button;
```

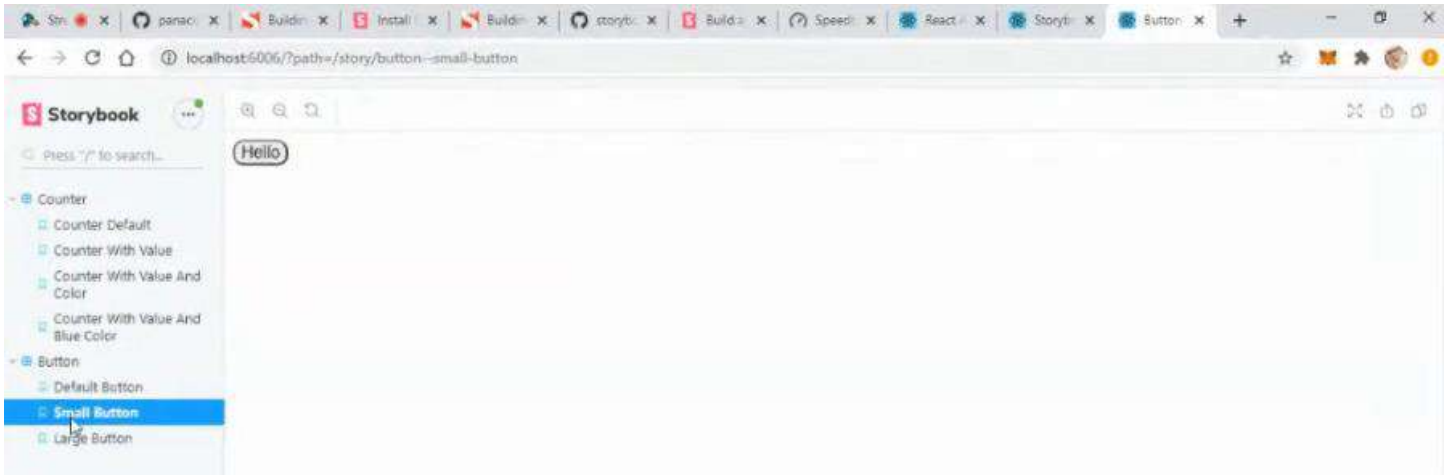
Button.css

```
.buttonDefault{
    border-radius:10px;
}
.small{
   font-size: 16px:
}
.large{
   font-size:22px;
}
```

- Go to the browser and in the text box click to default, large and small button as shown







- add different style on clicks

```
button.js
import React from 'react';
import './butoon.css';
function Button(props){
 const size=props.size;
 return(
  <button className={["buttonDefault",size.join(" ")}
  onClick={()=>{
  props.onClick();
 }}>{props.title}</button>
);

}}<{props.title}</button>

  </div>
```

```
    );
   }
   export default Button;
```

counter.js

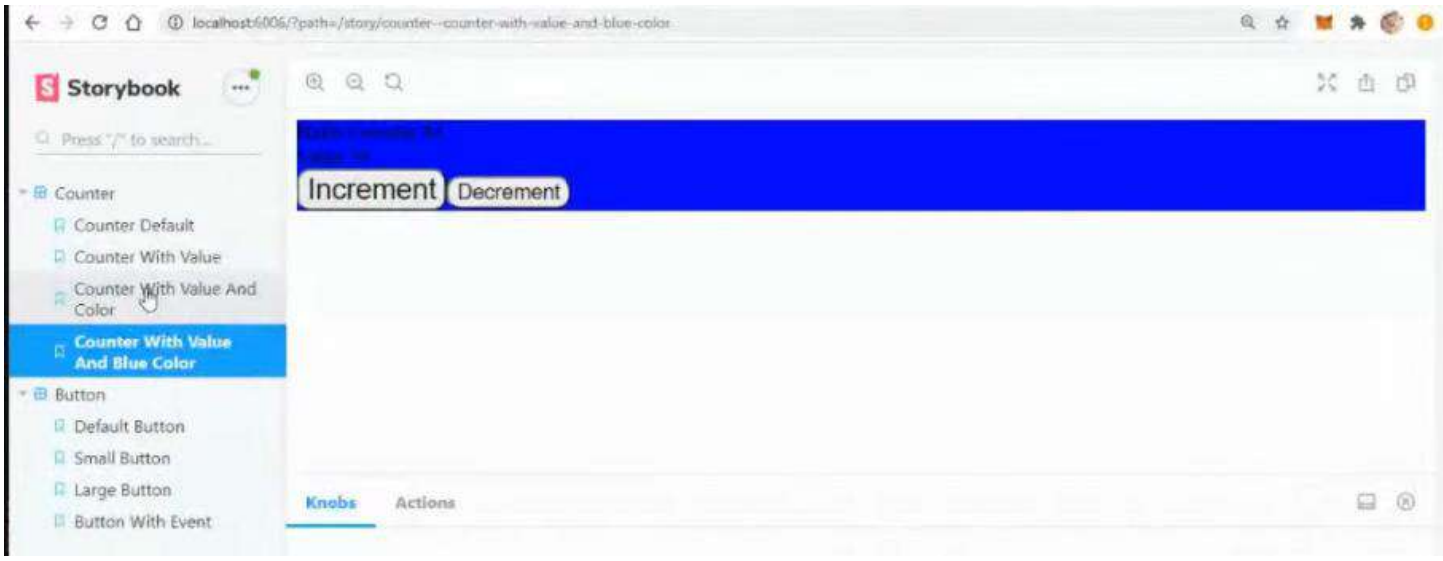
```
import React from 'react';

function Counter(props){
  const [value.setValue]=useState(props.initialValue);
      <br/>)
 return(
  <div style={{backgroundColor:props.backgroundColor}}>
      Hello Counter{props.initialValue}</br>
       Value {value}
      <br/>
      <Button title="Increment" size="large" onClick={()=>{
                  setValue(value+1);
}}/>
       <Button title="decrement"size="small" onClick={()=>{
              setValue(value-1);
}}/>/>
  </div>
 );
}
export default counter;
```

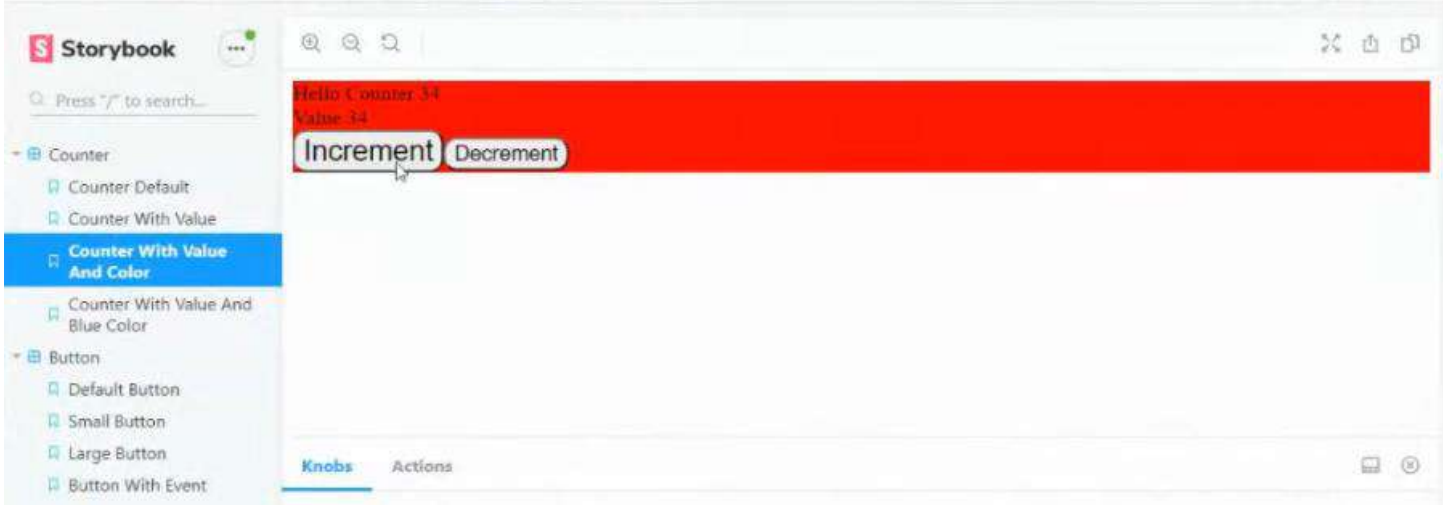
- add another function of ButtonwithEvent in Button.strories.js

Button.stories.js

```
import React from './react';

export default {
  title:"Button"
  component:Button
}
export function DefaultButton(){
  return(
    <Button title="Hello"/>
  );
}
export function smalltButton(){
  return(
    <Button title="Hello" size="small"/>
  );
}
export function largeButton(){
  return(
```

```
    <Button title="Hello" size="large"/>
  );
}
export function ButtonwithEvent(){
  return(
    <Button title="Hello" size="large"onClick={()=>
      alert("Inside button story")
     }}/>
  );
}
```





Adding Knobs Addon To Our Table

The knobs addon is a decorator addon and one of the most used in Storybook. It enables you to change the values (or props) of components without modifying the story function or the component itself.

In this section, you will be adding the knobs addon to our application. The knobs addon eases the stress of having to update the data in your stories manually by setting up a new panel in the storybook explorer where you can easily change the data passed. Without knobs, you'll have to go back to manually modifying your data.

Doing this would be inefficient and it will defeat the purpose of the storybook — especially in cases where those who have access to the stories do not have access to modify the data in the code.

The knobs addon doesn't come installed with a storybook, so you will have to install it as an independent package:

```
yarn add -D @storybook/addon-knobs
```

The installation of the addon knobs requires that the storybook instance be restarted to be effective. Therefore, stop the current instance of the storybook and restart.

- Once the addon has been installed, register it under the addons array in your stories configuration located in .storybook/main.js.

```
module.exports = {
  stories: ['../src/**/*.stories.js'],
  addons: [
    '@storybook/preset-create-react-app',
```

```
        '@storybook/addon-actions',
        '@storybook/addon-links',
        '@storybook/addon-knobs' // Add the knobs addon.
    ],
};
```

- With the addon registered, you can now go-ahead to implement the knobs addon in your table story. The student data is of type object, as a result, you will be using the object type from the knobs addon.

- Import the decorator and the object functions after the previous imports:

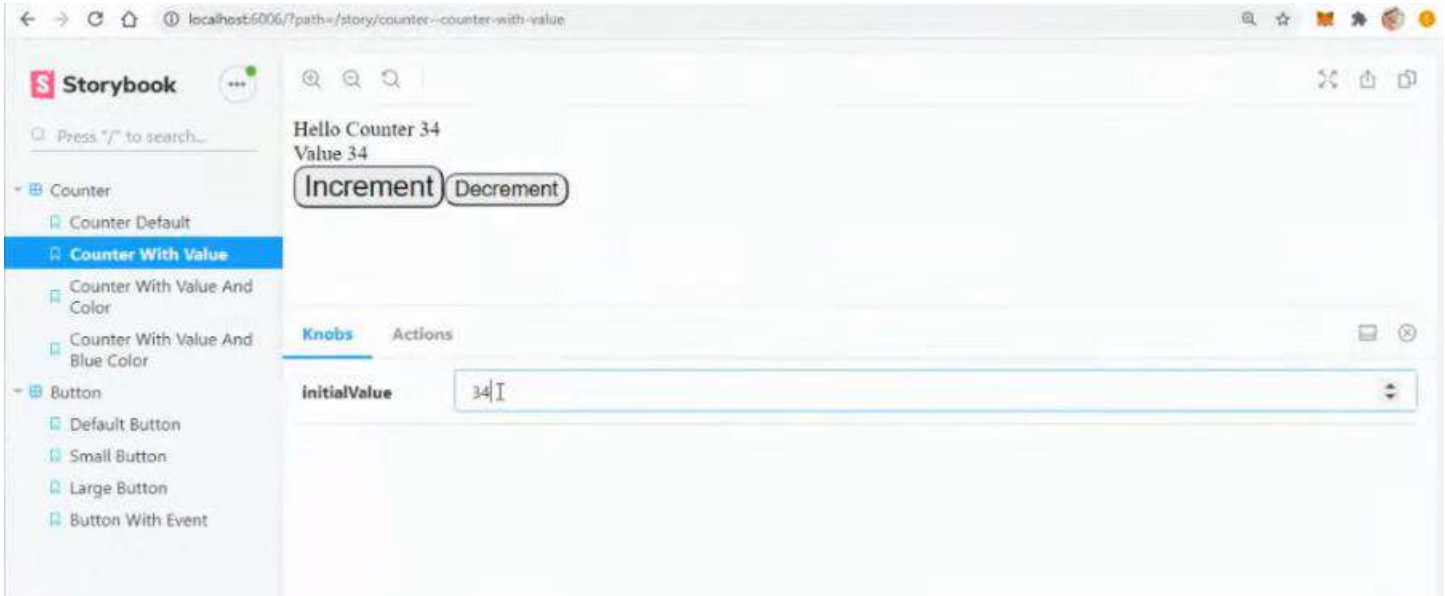
- import { withKnobs, object } from '@storybook/addon-knobs';Just after the component field in the default export, add another field:

```
decorators: [withKnobs]
```

- That is, your story definition object should look like this:

```
export default {

    title: 'Table component',

    component: Table,

    decorators: [withKnobs

Button.stories.js
import React from './react';
import{withknobs,number}from '@storybook/addon-knobs';
export default {
  title:"counter"
  component:"counter"
  decorators:[withKnobs]
}
export function DefaultButton(){
  return(
    <Button title="Hello"/>
  );
}
export function counterWithValue(){
  return(
    <counter initialValue={number('initialValue',34)}/>
  );
}
export function largeButton(){
```

```
  return(
    <Button title="Hello" size="large"/>
  );
}
export function ButtonwithEvent(){
  return(
    <Button title="Hello" size="large"onClick={()=>
      alert("Inside button story")
     }}/>
  );
}
```

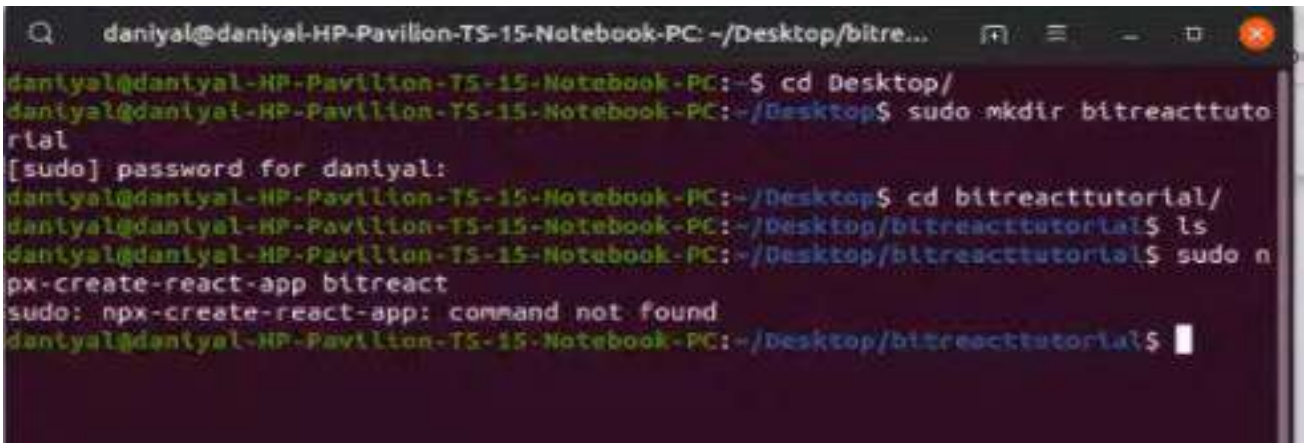


Session 20(A): Component Driven Development (CDD) Part 2

Bit for React

Bit lets you share and sync components between different projects and applications.

- open up terminal
- create a project of app react bit react npx create react bit app



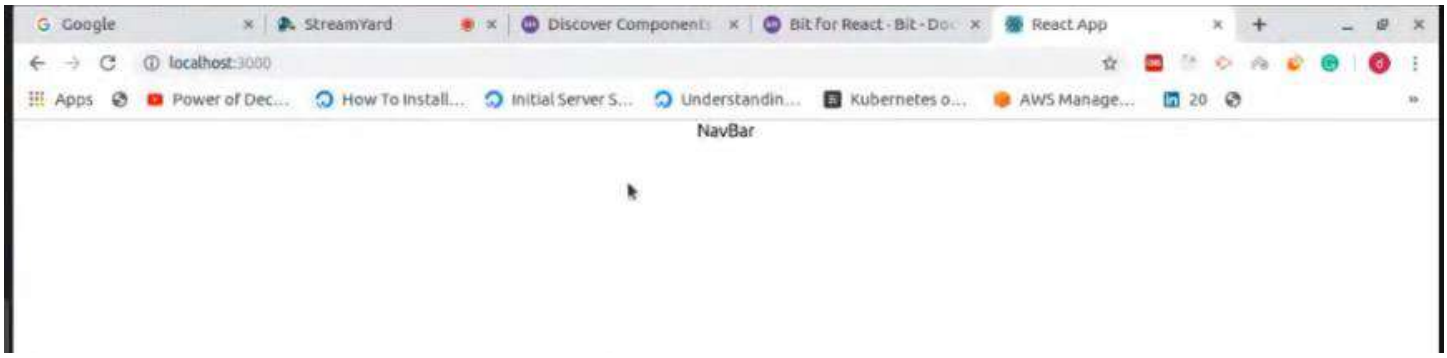
- create a component folder
- create navbar file

index.js

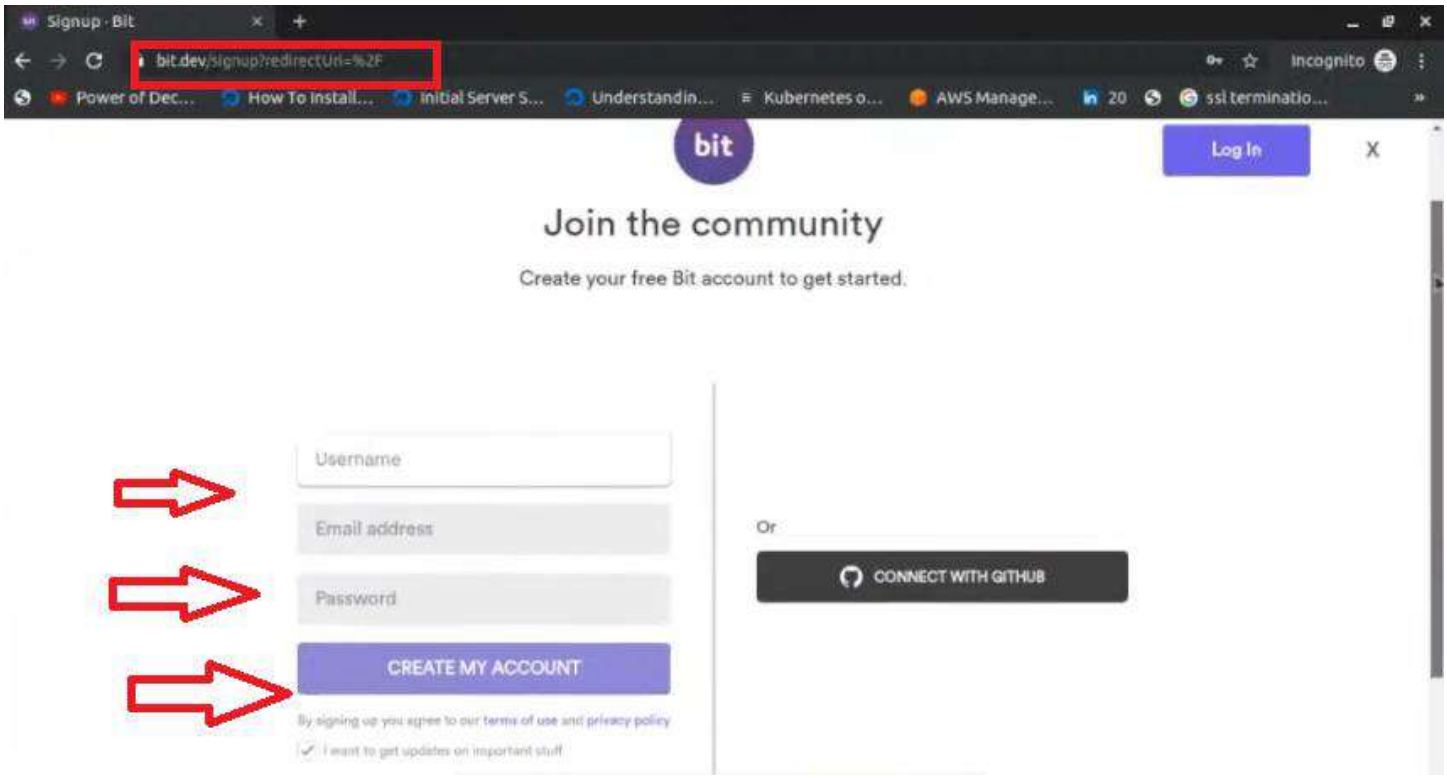
```
import React from './react';


const NavBar=()=>
  return(
    <div>
      NavBar
    </div>
  }
 }
export default NavBar;
```

App.js

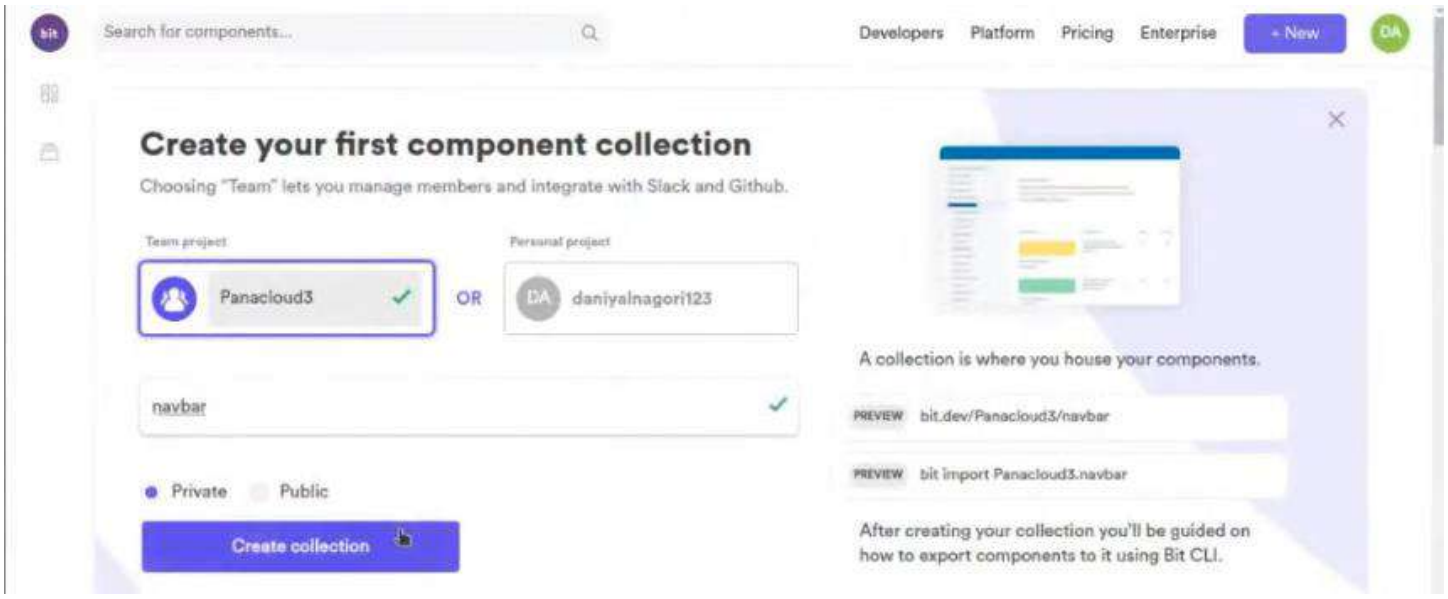
```
import React from 'react';
import './App.css';
import Navbar from "./component/navbar";
function App(){
  return(
    <div className="App">
      <Navbar/>
     </div>
  );
 }
export default App;
```



- create a bit account
- for that purpose goto bit.dev
- enter username
- email Address
- password
- click to create my account

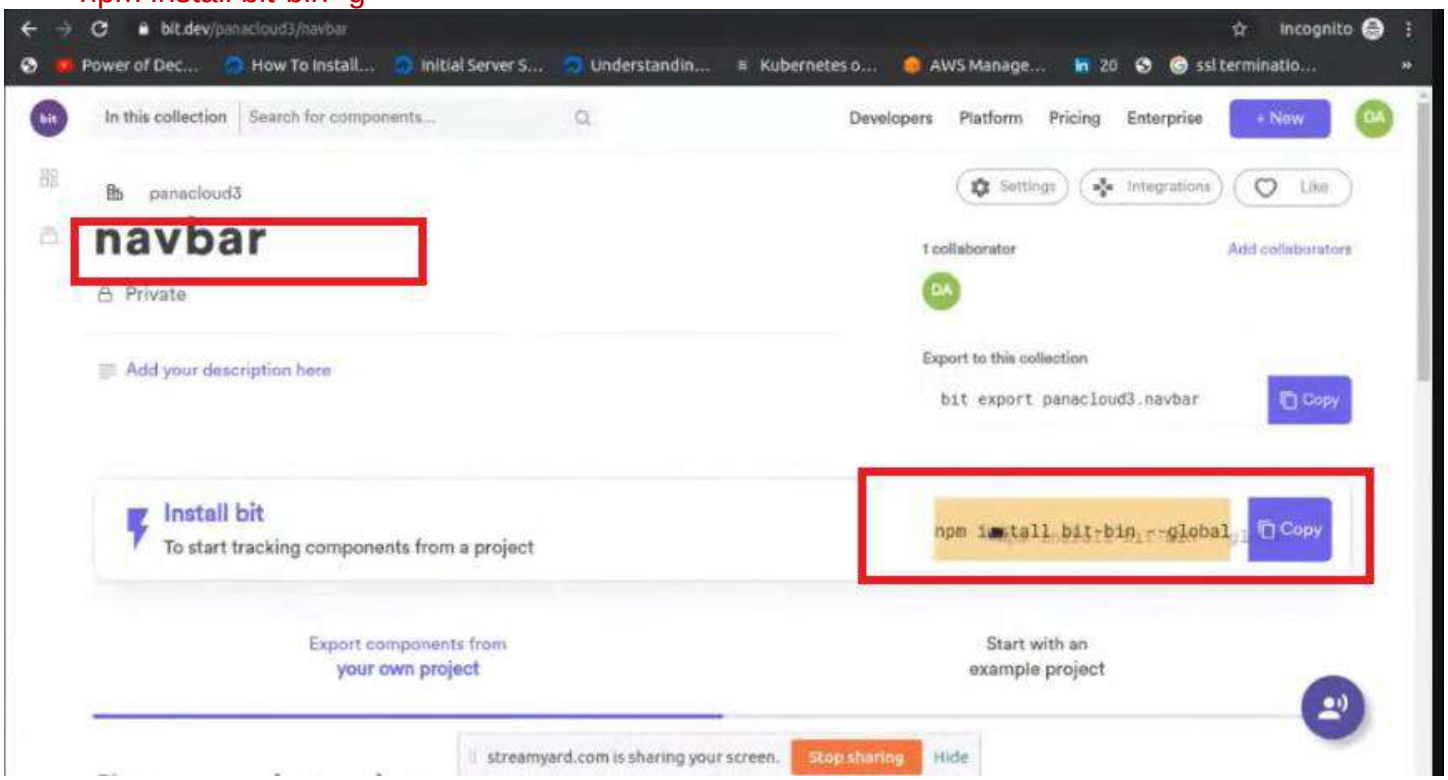


- create your first component collection
- add team project and name the organization
- enter either public or private
- Click to create a collection



- So the account has been created: install bit bin from the given command in your system

npm install bit-bin -g



- Visit Install Bit for other installation methods.

If you have Bit installed, verify the installation by running the command:

bit --version

- Login to Your Bit Account

Authenticate Bit to your bit.dev account. From the command-line run:

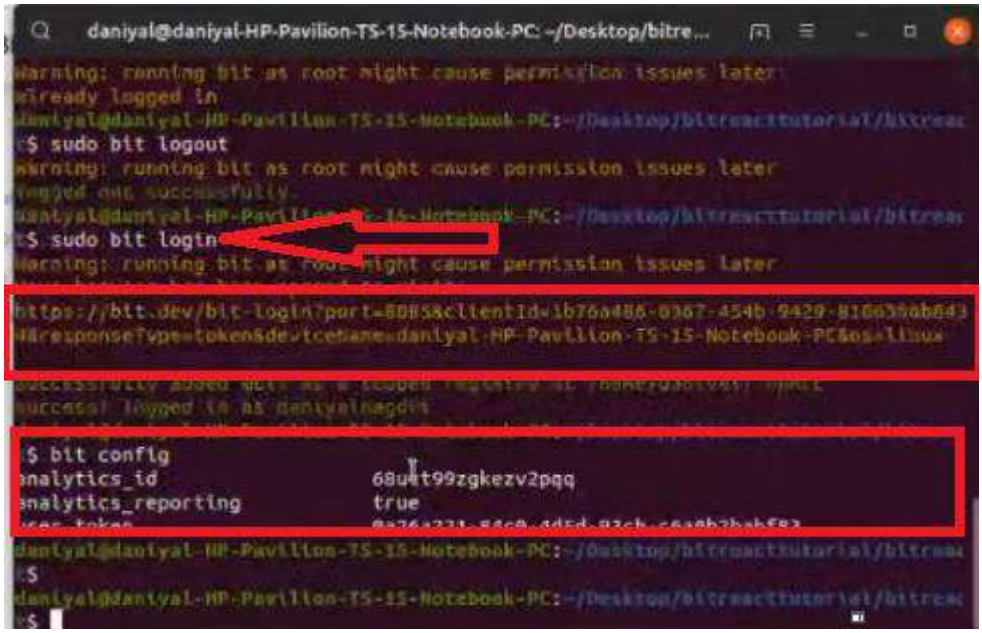
```
bit login
```

This will open your browser where you can log into your account. If you are already logged in, a success message will be displayed. You are now ready to start using Bit.

As part of the login process, Bit sets up your local configuration. You can see your configuration by typing:

```
bit config
```

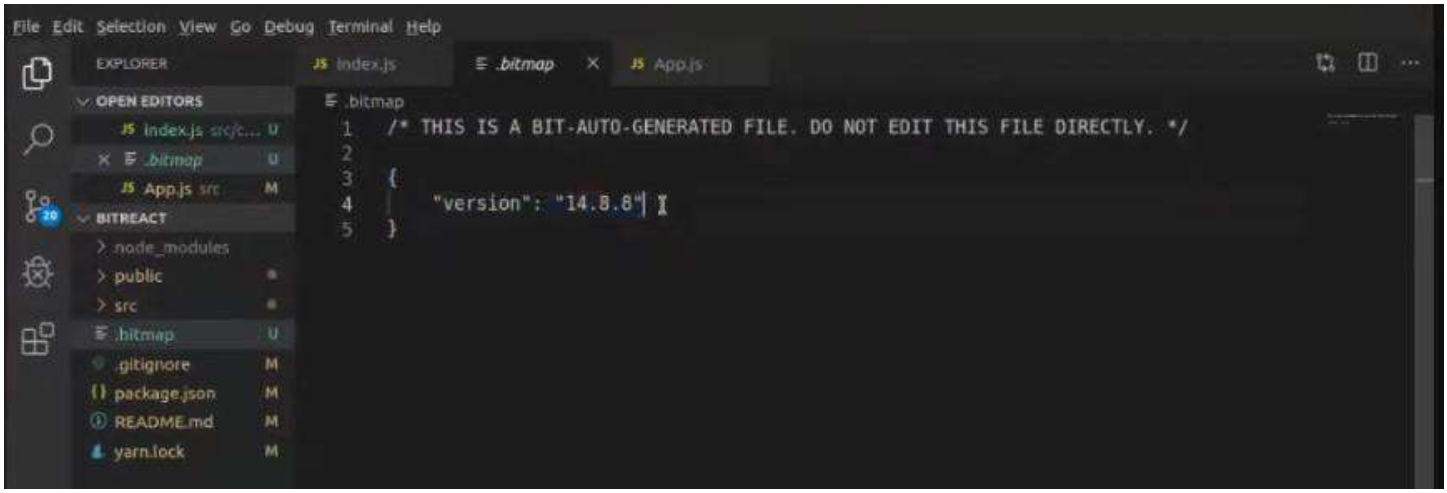
Besides, Bit adds the npm registry used by Bit to your npmrc configuration. (by default located in $HOME/.npmrc according to your OS).



- create a workplace by using this command

```
$ bit init --package-manager yarn
successfully initialized a bit workspace.
```

- so in our project, we have BBITREACT and version as well



- **Share a React Component**

Now, we will track the product-list component from the React tutorial project. The component will be tracked with the id product-list.

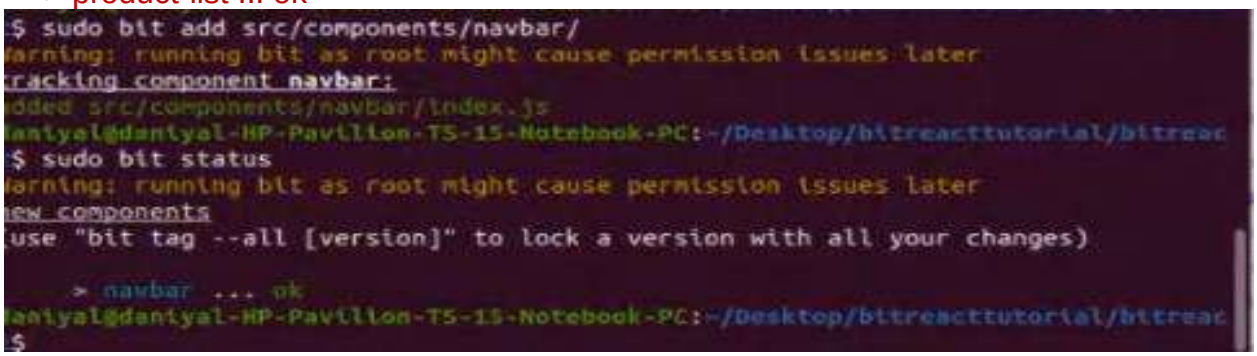
- **Track a New Component**

To track the product list component, we will need to tell Bit about the component and the files that are related to it. As all the files are located under the product-list directory, the simplest way is to add all the files in the directory to your component. Bit will create a component named after the directory name.

```
$ bit add src/components/product-list
tracking component product-list:
added src/components/product-list/index.js
added src/components/product-list/product-list.css
added src/components/product-list/products.js
```

When creating new components, you need to make sure that you have properly added all of the files required for the component. Bit can analyze the component for you and verify that all files are included. You can do that by checking the status of the component:

```
$ bit status
new components
(use "bit tag --all [version]" to lock a version with all your changes)

    > product-list ... ok
```



- **Install React Compiler**

So far, we have provided Bit with the source file of the component. But to consume the files in other projects, the component needs to be built.

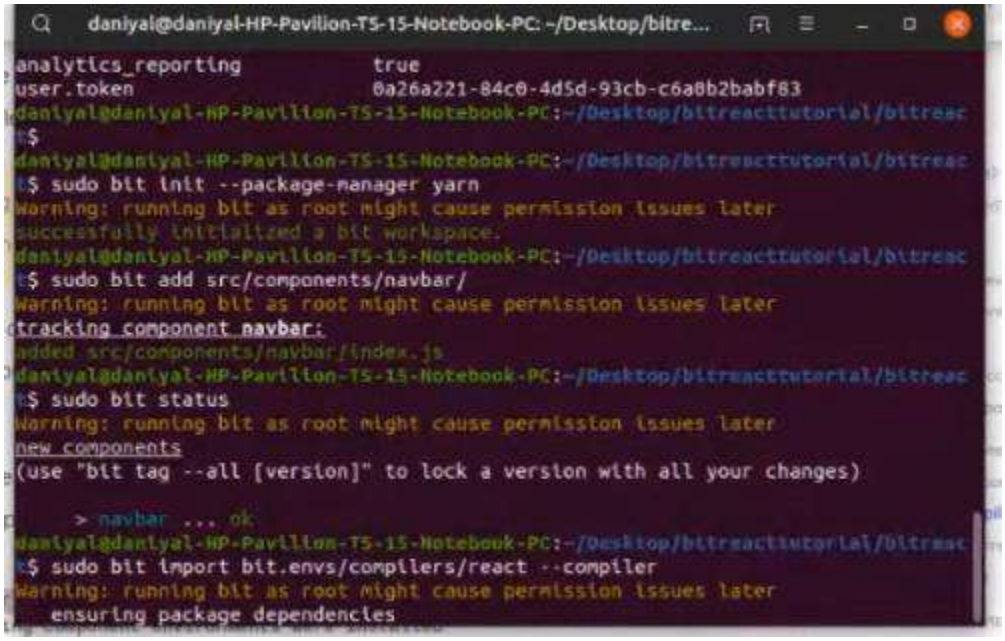
- **For building the React component, you'll need the React compiler.**

Install the compiler and run this command inside the React tutorial repository:

```
$ bit import bit.envs/compilers/react --compiler
the following component environments were installed
- bit.envs/react@0.1.3
```

The React compiler is now set as the default compiler for the Bit workspace inside this repository. You can check the package.json and verify that the compiler is installed by locating the following entry in the Bit section:

```
{
  "env": {
    "compiler": "bit.envs/compilers/react@1.0.2"
  },
}
```



- To build your component, run this command inside your react project:

```
bit build
```

- Run this command to tag component

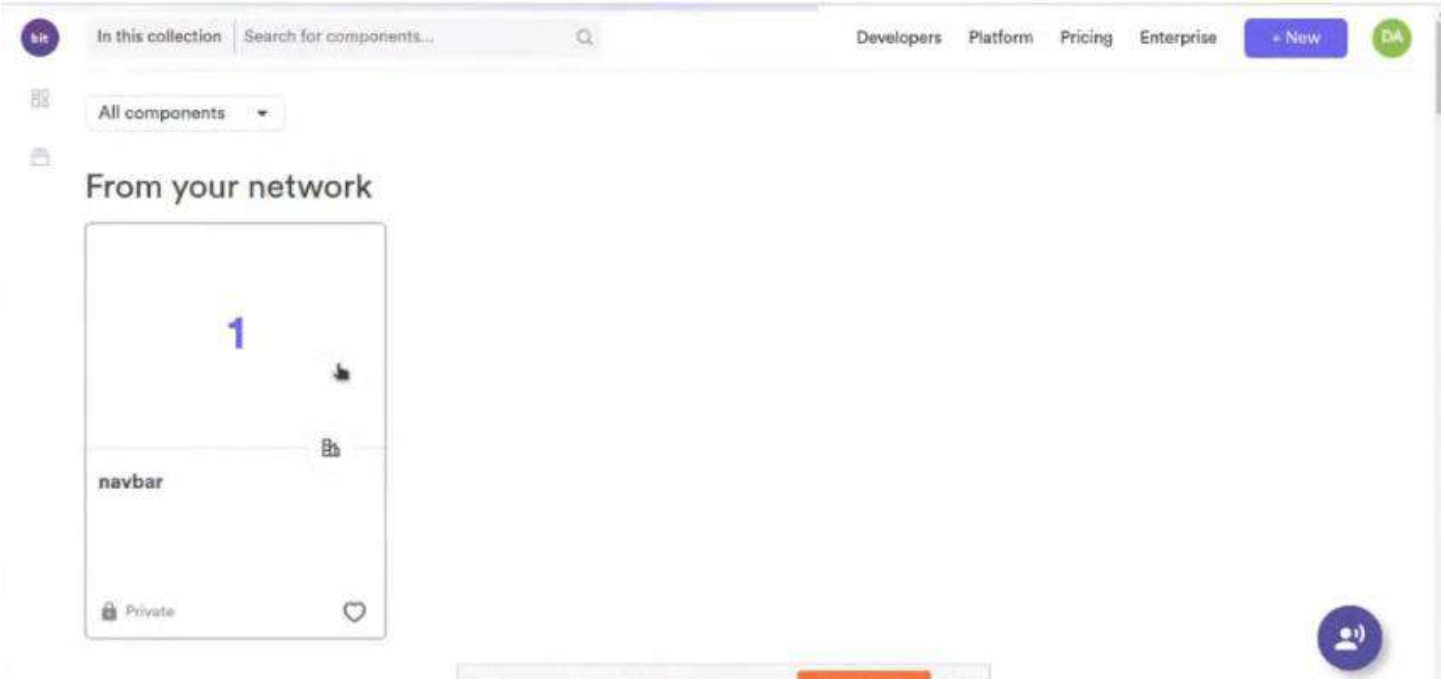
```
$ sudo bit tag --all 0.0.1
```

$ bit tag --all 0.0.1

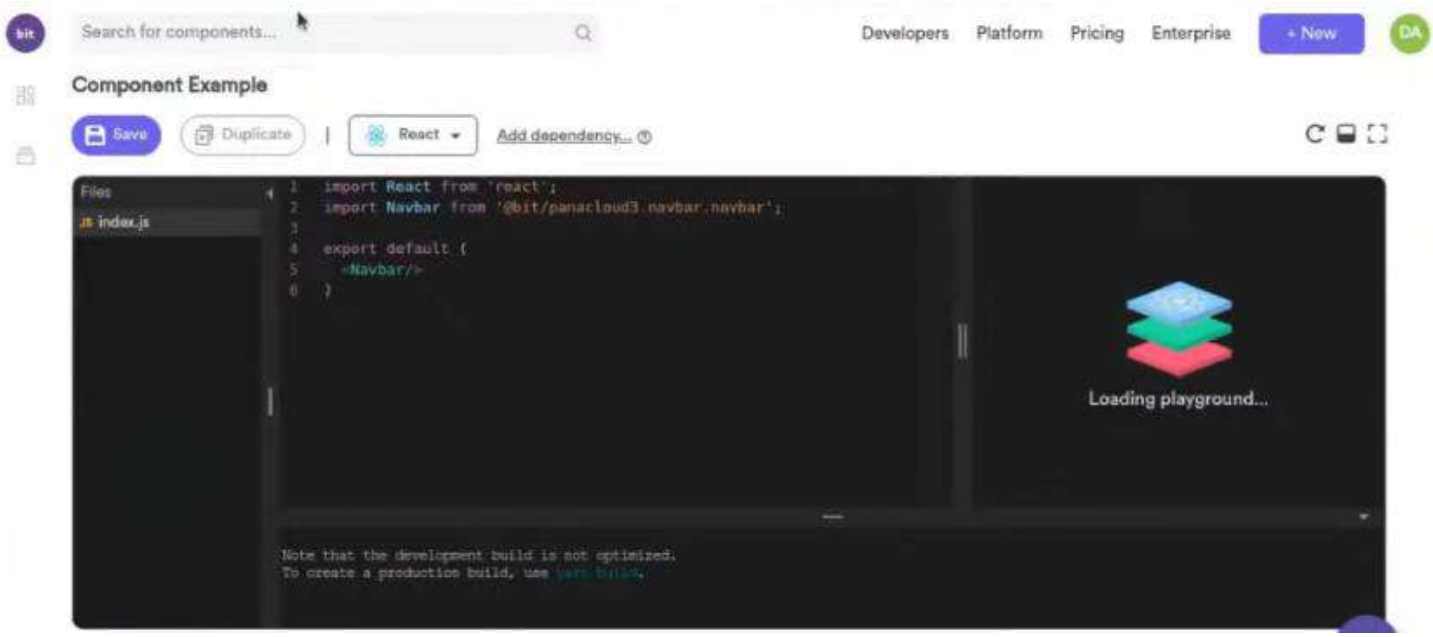
- To export the component to your bit.dev collection, we will use the export command and the full name of the collection, structured as <organization name>.<collection>

```
$ sudo bit export panacloud3.navbar
warning: running bit as root might cause permission issues later
  exporting component
```

- Go to the collection



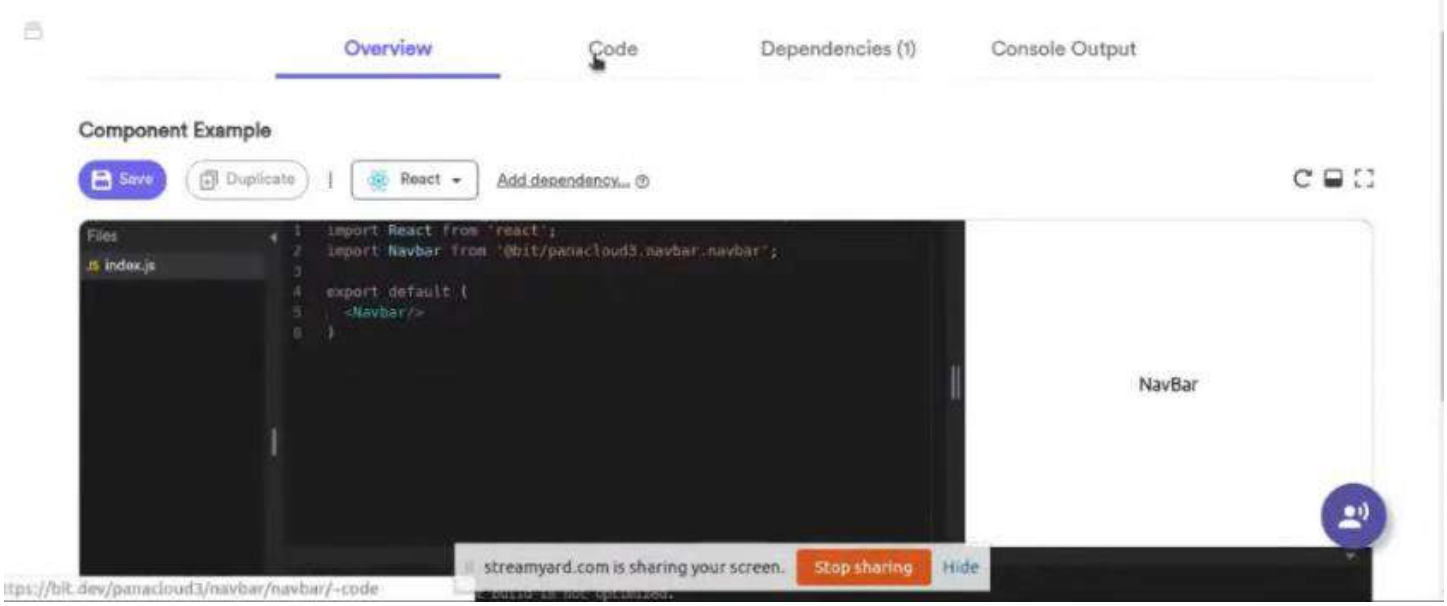
- Click to navbar and this will open as



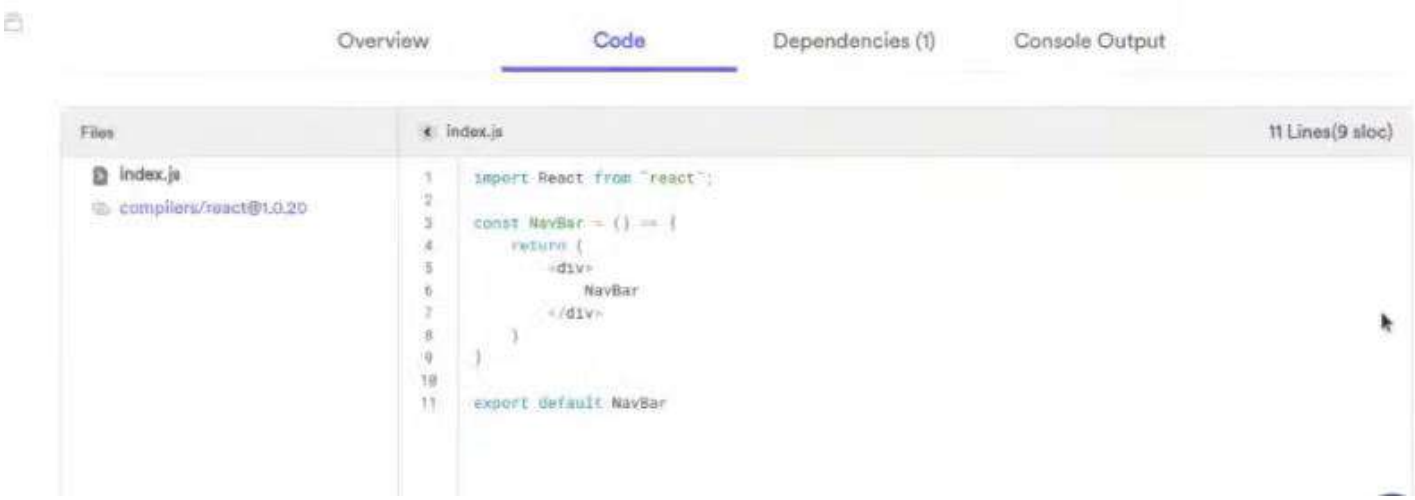
- create another project and import this component in it by the following command

```
kdir bitreactanotehrproject
danlyal@danlyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/bitreacttutorial$ cd bit
reactanotehrproject/
danlyal@danlyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/bitreacttutorial/bitreac
tanotehrproject$ ls
danlyal@danlyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/bitreacttutorial/bitreac
tanotehrproject$ sudo npx create-react-app bitcomponent
[..................] \ fetchMetadata: still resolveWithNewModule create-react-ap
```

- So whatever we exported the component is here



Code is there



- If you want to see all the components you have you can run:

  bit list

```
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC: ~/Desktop/bitre...
Compiled successfully!

You can now view bitreact in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.100.17:3000

Note that the development build is not optimized.
To create a production build, use yarn build.

^C
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/bitreacttutorial/bitreac
t$ bit list
found 1 components in local scope

| component ID | local version | used version |
|---|---|---|
| panacloud3.navbar/navbar | 0.0.1 | 0.0.1 |

daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/bitreacttutorial/bitreac
t$
```

So far we have two projects there

```
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC: ~/Desktop/bitre...
piled successfully!

can now view bitreact in the browser.

ocal:            http://localhost:3000
n Your Network:  http://192.168.100.17:3000

e that the development build is not optimized.
create a production build, use yarn build.

iyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/bitreacttutorial/b
bit list
nd 1 components in local scope

| component ID | local version | us ve on |
|---|---|---|
| panacloud3.navbar/navbar | 0.0.1 | 0. |

iyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/bitreacttutorial/b
```

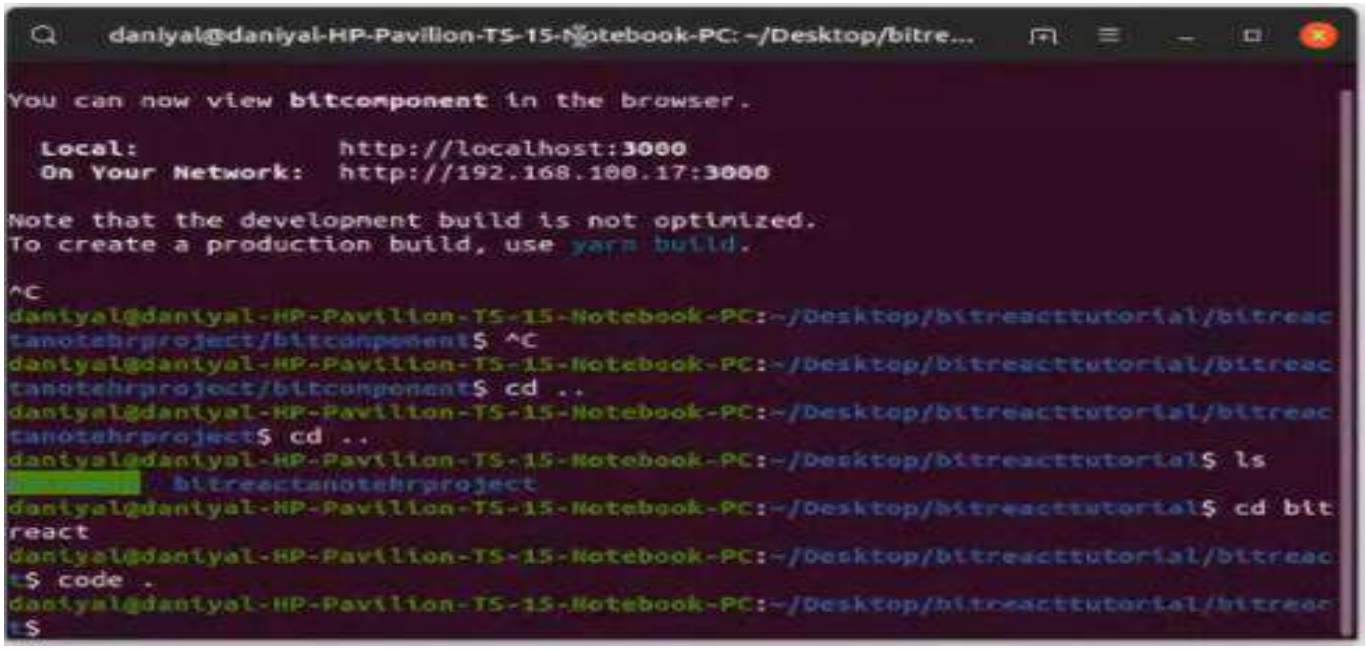
```
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC: ~/Desktop/bitre...
  yarn test
    Starts the test runner.

  yarn eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back

We suggest that you begin by typing:

  cd bitcomponent
  yarn start

Happy hacking!
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/bitreacttutorial/
tanotehrproject$ ls
bitcomponent
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/bitreacttutorial/
tanotehrproject$ cd bitcomponent/
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/bitreacttutorial/
tanotehrproject/bitcomponent$ ls
node_modules  package.json  public  README.md  src  yarn.lock
daniyal@daniyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/bitreacttutorial/
tanotehrproject/bitcomponent$
```

- Now in the first project run this command

```
tanotehrproject/bitcomponent$ sudo yarn add @bit/panacloud3.navbar.navbar
```

```
App.js
import React from 'react';
import './App.css';
import Navbar from "..@bit/panacloud3.navbar.navbar"
function App(){
  return(
    <div className="App">
      <Navbar/>
    </div>
  );
}
```

- now run this code of bit react



- create a header component
- create a index.js file in it

index.js

```
import React from 'react';
const Header =()=>{
  return(
    <div>
      Hello from Header
    </div>
  )
}
export default Header
```

- export this component of header and tack this
- build bit command and run this

```
danyal@danyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/bitreacttutorial/bit
$ sudo bit add src/components/header/
Warning: running bit as root might cause permission issues later
tracking component header:
added src/components/header/index.js
danyal@danyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/bitreacttutorial/bit
$ sudo bit build
Warning: running bit as root might cause permission issues later
  building components...
   building component - header
```

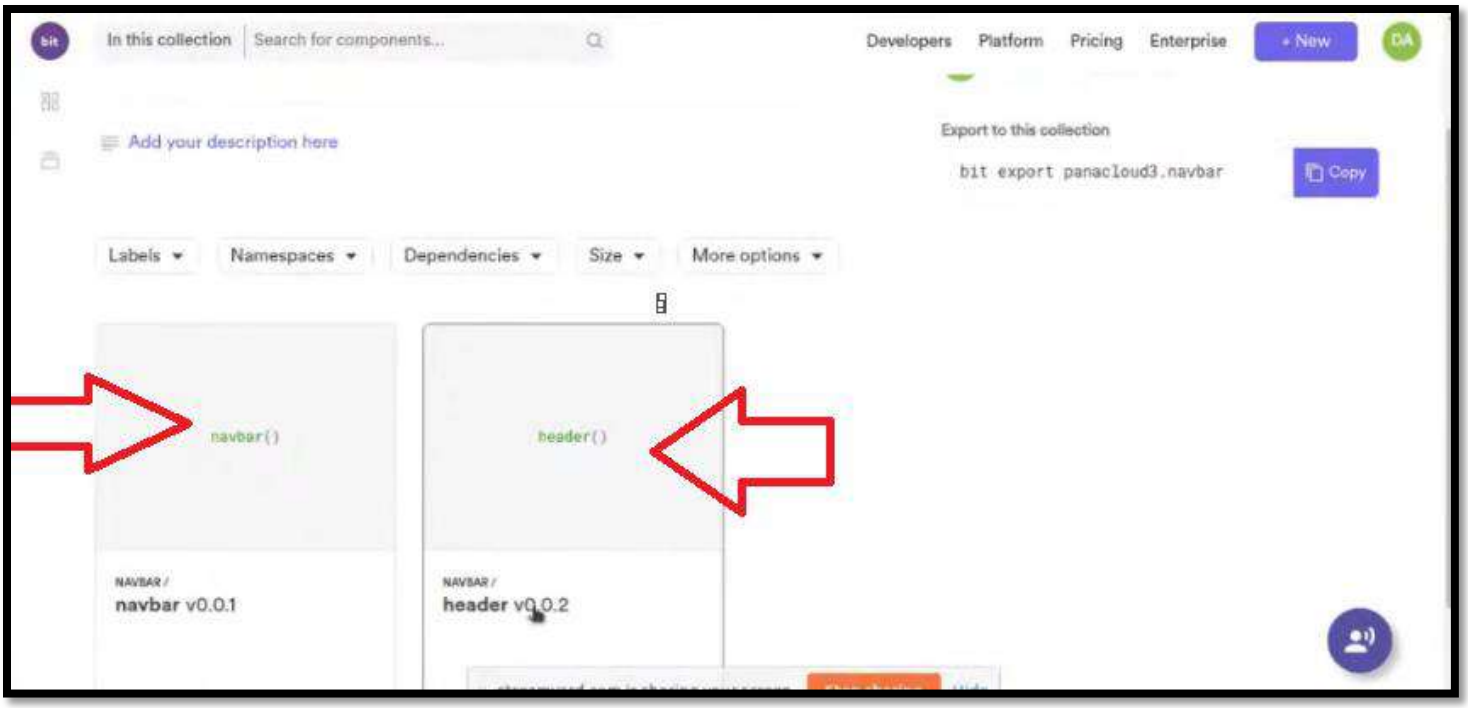
- tag it using header 0.0.2

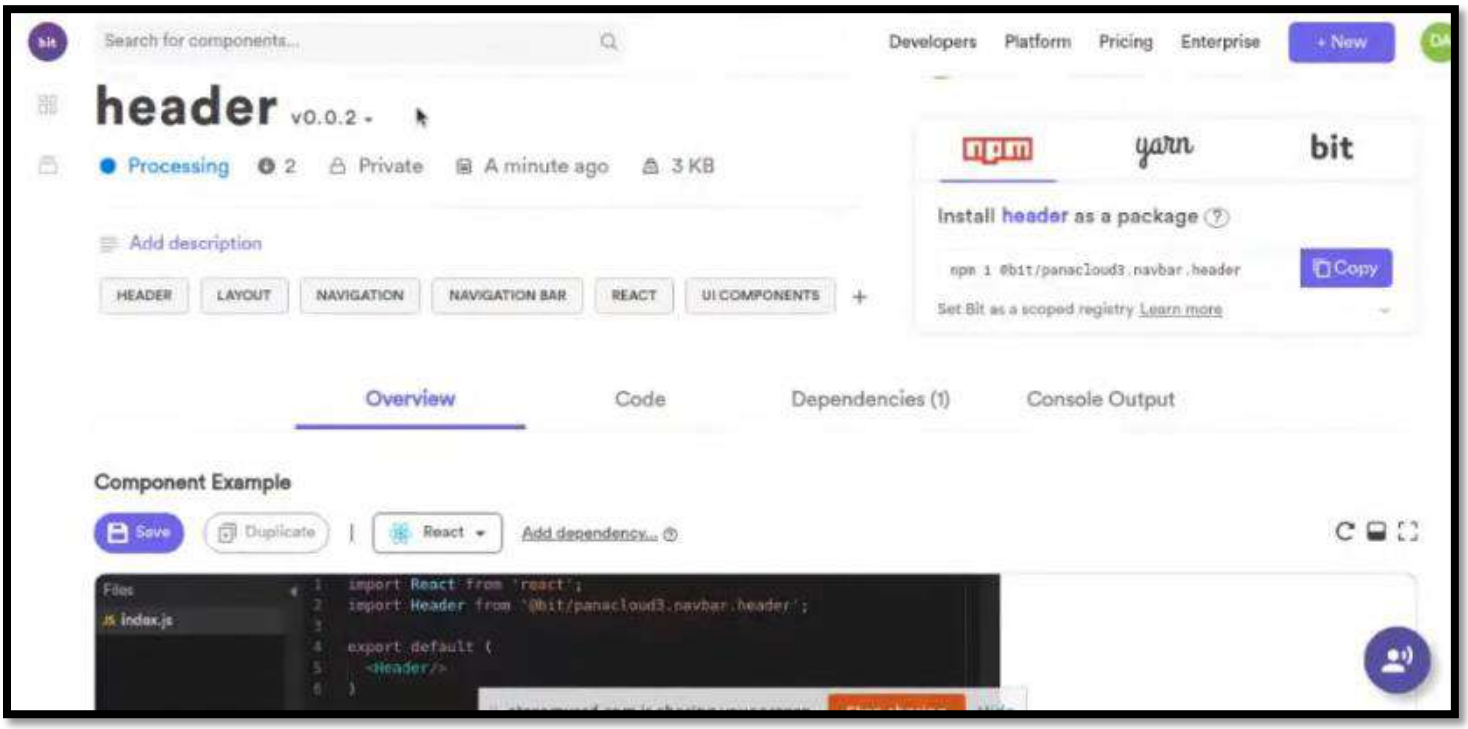
```
danyal@danyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/bitreacttutorial/bit
t$ sudo bit tag header 0.0.2
Warning: running bit as root might cause permission issues later
  importing components(node:11101) [DEP0005] DeprecationWarning: Buffer() i
precated due to security and usability issues. Please use the Buffer.alloc()
ffer.allocUnsafe(), or Buffer.from() methods instead.
✔ building component - header
1 component(s) tagged
(use "bit export [collection]" to push these components to a remote")
(use "bit untag" to unstage versions)

new components
(first version for components)
     > header@0.0.2

danyal@danyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/bitreacttutorial/bit
t$ sud
```

Now export it

```
t$ sudo bit export panacloud3.navbar
Warning: running bit as root might cause permission issues later
exported 1 components to scope panacloud3.navbar
danyal@danyal-HP-Pavilion-TS-15-Notebook-PC:~/Desktop/bitreacttutorial/bit
t$
```

- Now go to the browser, refresh the page and look there are two components of navbar and header respectively


In this collection
Search for components...
Developers
Platform
Pricing
Enterprise
+ New
Add your description here
Export to this collection
bit export panacloud3.navbar
Copy
Labels
Namespaces
Dependencies
Size
More options
navbar()
header()
NAVBAR /
navbar v0.0.1
NAVBAR /
header v0.0.2


Search for components...
Developers
Platform
Pricing
Enterprise
+ New
header v0.0.2
Processing
2
Private
A minute ago
3 KB
npm
yarn
bit
Install header as a package
npm i @bit/panacloud3.navbar.header
Copy
Set Bit as a scoped registry Learn more
Add description
HEADER
LAYOUT
NAVIGATION
NAVIGATION BAR
REACT
UI COMPONENTS
Overview
Code
Dependencies (1)
Console Output
Component Example
Save
Duplicate
React
Add dependency...
Files
index.js
import React from 'react';
import Header from '@bit/panacloud3.navbar.header';
export default (
<Header/>
)

Component Example
Save
Duplicate
React
Add dependency...
Files
index.js
import React from 'react';
import Header from '@bit/panacloud3.navbar.header';
export default (
<Header/>
)
Hello from daniyal
Note that the development build is not optimized.
To create a production build, use

Session 21

What are modern applications and how to develop them?

Modern App:

A modern application is one that supports multiple clients – whether the client is a UI using the React JavaScript library, a mobile app running on Android or iOS, or a downstream application that connects to the application through an API.

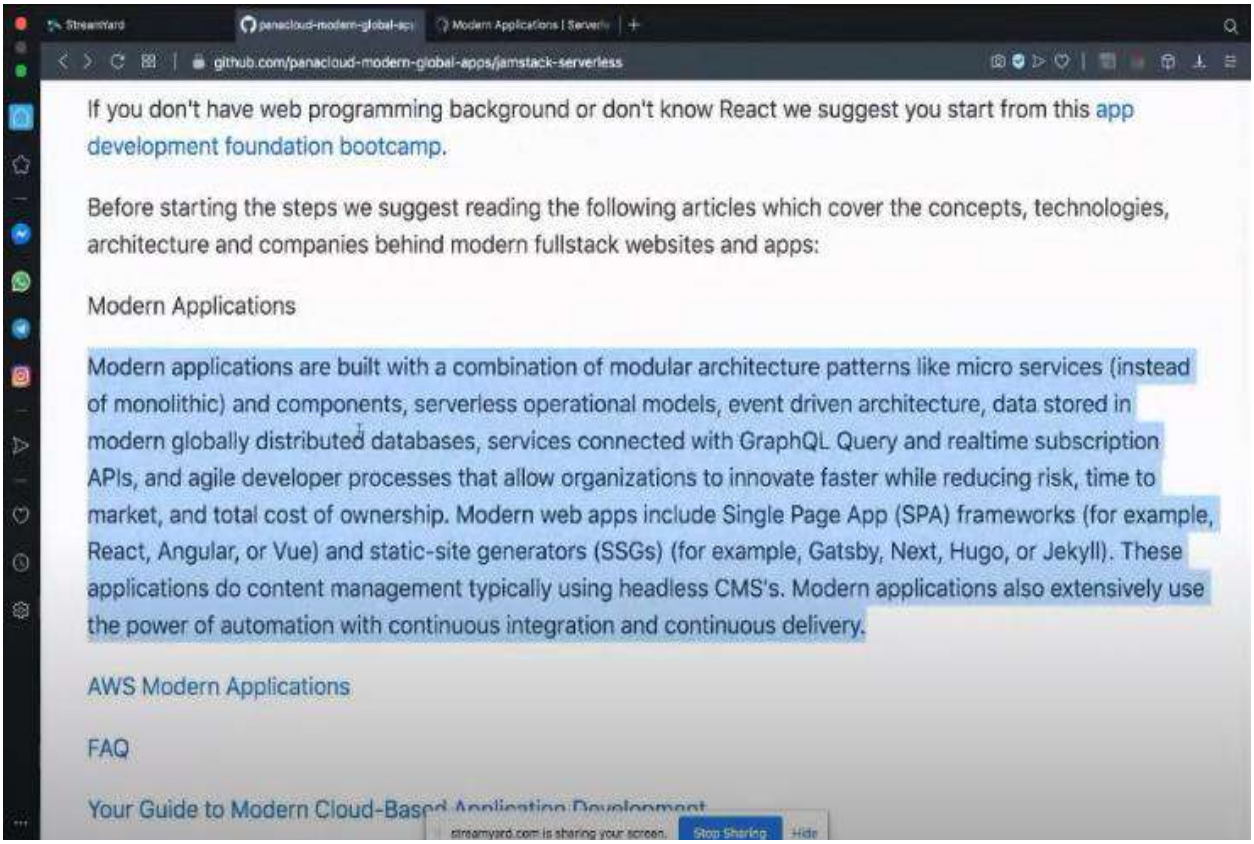
Monolithic App:

Monolithic applications, also referred to as monolithic architecture, describe a single-tiered software application where different components are combined into a single application on a single platform, thus making it self-contained and independent from other applications.

Difference between monolithic and modern app:

Nowadays the industry suggests you work on modern ap instead of monolithic apps. A monolithic app is the package of everything like bank end, front foot, and everything else, but it is more difficult to make changes or do innovation in the monolithic app, while on the other hand in the modern app it would be easier and faster. In a monolithic app, all the features are dependent on others, while in the modern app the features are independent or develop by themselves. Monolithic apps are very difficult to scale

Modern Application



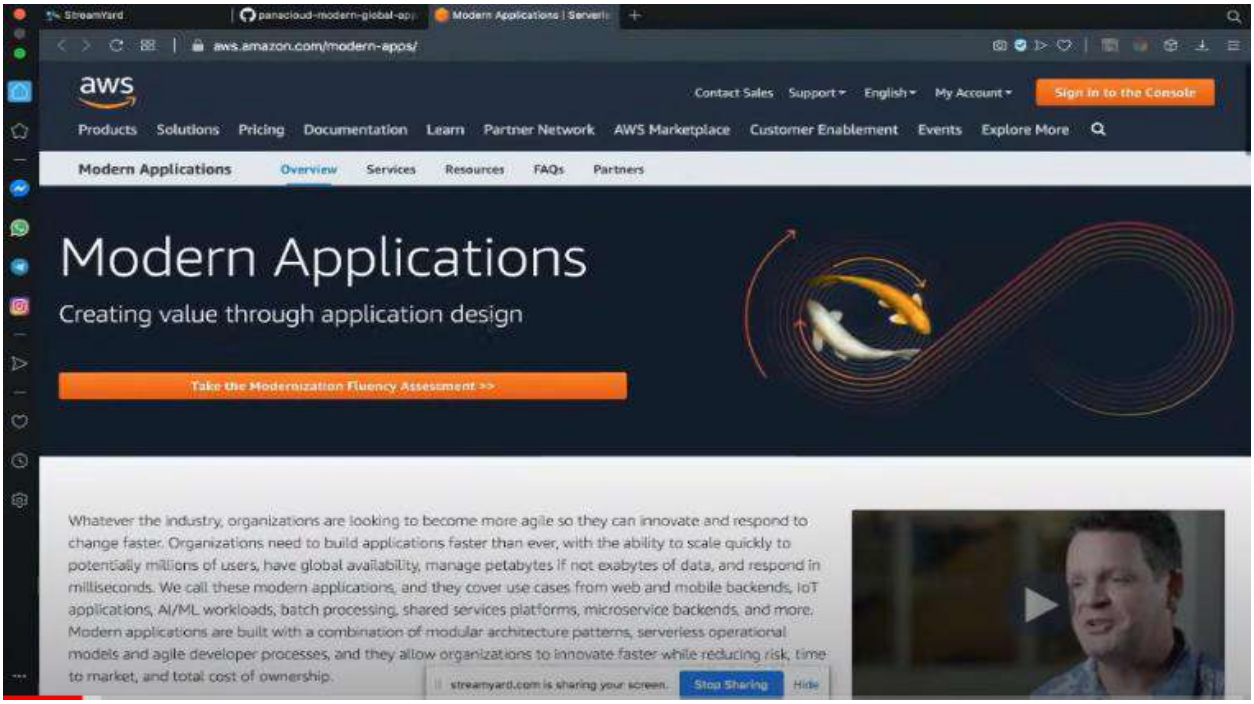
Modern App Developing

Modern app develop both the single page and static page site.These management includes headline CSS e.g contentful.

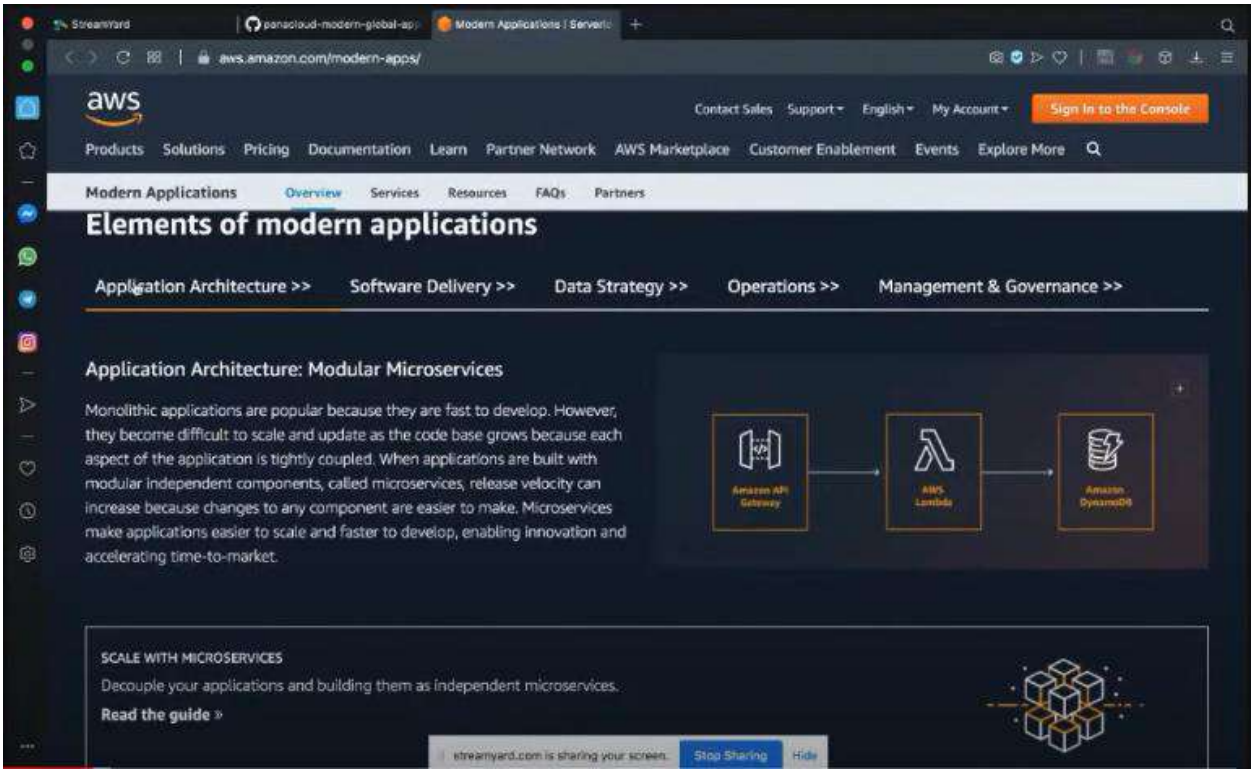
Three Client site framework

1. Create react app
2. Gatsby
3. Next.js

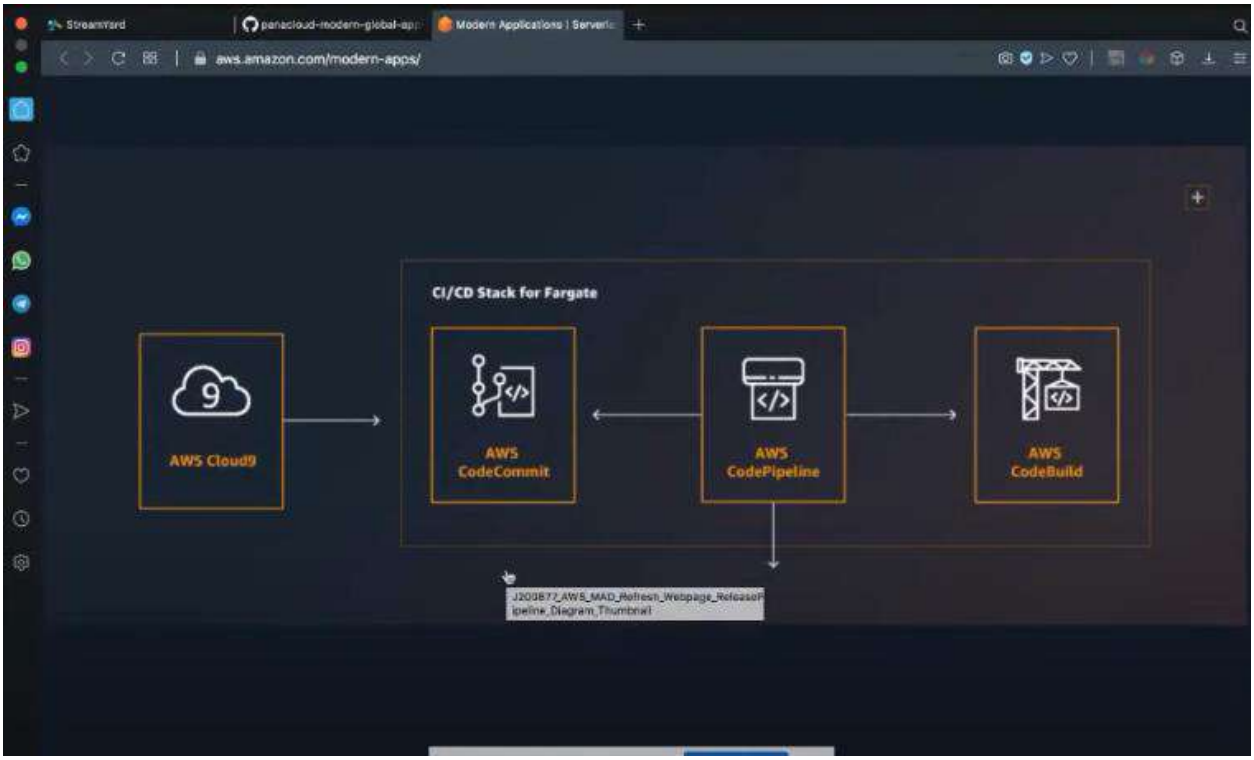
AWS Application



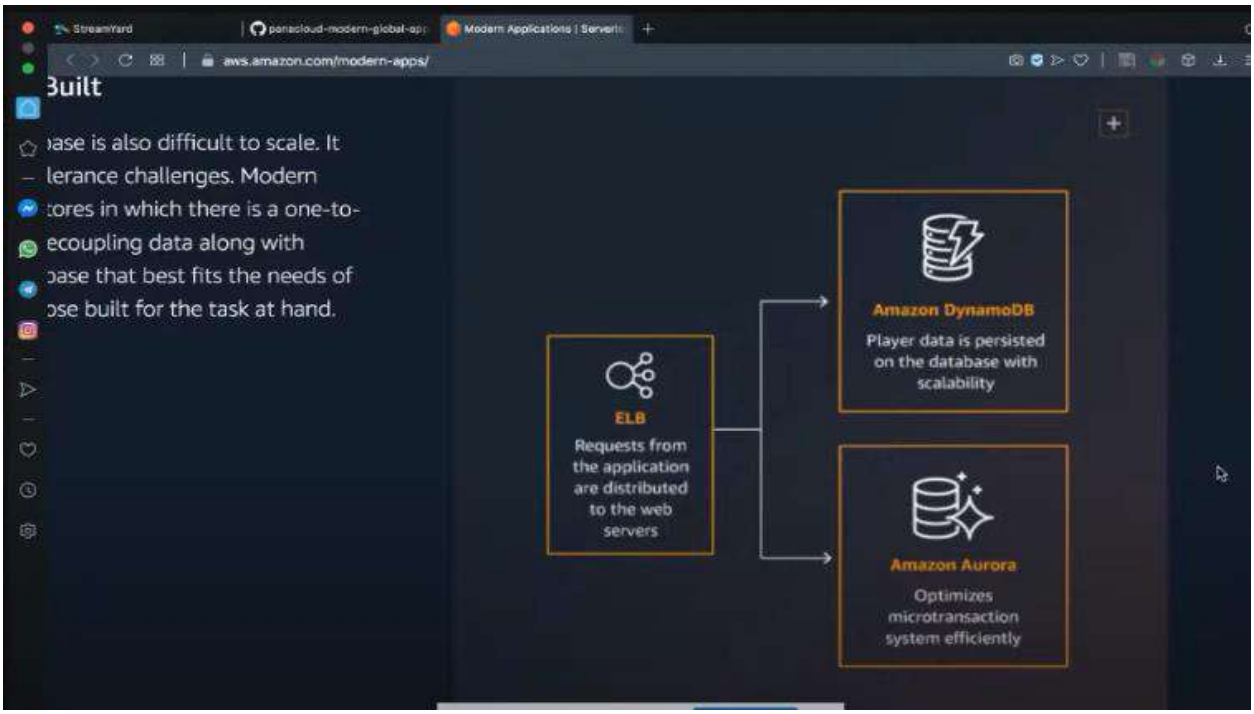
Different elements of modern application



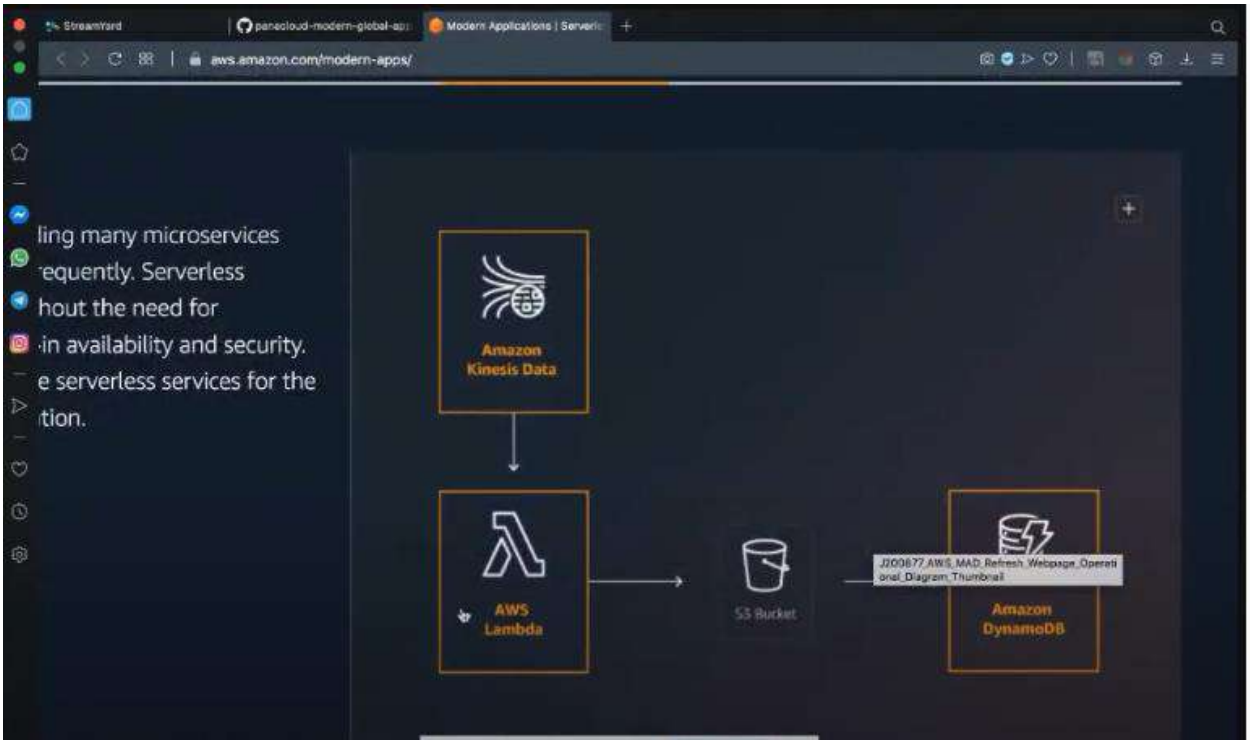
AWS CICD



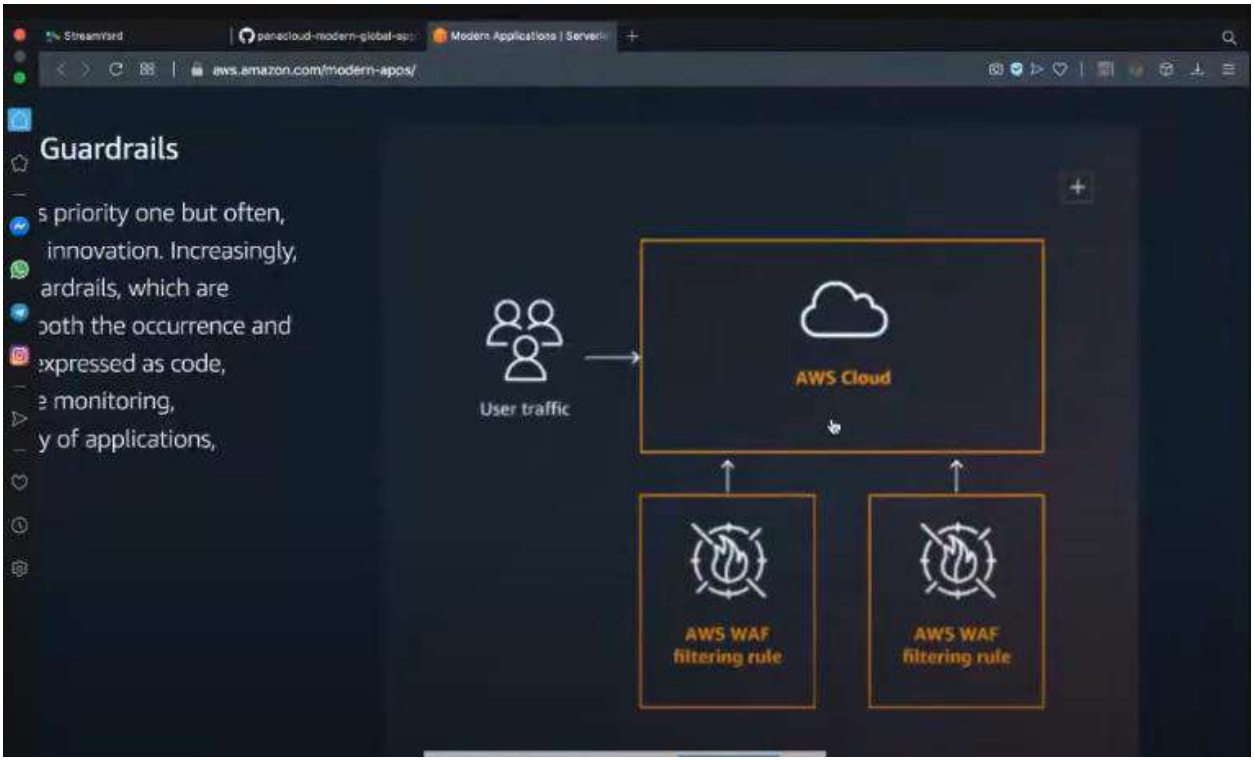
Data Strategy



Operations



Management and governence



Concepts

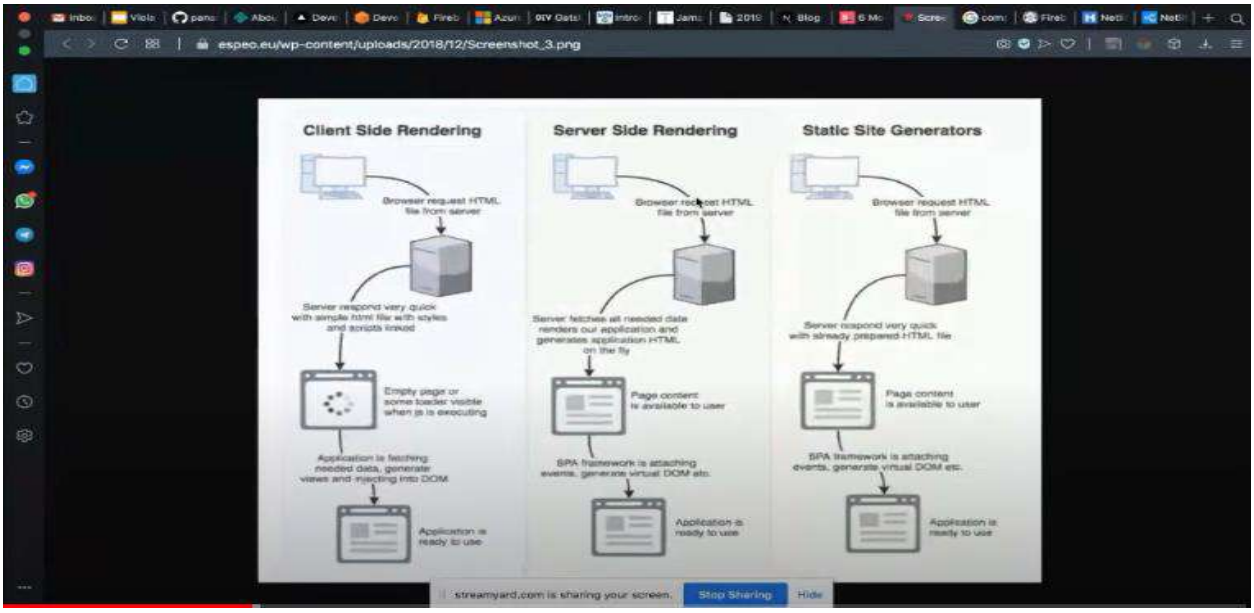
1. Server site rendering
2. Server site generation
3. Client site rendering

Client site rendering

- Complete HTML Recieved 9(java script)
- React or Angular work as CSR
- Pop-up
- Single page Webapp

Server Site Rendering

- React convert page in HTML
- Received request on a new page
- Helps in SEO



Static Site Generation

- Compile the code
- Compilation Time
- Pages Generation
- Most faster than other renderings
- All process done at build time

Static site generation is easily accessable to server?

Whenever you recieved the data, it will render the data. Every request is on the server. Have a static server you don't need any helper.

For developing a App

If you want static site, Gatsby is thebest option because it works on both CSR ans SSG.

Jamstack

It stands for 3 words

- Java script
- API'S
- Pre render mark up

Explanation

- Prebuild API
- Combination of existing process

Gatsby

- Build pages on compilation time
- SSD

What is Contentful?

Content management system where we put our data, manage our data. It is headless system. It will provide all of the site data.

Netlify

- Build the data

- Deploy the data
- Contentful Facilities
- Complete GrapiQL
- Integrate the Github repository
- CISD
- Virtual Machine

Serverless Lamda Function

- Same ascloud function
- Run the function

Session-32

Gatsby

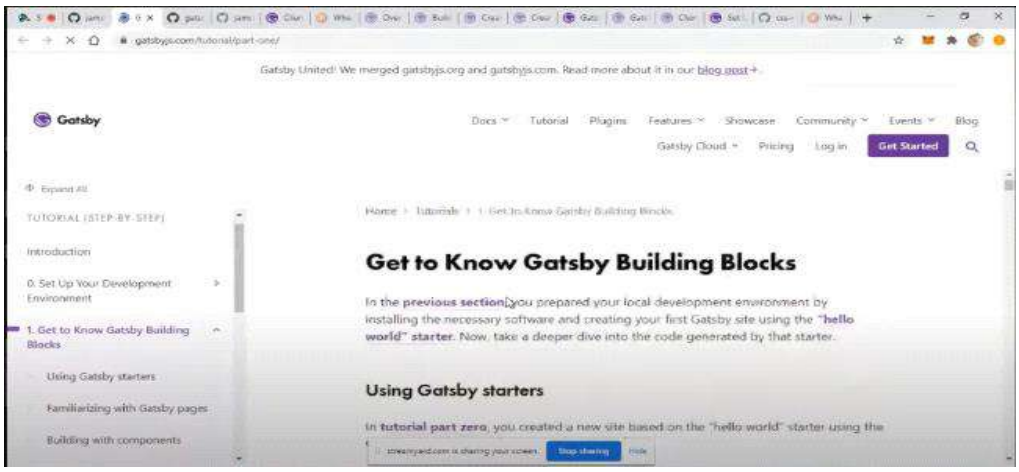
What is Gatsby ?

Gatsby is an application which works on both the client site rendering or server site generation. For example:

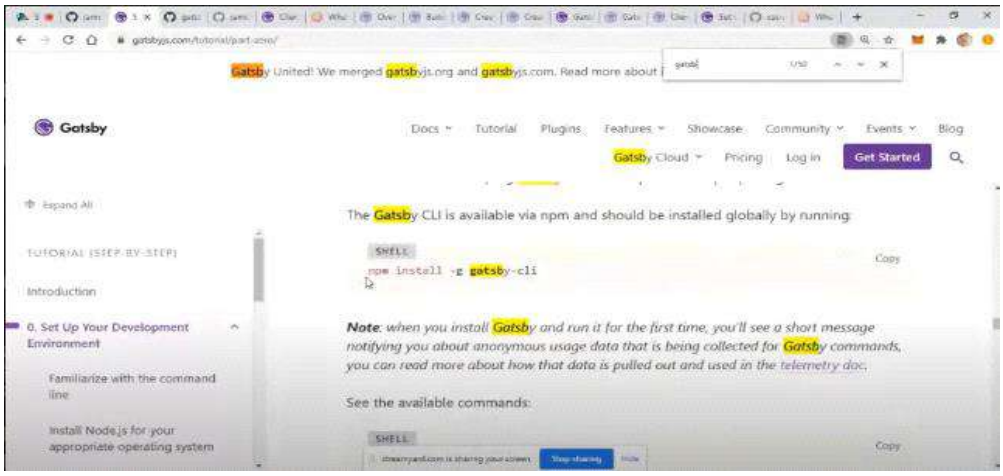
If you work on a large scale or a bulk of products you have to understand what you really want, do you really need a client site rendering or to rebuilt a page every time or not.

How to Install Gatsby?

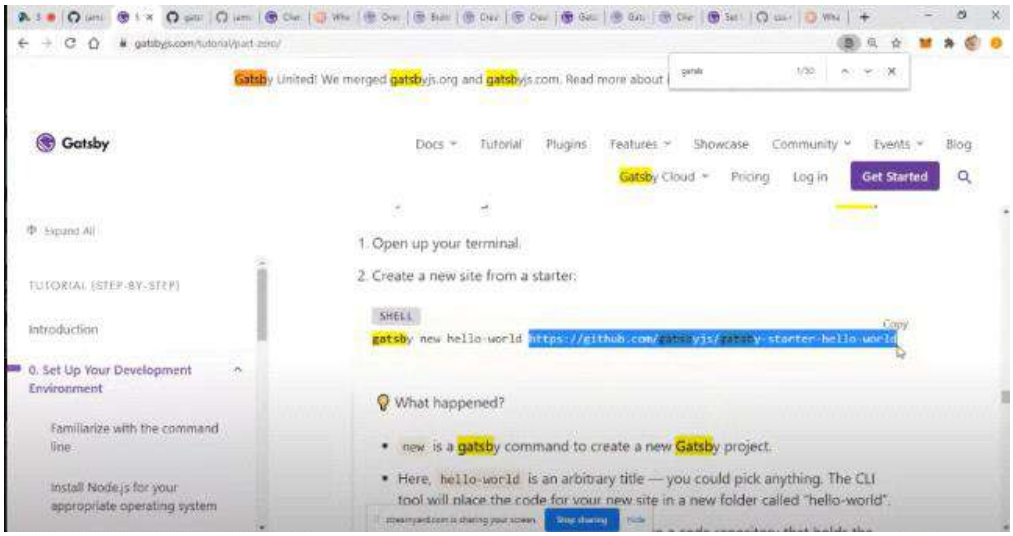
- Go to Gatsby official website



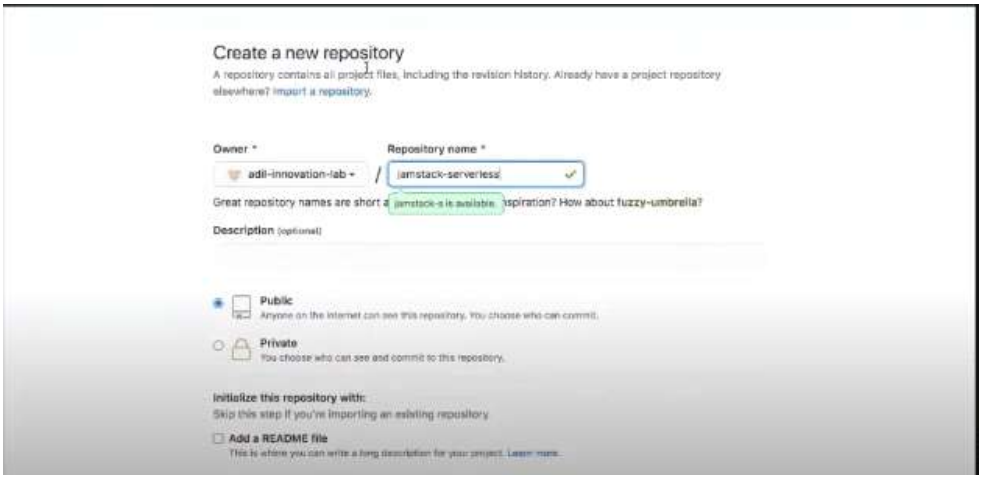
- Install "gatsby-cli"

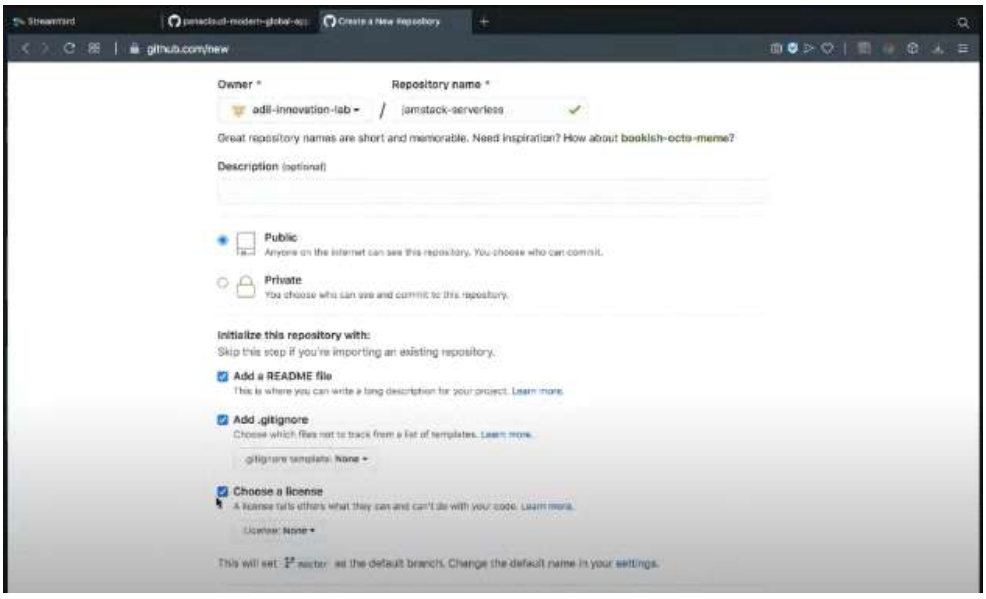


- To run the command

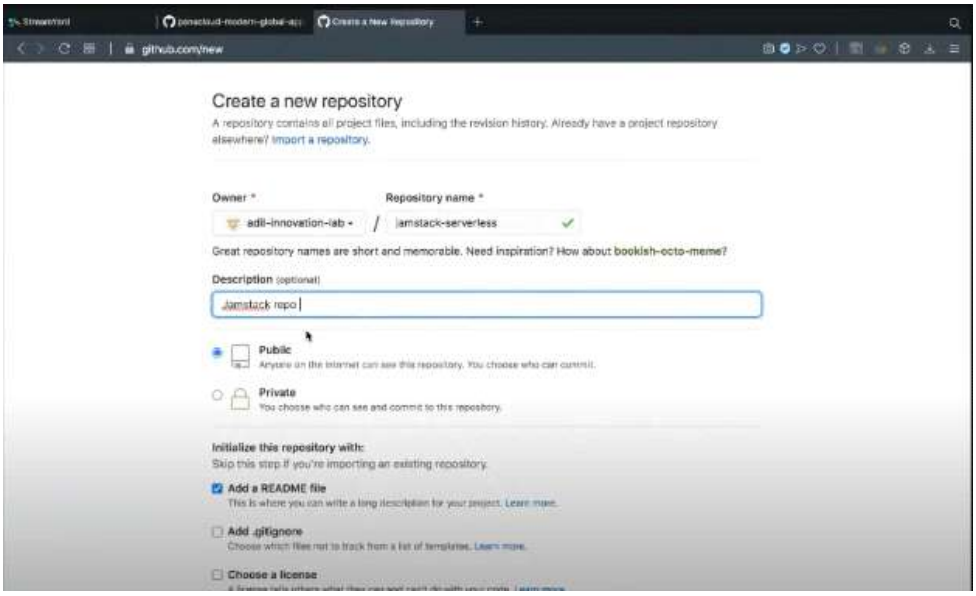


Create a new respository:



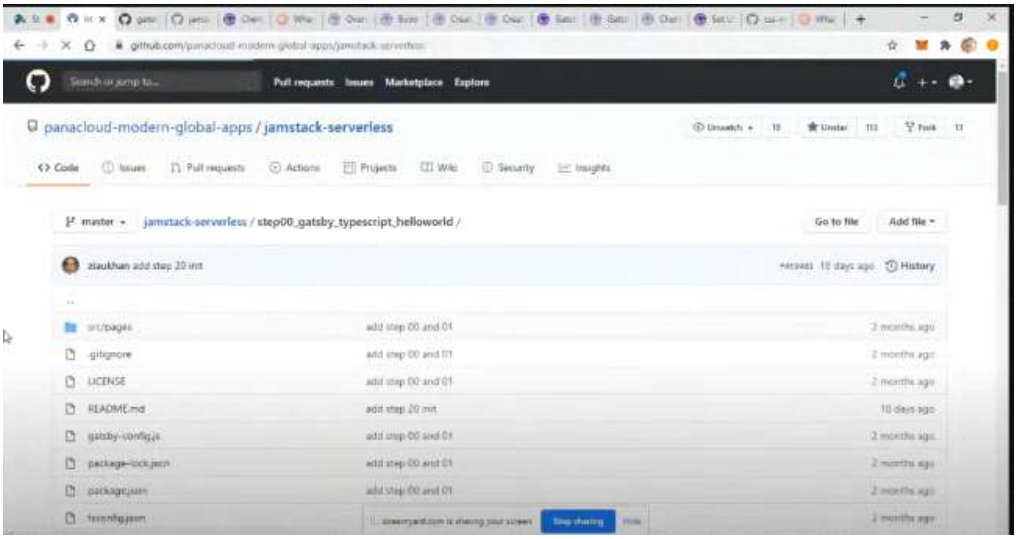


2. Add description

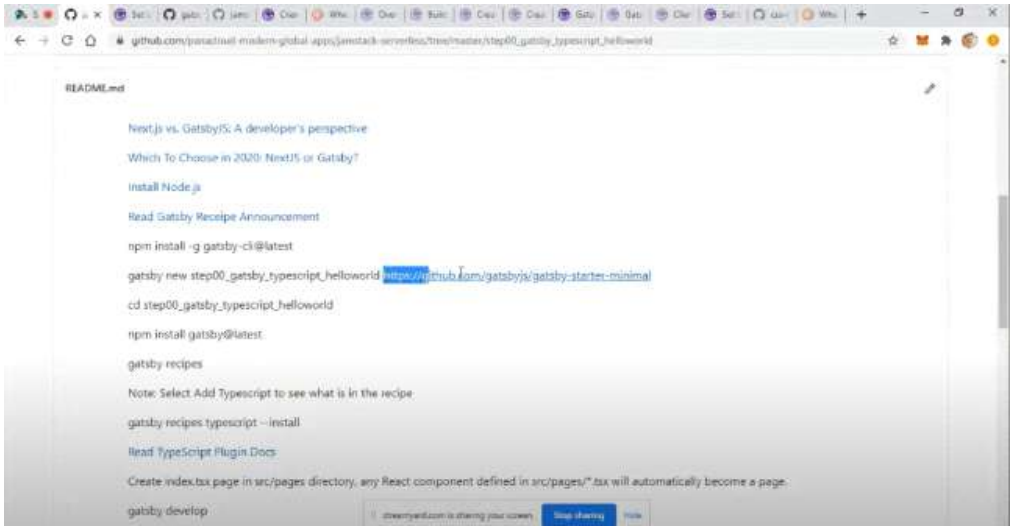


Step 00

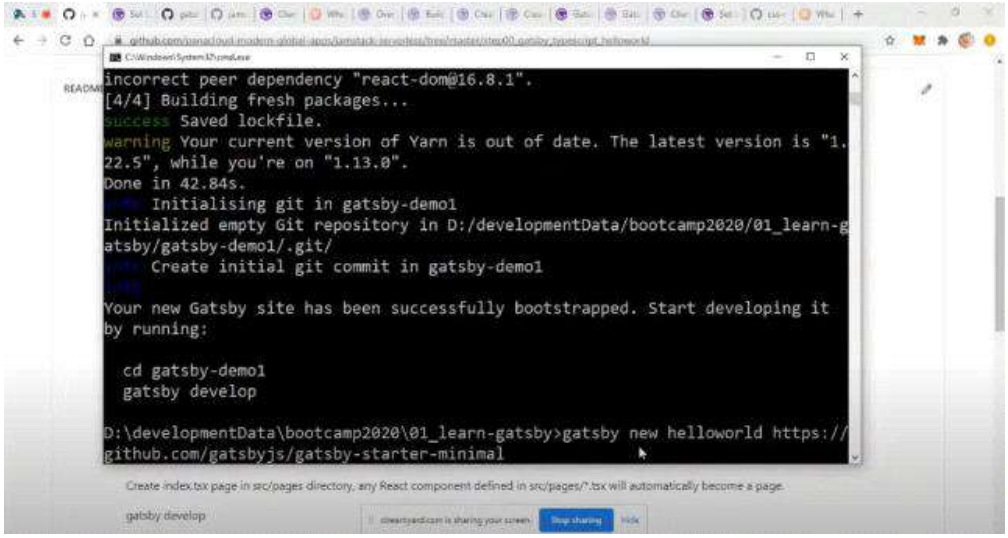
Gatsby Typescript Helloworld



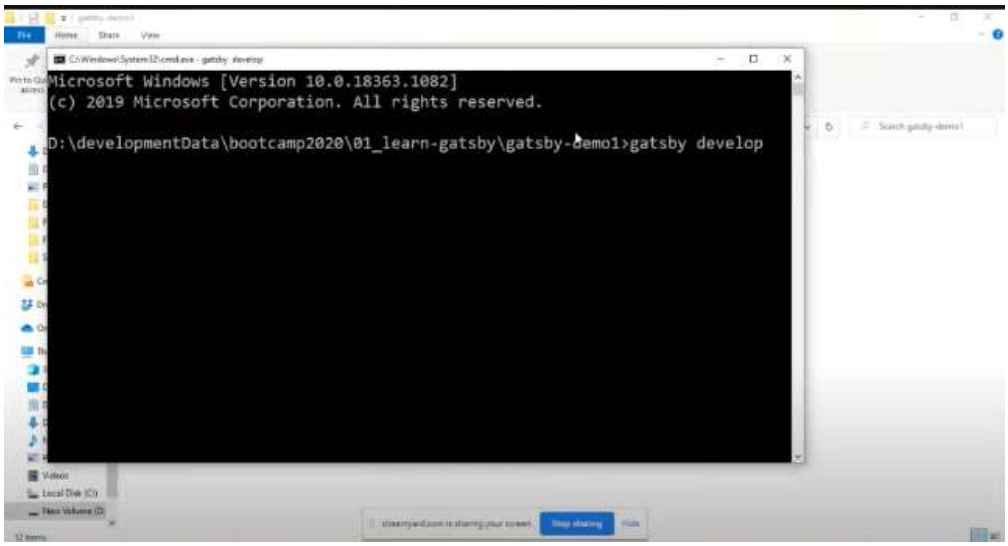
- Use Gatsby starter minimal



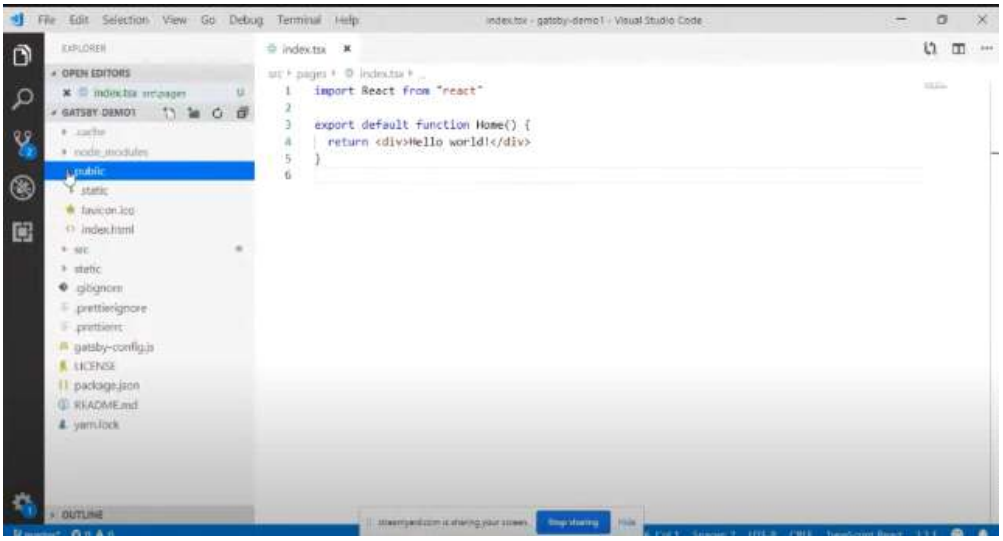
- Paste it on Gatsby developer



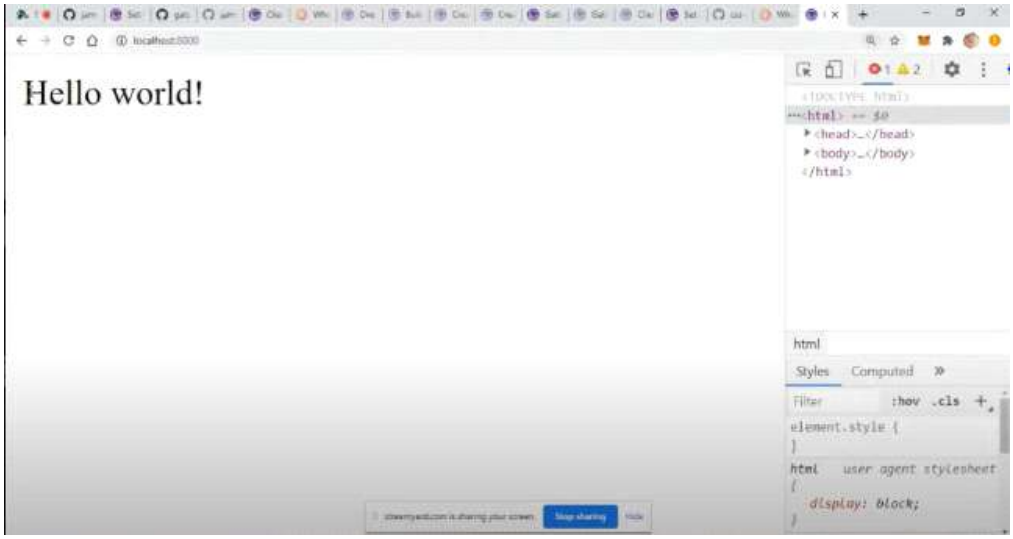
- Gatsby develop using node.js



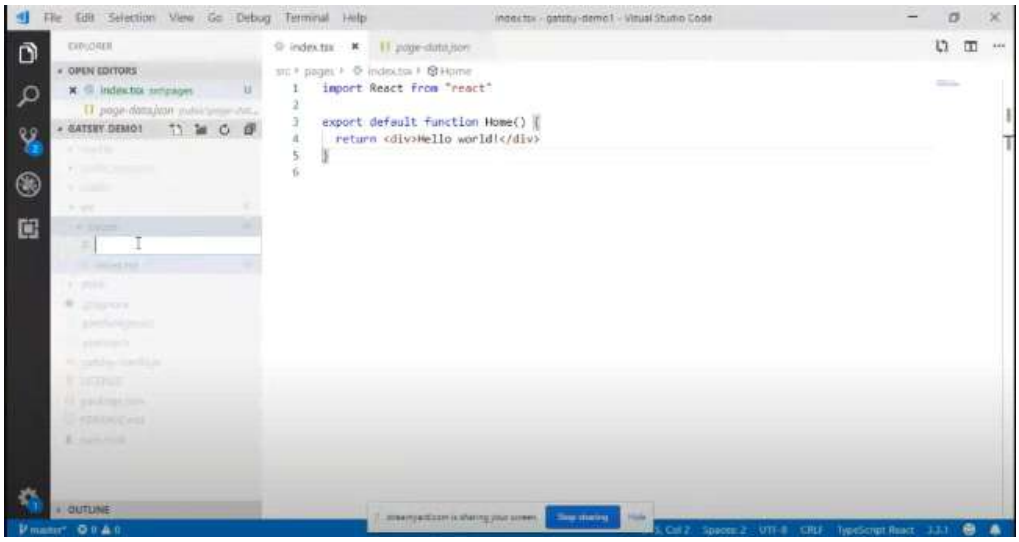
- After developing you have to check the pages name



- Project Completed



- To add new pages, add new file in pages



- Copy the code from previous page

Import React from “react”

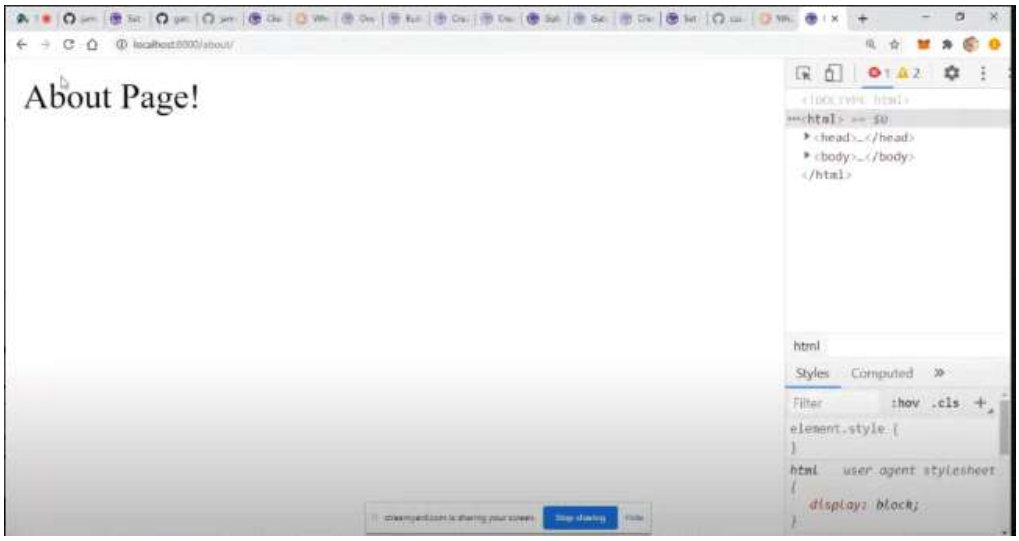
Export default function Home () {
  Return HelloWorld
}

- Paste the code in new page
    - To change the page name

Import react from "react"
export default functionn About () {
return About page!
}

Then behind this Gatsby develop will run the commands, as you save the project

- To check a new page “localhost8000/about/”



Step 1

Page Component Liniking

- To add links on page

```
Import react  from "react"
Import { link } from "gatsby";

export default function Home() {
return (
<div>
About page!
<link to="/about">About<link>
 </div>
}
```

- Add page name with the link

- To navigate with button

```
Import react  from "react"
Import { link } from "gatsby";

export default function Home() {
return (
<div>
About page!
<br/>
<link to="/about">About<link>
 </div>
}
```

  - If you want the components based on some button click you just have to create this code

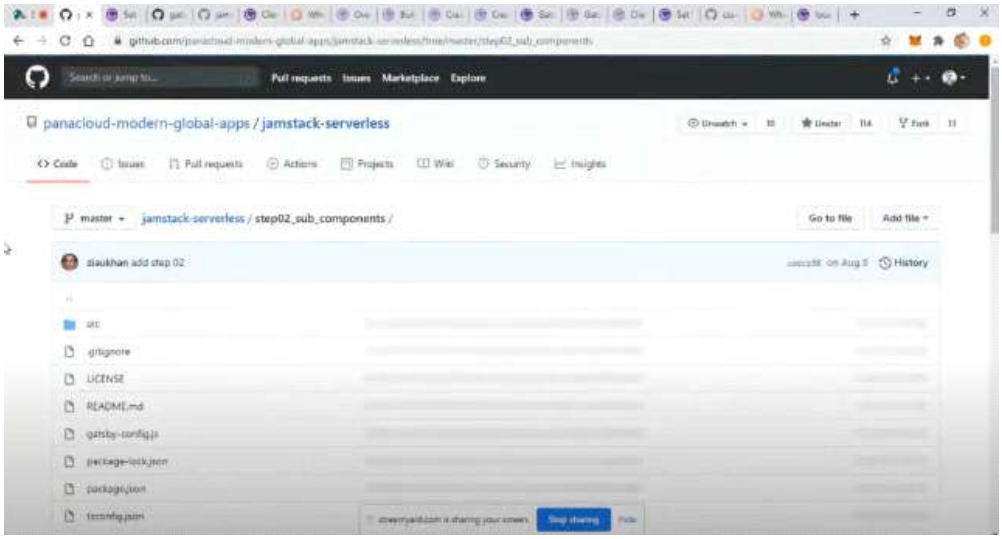
```
Import react  from "react"
Import { link } from "gatsby";

export default function Home() {
   return (
       <div>
            About page!
            <br/>
             <link to="/about">About<link>
             <button onClick= {(=>{
                 navigate ('/about')
        }}>About page with button</button>
     </div>
```

Step 2

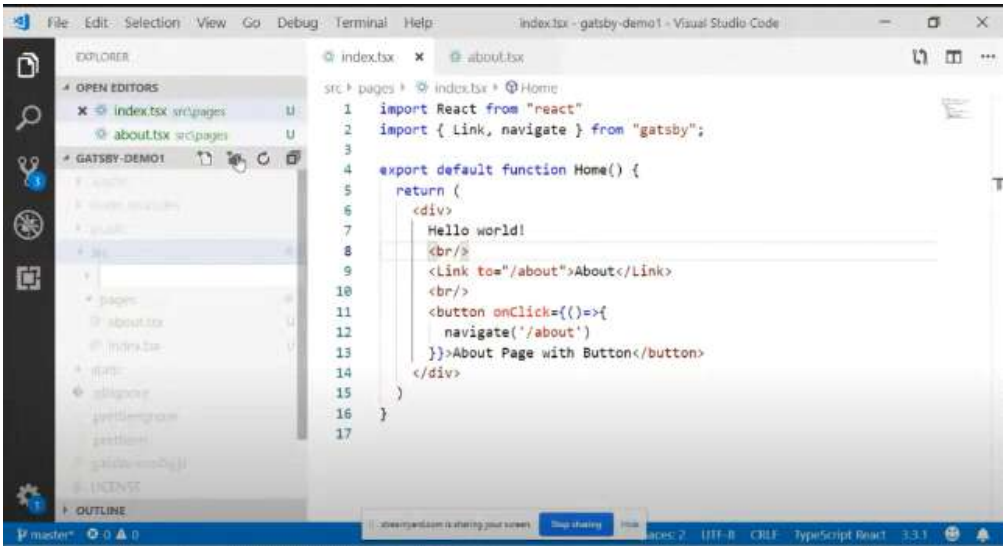
Sub Components

How To Built a Sub Component ?

You have to makes some changes in previous and sub components in your project that play a vital role in your page. For Example: You want to add header or any component you just use sub components.



- Create a folder name “component”



- Add a component name
- Copy the code from previous page
  - Import react from "react"
  - export default functionn About () {
  - return About page!
  - }

- Paste it on the new component

```
Import react  from "react"
export default functionn About () {
return <div>About page! </div>
}
```

- Make some changes in the code

```
    import react from "react"
   export default function Header ({title}) {
         return (
          <div>
              <h1>{title}</h1>
         </div>
  )
 }
```

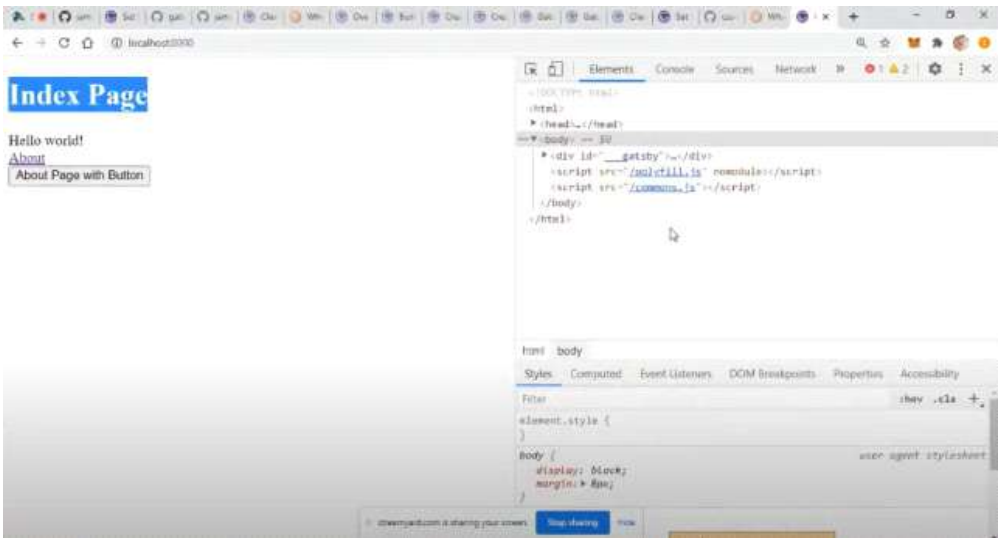
- Go to index page and add header inside the div

```
Import react  from "react"
Import { link } from "gatsby";

export default function Home() {
   return (
       <div>
           <header title= "index pae"/>
            About page!
            <br/>
             <link to="/about">About<link>
             <button onClick= {(=>{
                 navigate ('/about')
       }}>About page with button</button>
    </div>
```

- Same with about page

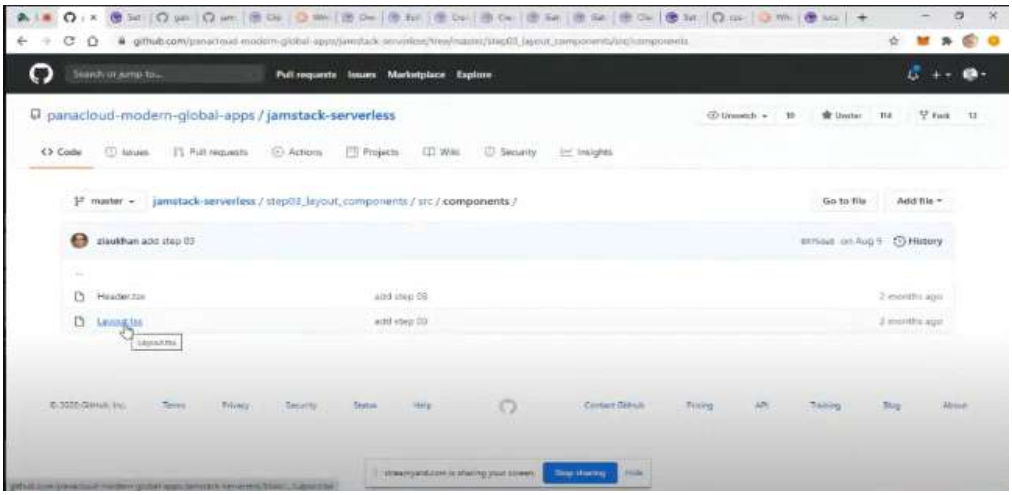
```
Impoort react  from "react"
export default functionn About () {
return
 <div>
      <Header title -"About pae title"/>
      About page!
 </div>
}
```

- Go on “localhost8000”

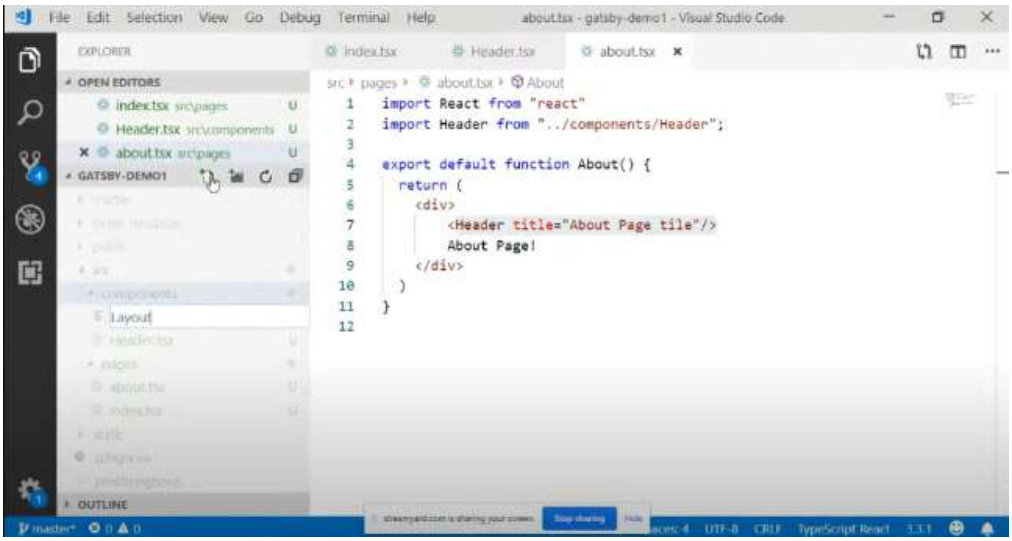


Step 3

Layout Components

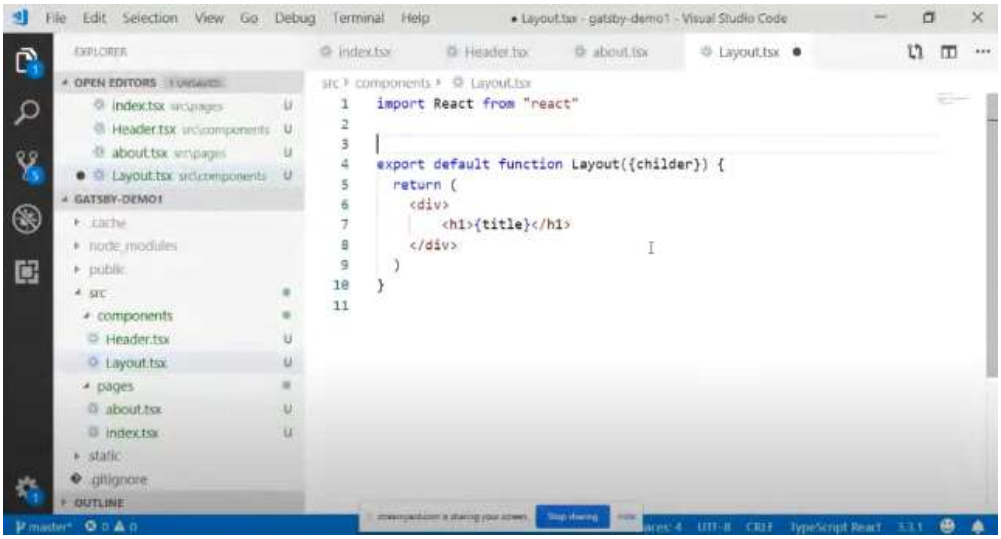


- Create a new component named as “layout”



- Copy the header’s code in layout component
  - Import react from "react"
  - Import { link } from "gatsby";
  -
  - export default function Home() {
  - return (
  -
  - <header title= "index pae"/>
  - About page!
  -
  - <link to="/about">About<link>
  - <button onClick= {(=>{
  - navigate ('/about')
  - }}>About page with button</button>
  -
  -

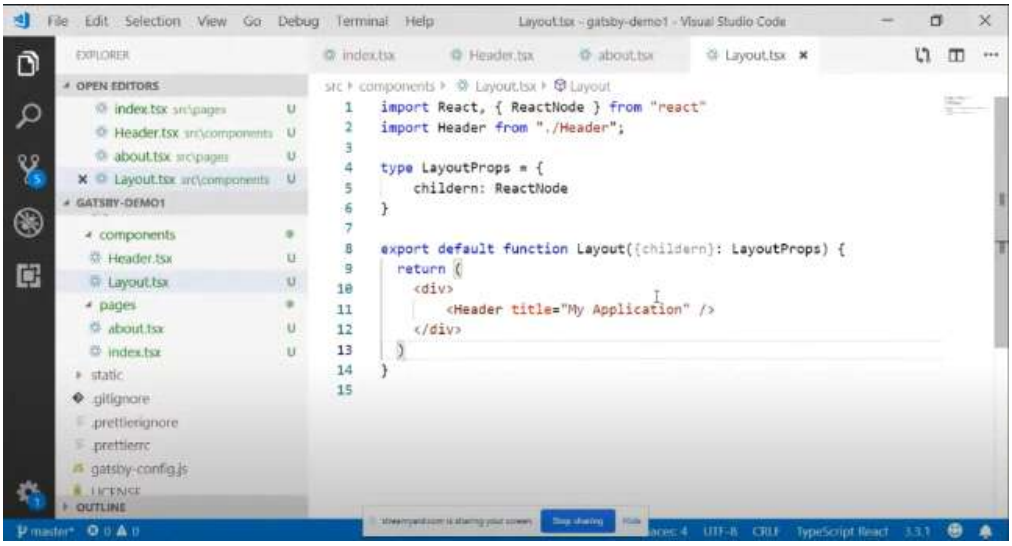
- Add the layout name



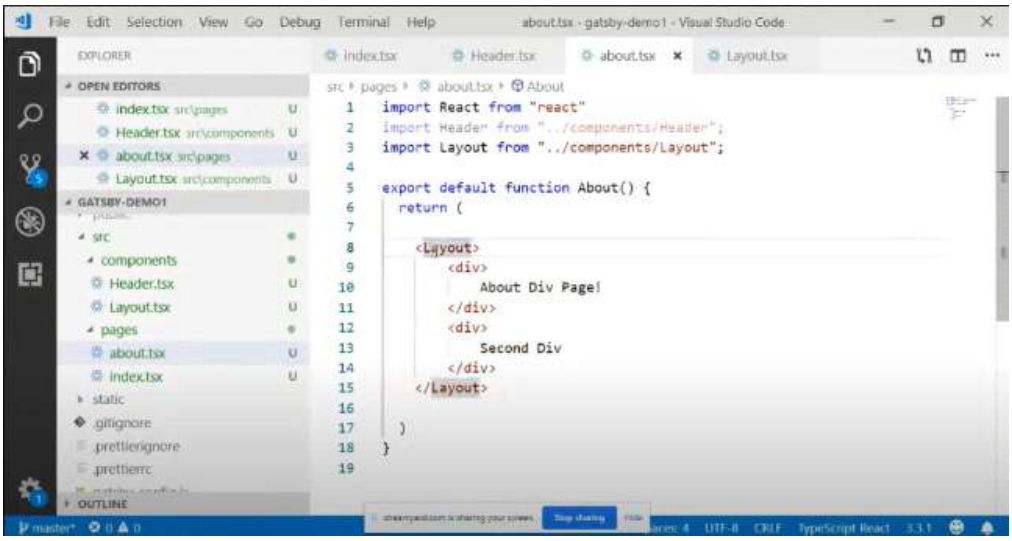
- Add layout props

```
Import react  from "react"

type layoutprops {
      children : ReactNode
}
export default functionn layout ( {children} . layout props) {
return (
 <div>
    <h1>(title)</h1>
 </div>
}
```

- Add header inside the div



- To add div component in the about page

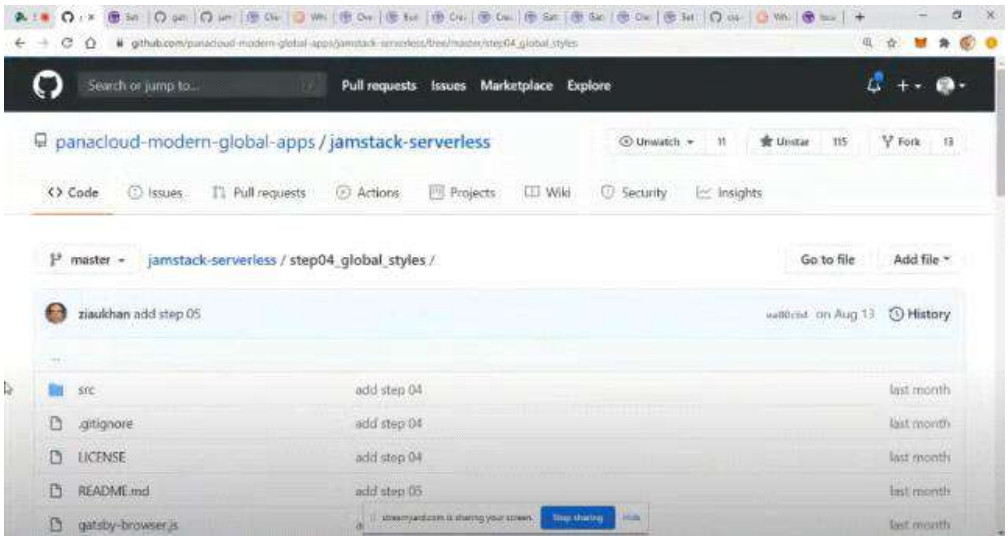


- Add all your changes in index page

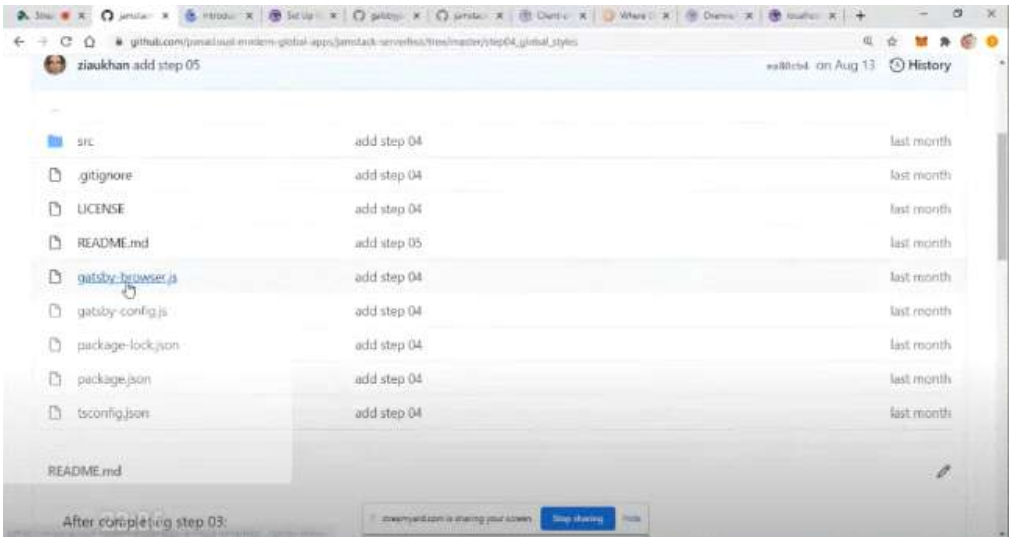
```
Import react  from "react"
Import { link } from "gatsby";

export default function Home() {
   return (
      <layout headerTitle="index page header">
       <div>
           <header title= "index pae"/>
            About page!
            <br/>
             <link to="/about">About<link>
             <button onClick= {(=>{
                 navigate ('/about')
        }}>About page with button</button>
     </div>
```

Step 4

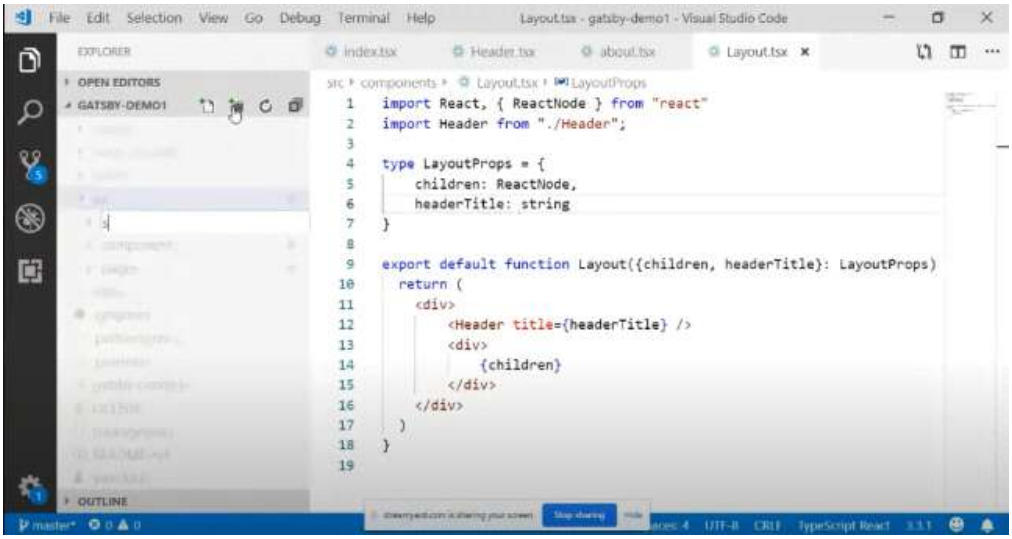
Global Styles



- Go to Gatsby-browser.js



- Create a new folder in src

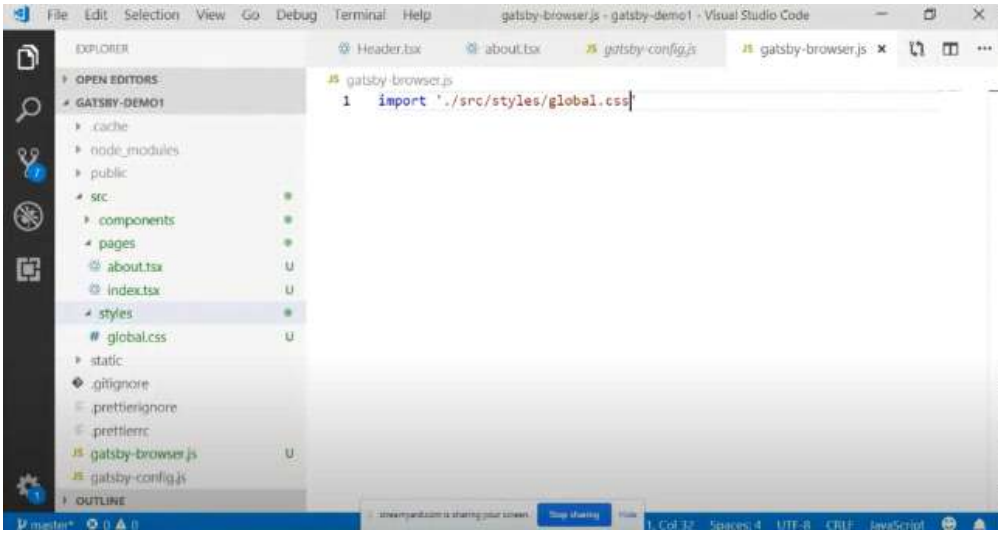


- Create a new folder in styles

- Create global.css in styles

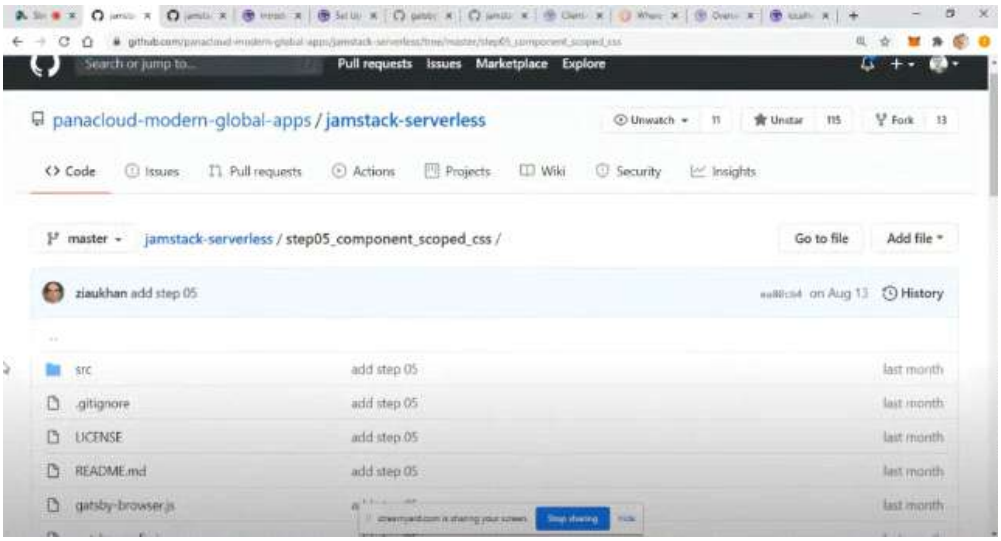
```
Body {
    Background-color: (color)
}
```

- Create a new folder named as “Gatsby-browser.js”
- Import CSS in Gatsby-browser.js

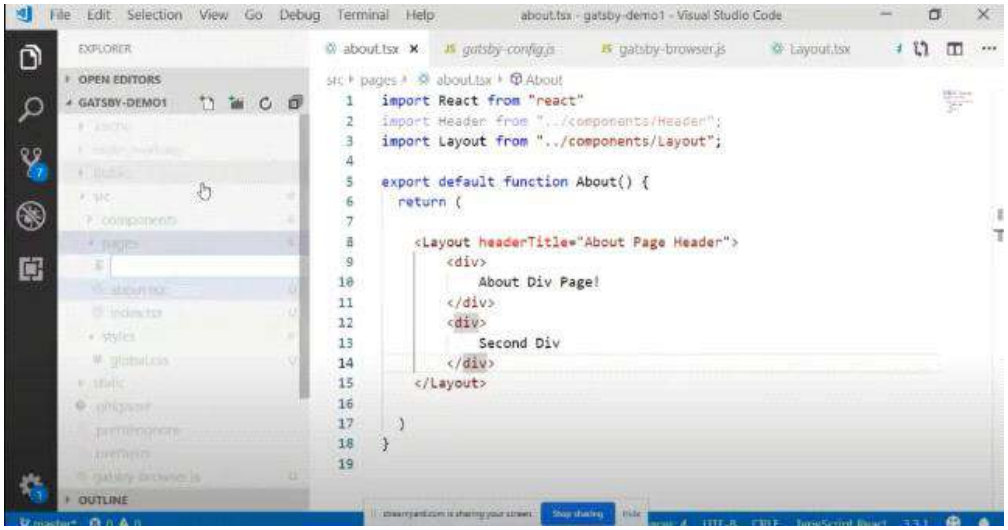


Step 5

Component Scoped CSS



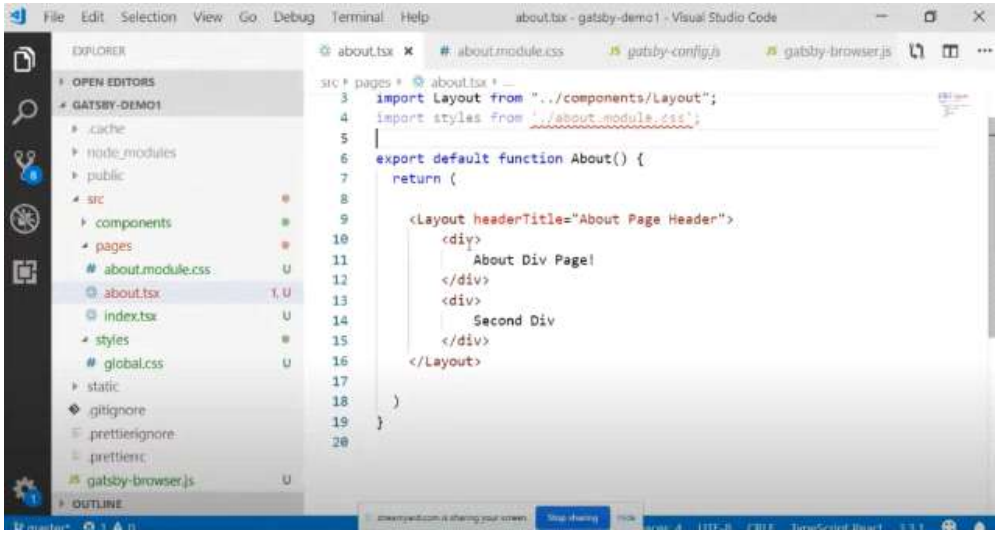
- Add new file in pages



- Create a CSS

```
.myTitle {
    Background-color: (Color)
}
```

- Import global CSS on about page



- Add classname

Session-33

Step No. 6

Programmatically Create Pages

Where Does Logic Goes In Jamstack?

Option 1: Write it into an HTML ourselves
Option 2: Write structured data and do logic at build time
Option 3: Do logic at edge
Option 4: Do logic at run time

Difference between Build time & Run time

Build Time

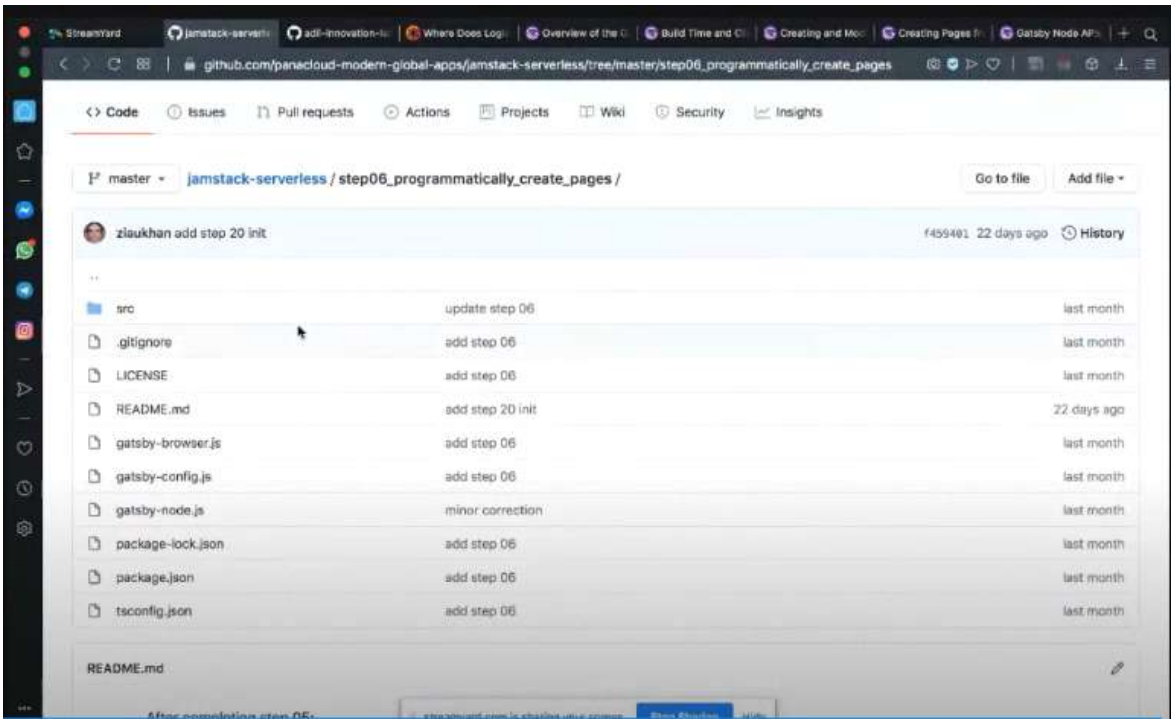
A product begins its life in "build time". This is where the product is created. The WiX toolset usually plays a significant role during the latter phases of "build time" when the product is packaged, signed and released.

The bulk of the WiX toolset is used during build time.

Run Time

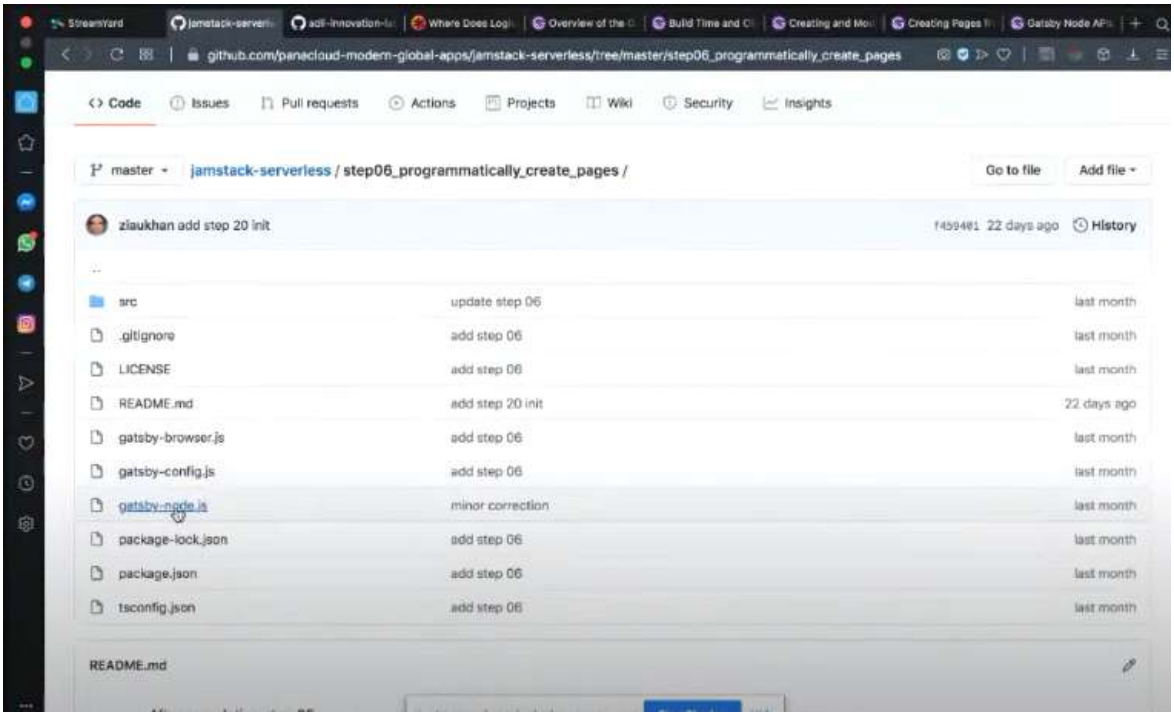
Hopefully, a product spends the bulk of its life doing what it was designed to do in "run time". Very little of the WiX toolset participates in the run time of a product although some components, such as DTF, may be useful.

The cycle repeats for updates to the product.



Steps of creating pages:

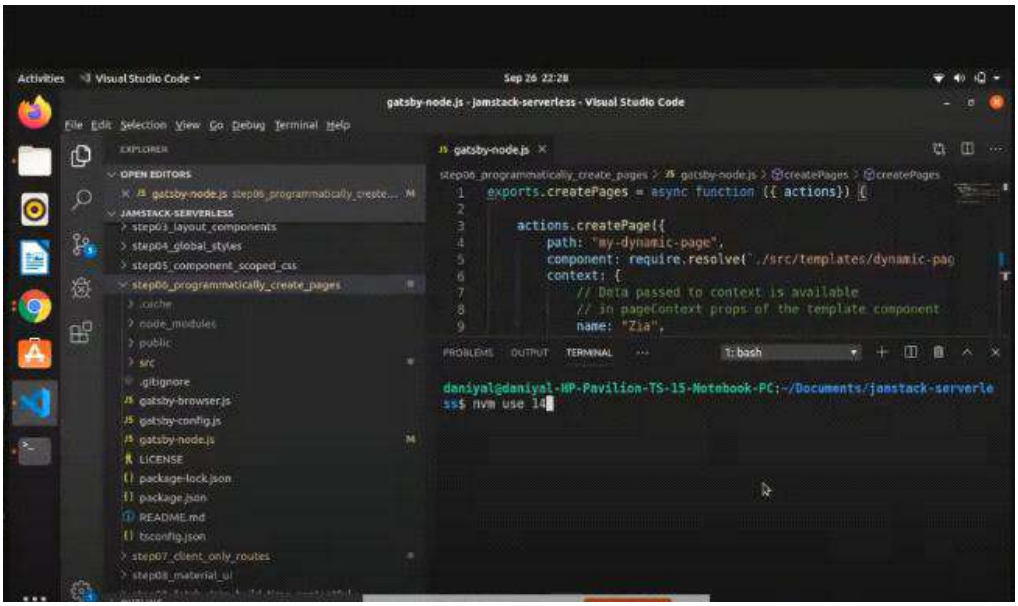
- Go to gatsby-node.js



Action: create page

```
      Path: "my dynamic page"
      Component: require.resolve
      Context: {
      // Data passed to context is available
     // in pageContext props of the template component
     name."admin".
  },
});
console.log("End of the gatsby node file");
}
```

5. Add sudo gatsby develop to gatsby-node.js



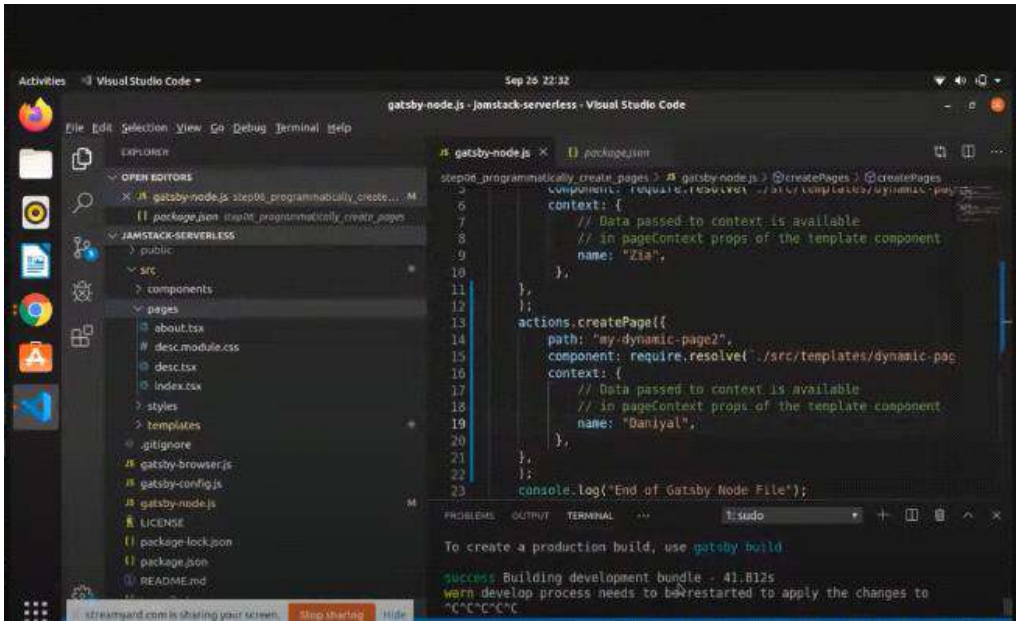
<u>For multi-pages :</u>

You just have to copy the same process as you use before for creating a page.

```
     Action: create page
     Path: "my dynamic page 2"
     Component: require.resolve
     Context: {
      // Data passed to context is available
     // in pageContext props of the template component
     name."Howdy".
  },
```

```
});
console.log("End of the gatsby node file");
}
```

5. Add this code to gatsby-node.js

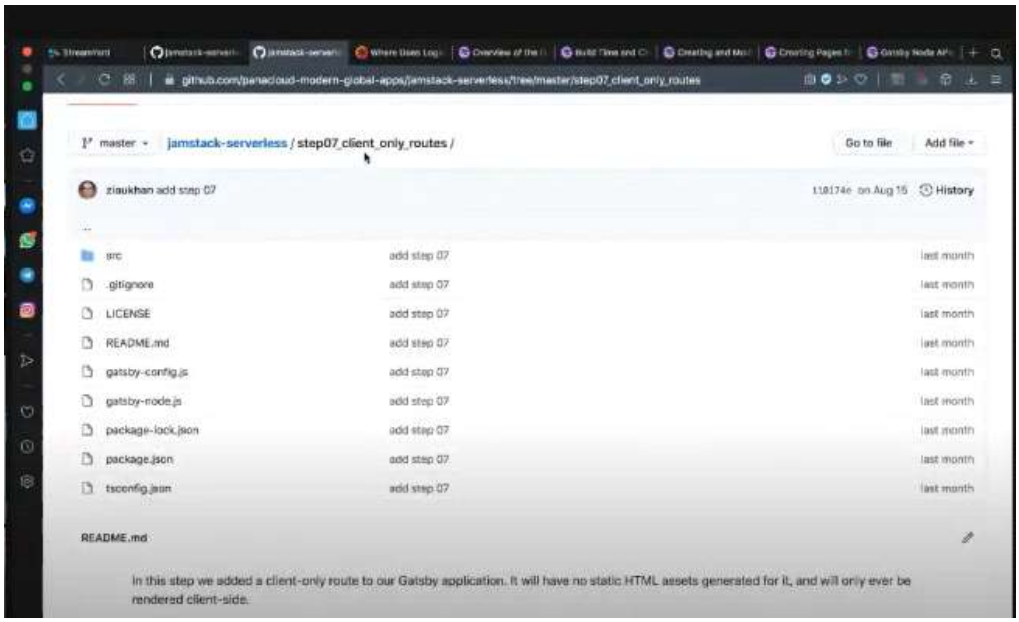


For E-commerce websites :

```
api = "sofa"
 Action: create a page
 Path: "URL"
 Component: require.resolve
 Context: {
        // Data passed to context is available
       // in pageContext props of the template component
       name."Data".
  },
});
console.log("End of the gatsby node file");
}
```

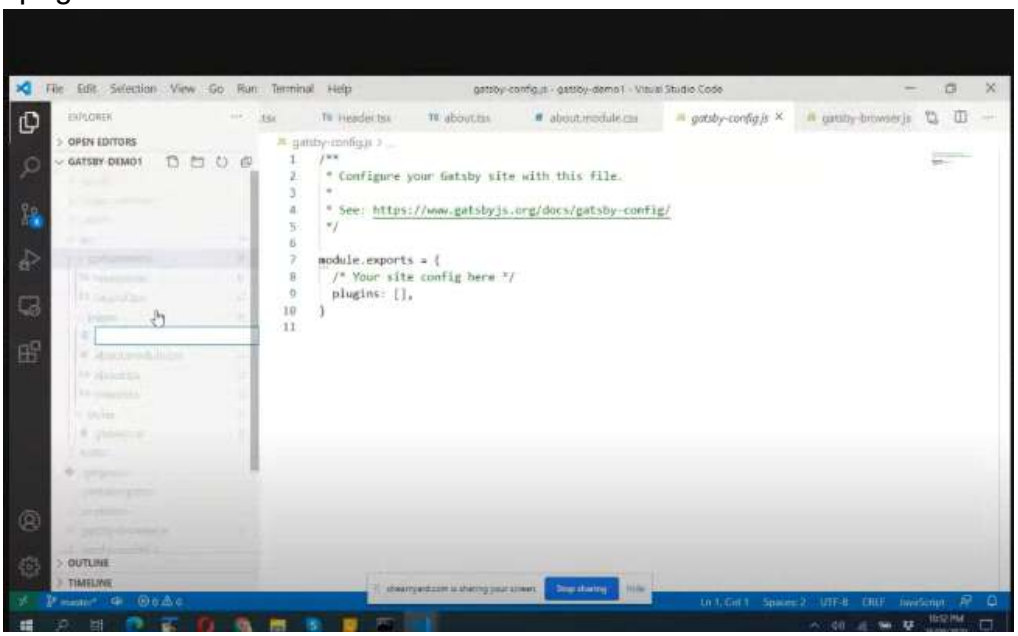
Step No. 7

Client Only Route

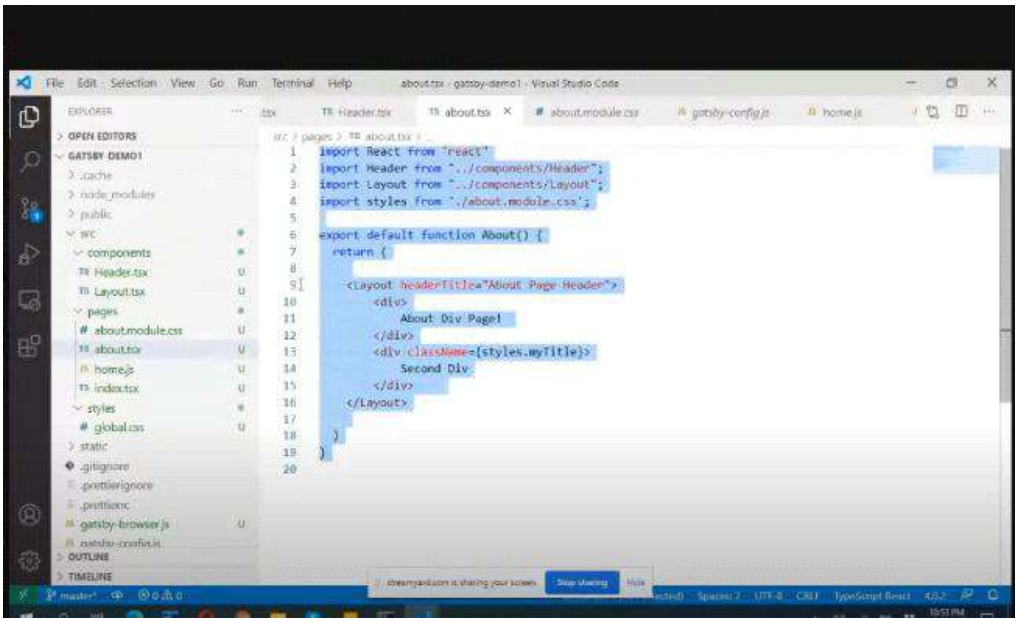


Steps of Clients Only Route :

1. Create a page

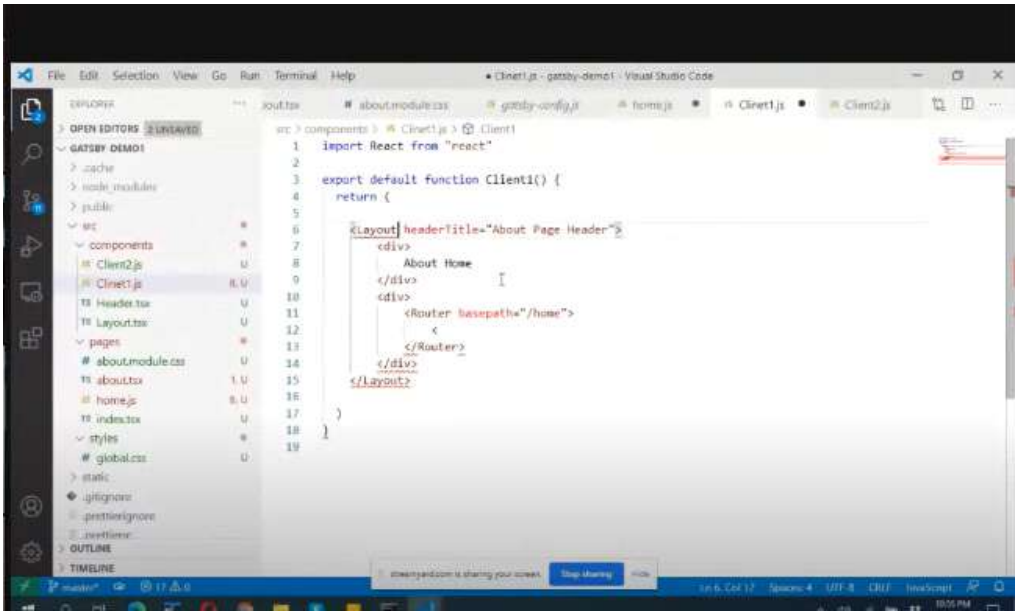


2. Copy the code that you build in previous pages



3. Paste it on a new page
4. Replace the name with a new one
5. Make few changes:
   . After page name

   <router basepath="/home">

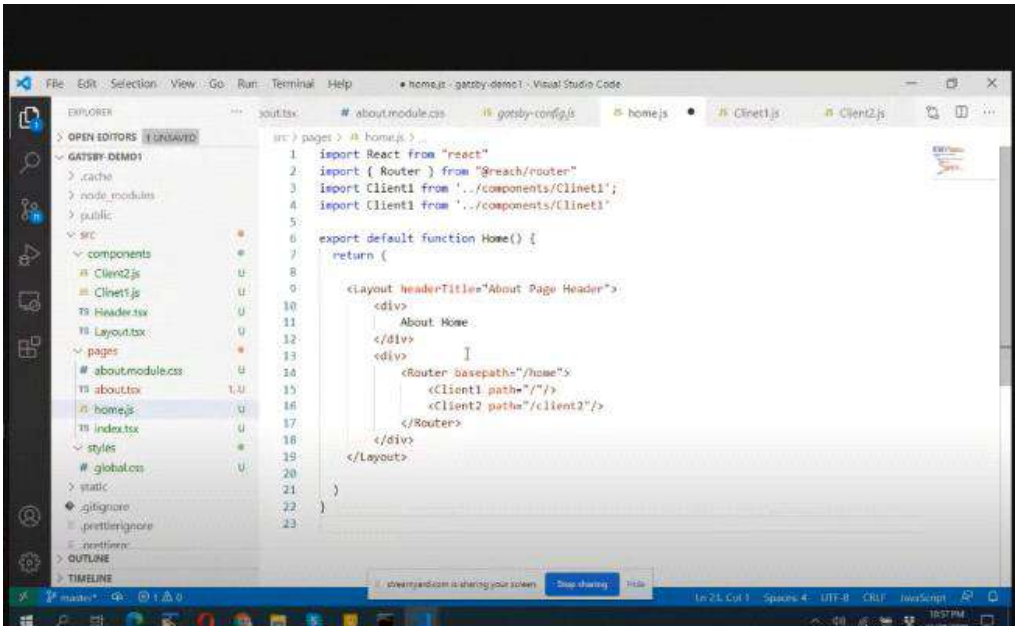


6. Make new components
7. Make changes in new components with the same code
   . Skip the layout
   . Change the page name with a new page
8. Copy the code of the component

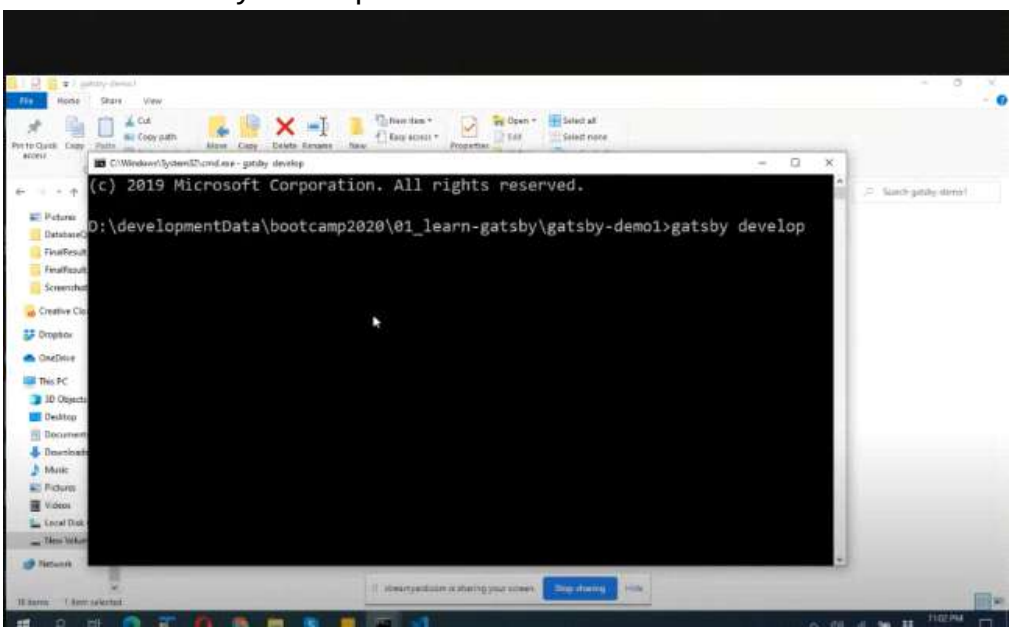
9. Use this code for other components just by changing the components name in code
10. Now add the client in your page code like that:

```
<div>
 <div>
        <router basepath="/home">
            <client path="/"/>
        <router>
```

11. Add component name on the top before layout
import client from ' ../components/client'

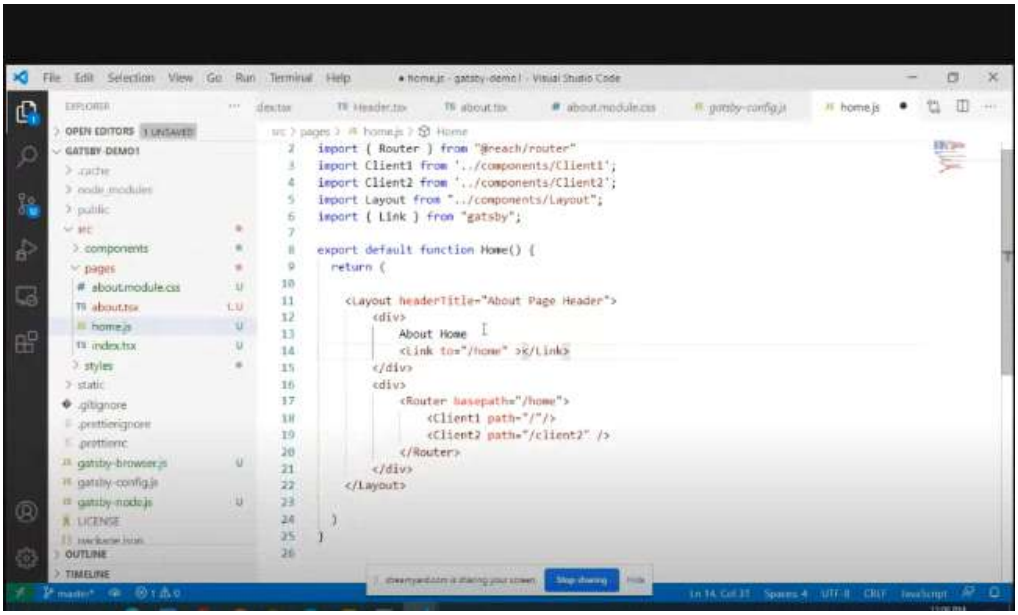


12. Add this code in Gatsby developer



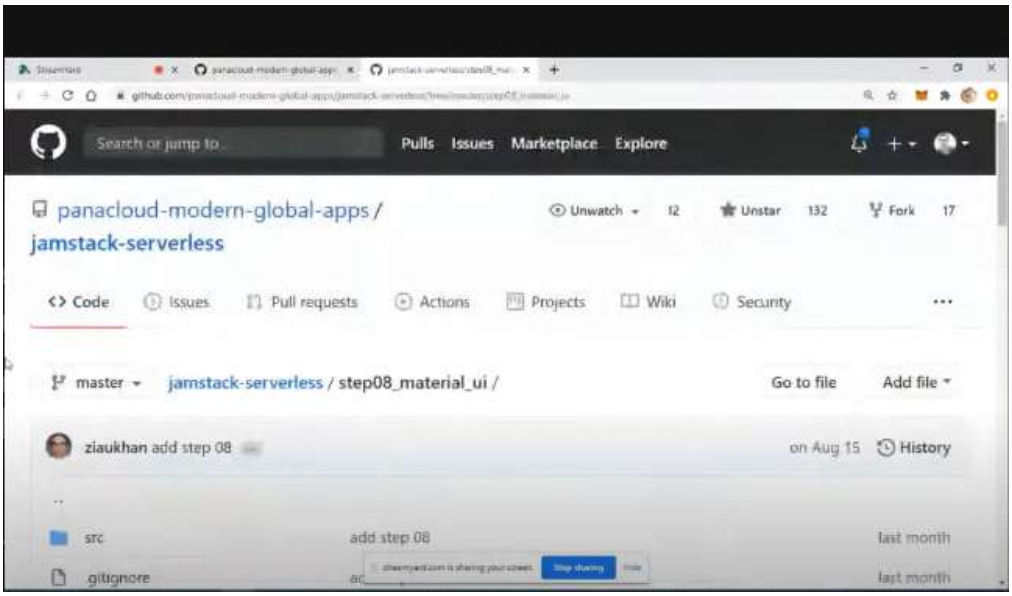
13. Add gatsby-node.js
14. Add create page function provided by gatsby
14. Add component function
16. Add pagepath
17. Go to localhoost8000/home and check your changings
18. To add the link just go home page and after the home code, you just have to add the lick of components pages.

like: <link to"/home" >client</link>



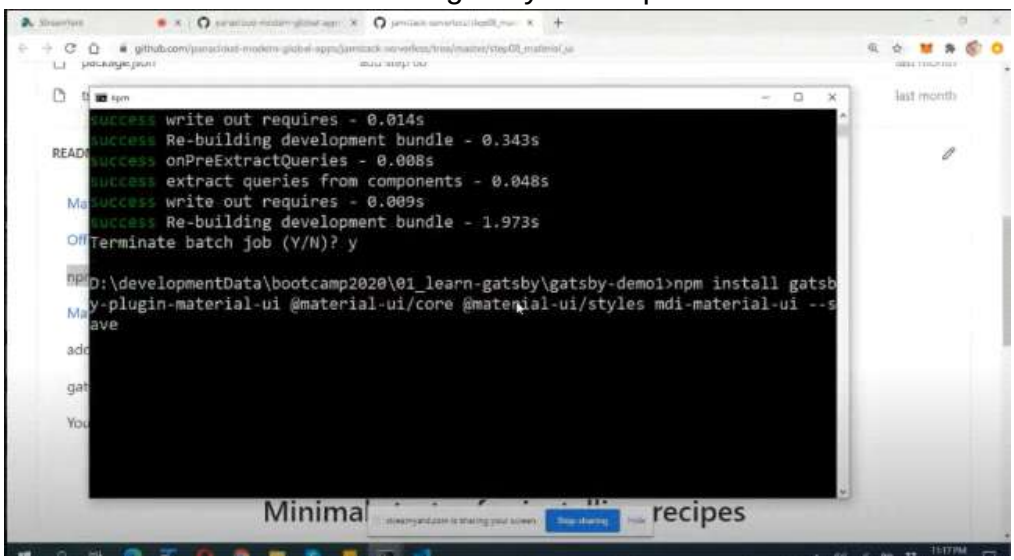
Step No. 8

Material UI

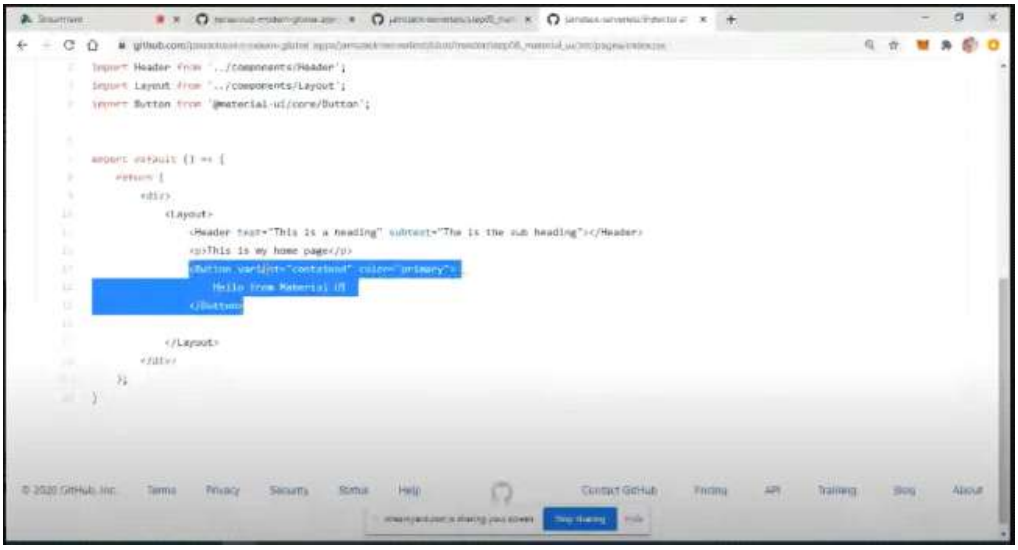


Steps of Material UI :

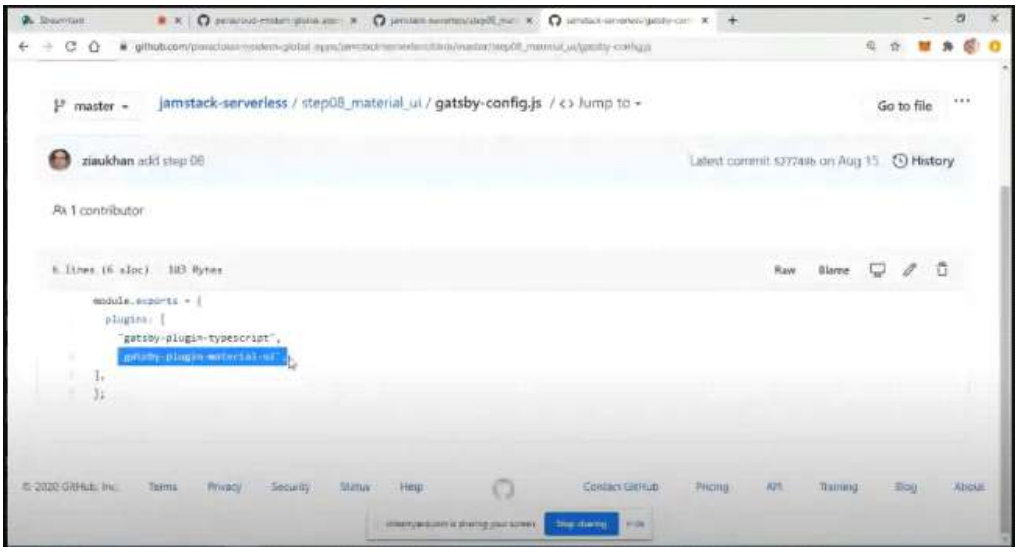
- Just install the material UI module in gatsby developer



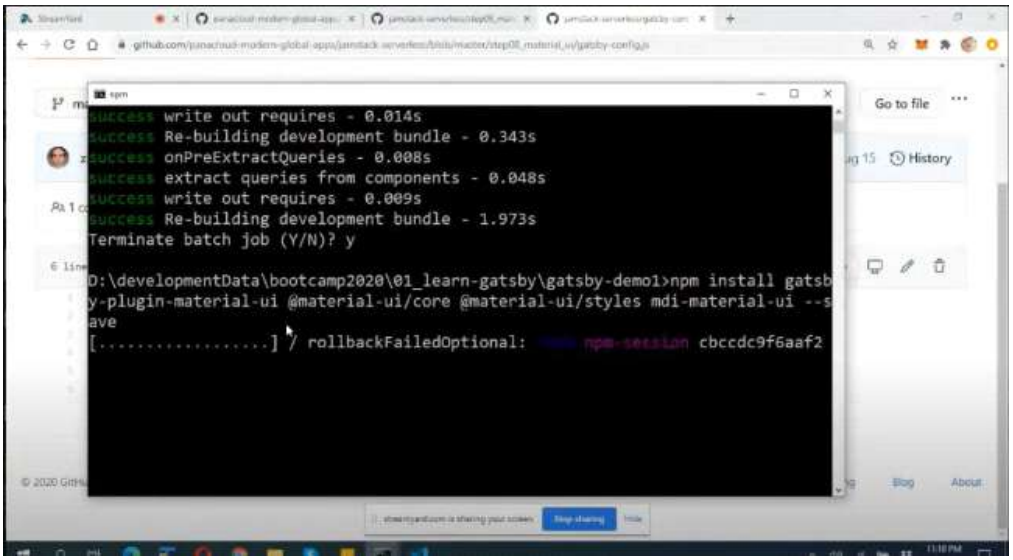
- You only saw a material UI button in components



- Go to gatsby-config.js



4. Add material UI component from there to gatsby developer
5. Install the plugin



Session-34

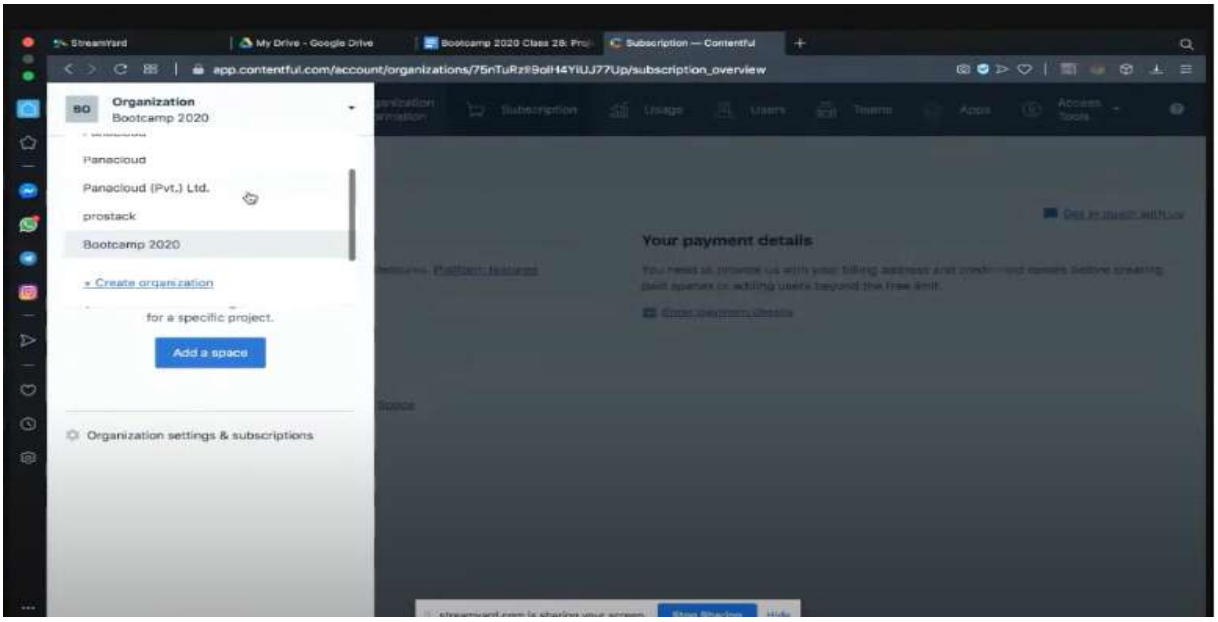
Step 9

Fetch Data Build Time Contentful

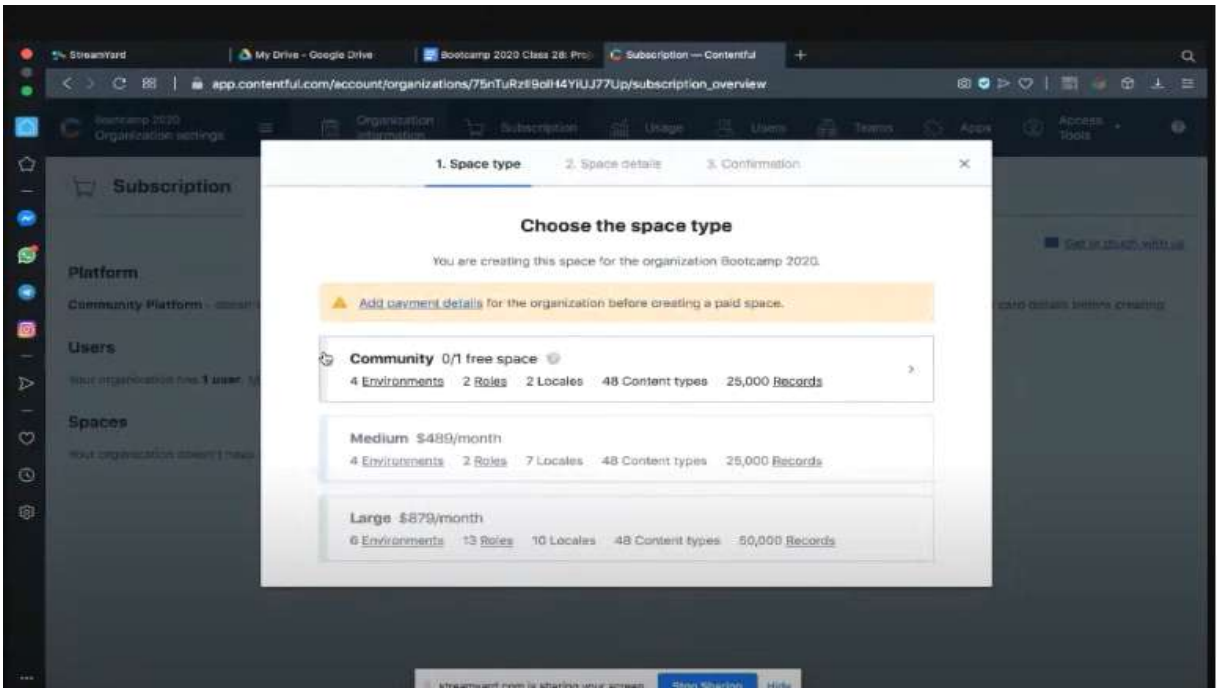
How to fetch data?

Following are the some important points to fetch data:

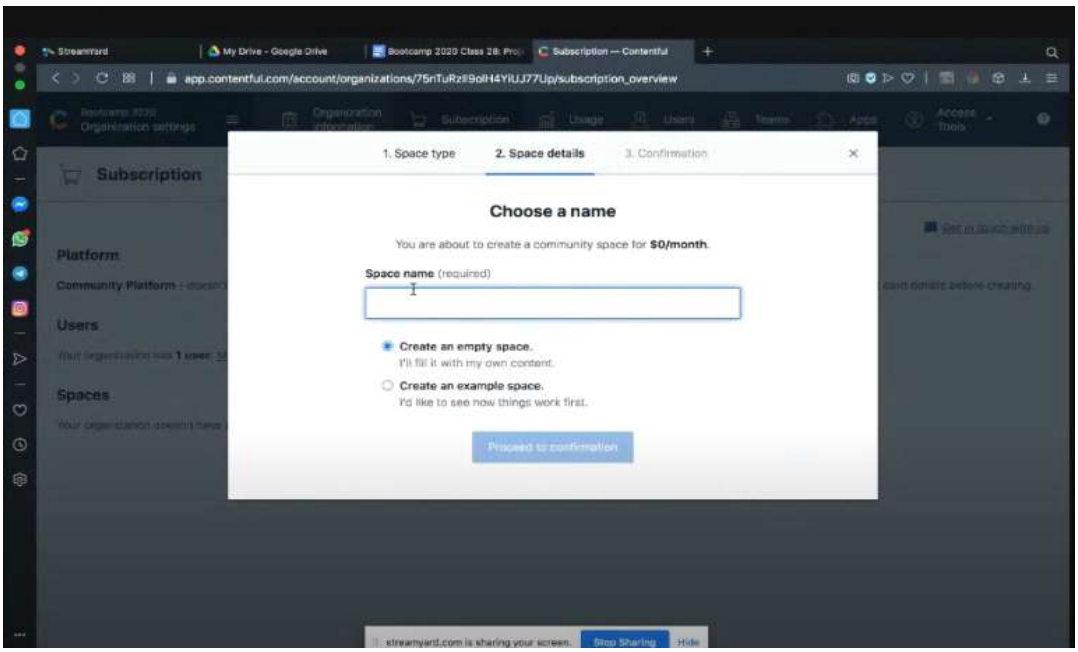
1. Create an organisation



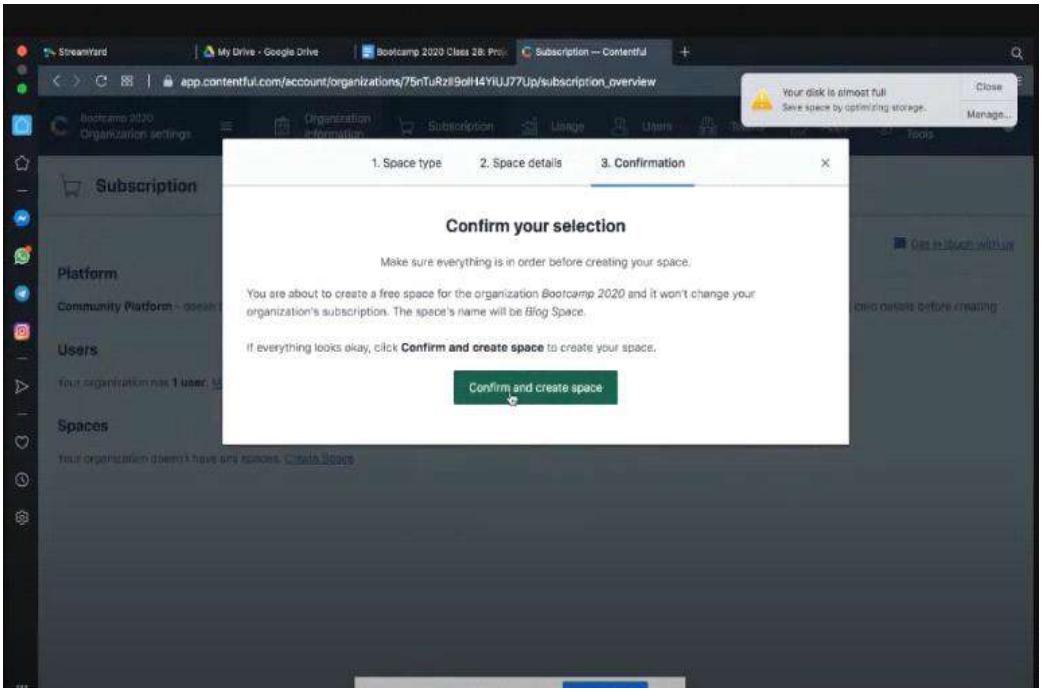
2. Add a space (community)



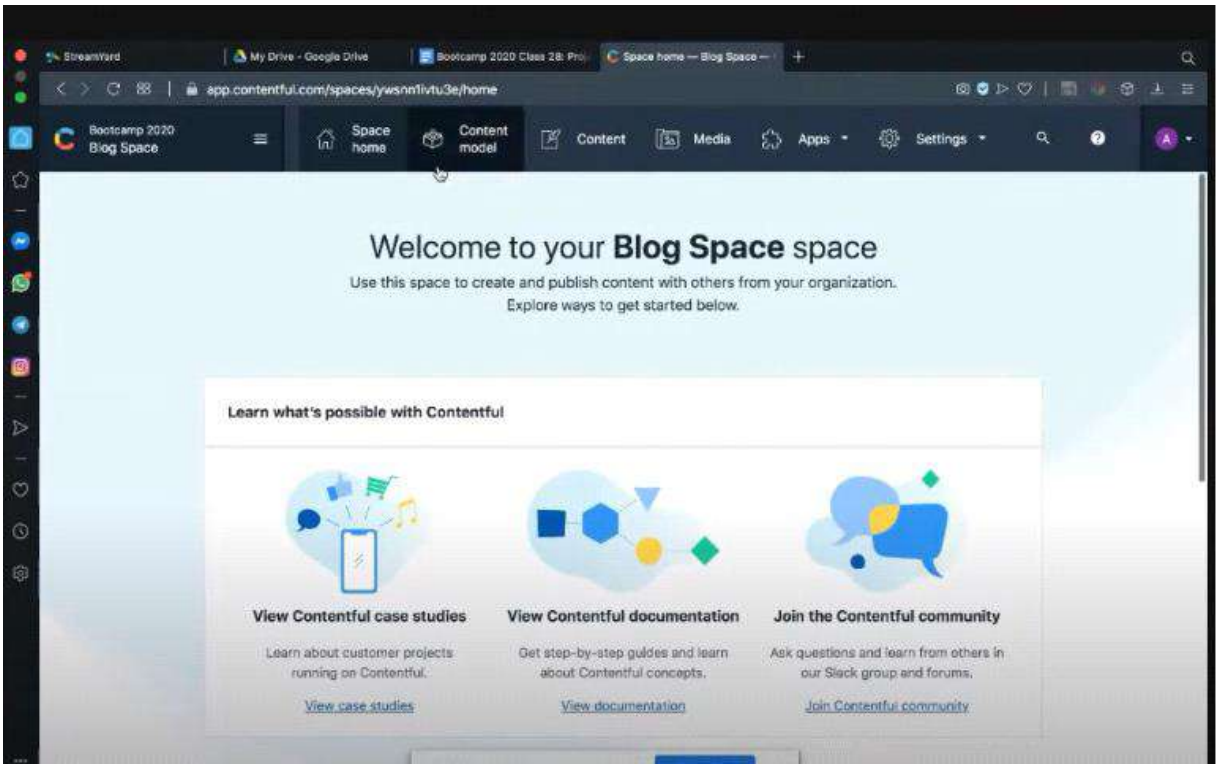
3. Add a space name



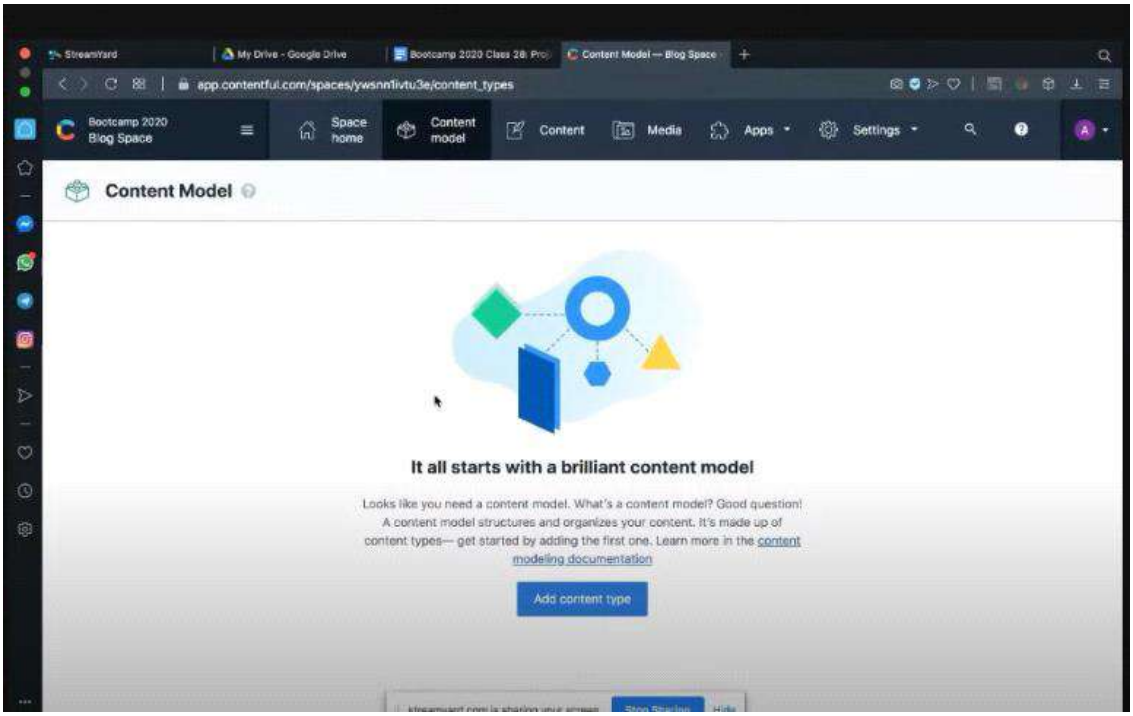
4. Confirm your selection



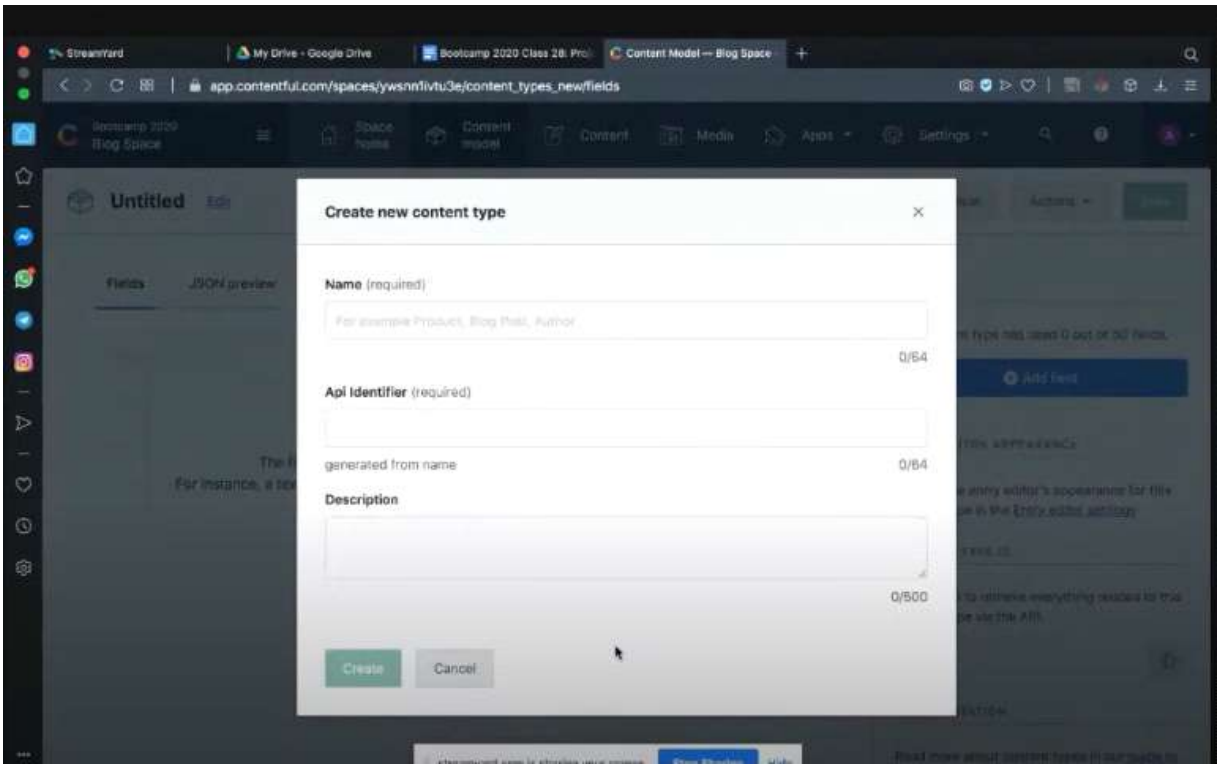
5. You will go to your space



6. Go to the content model

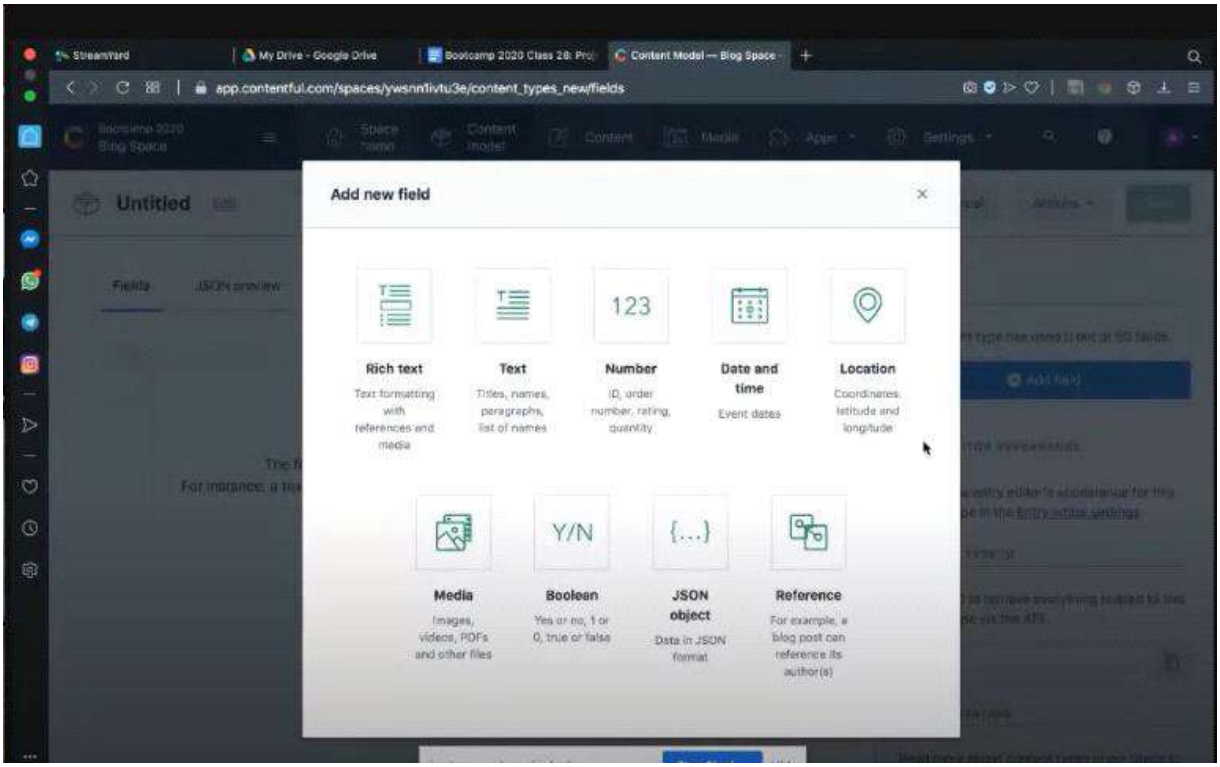


7. Add content type



8. After the creation of content type

9. Go to “Add a new field”

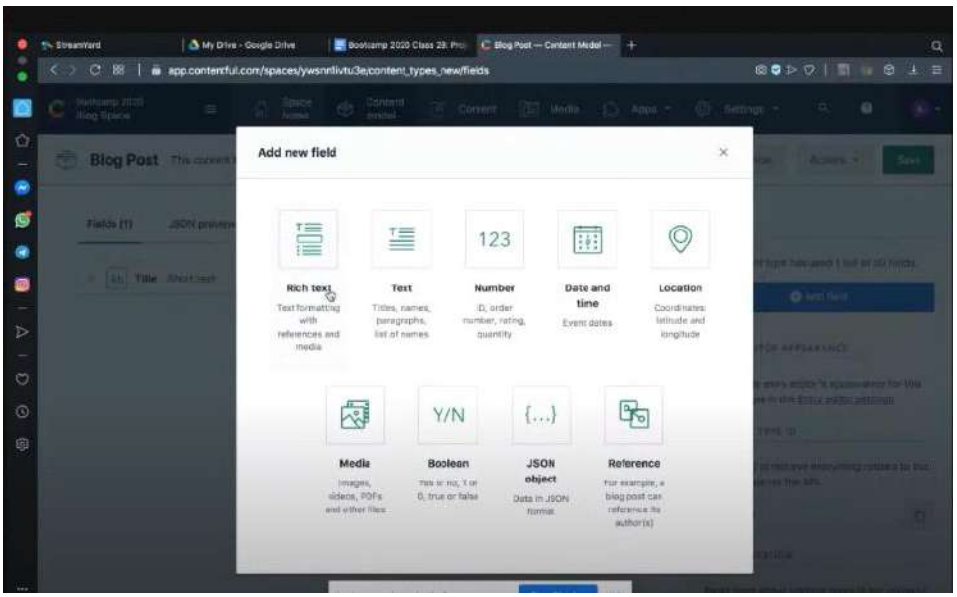


10.Go to text
11. Create a text

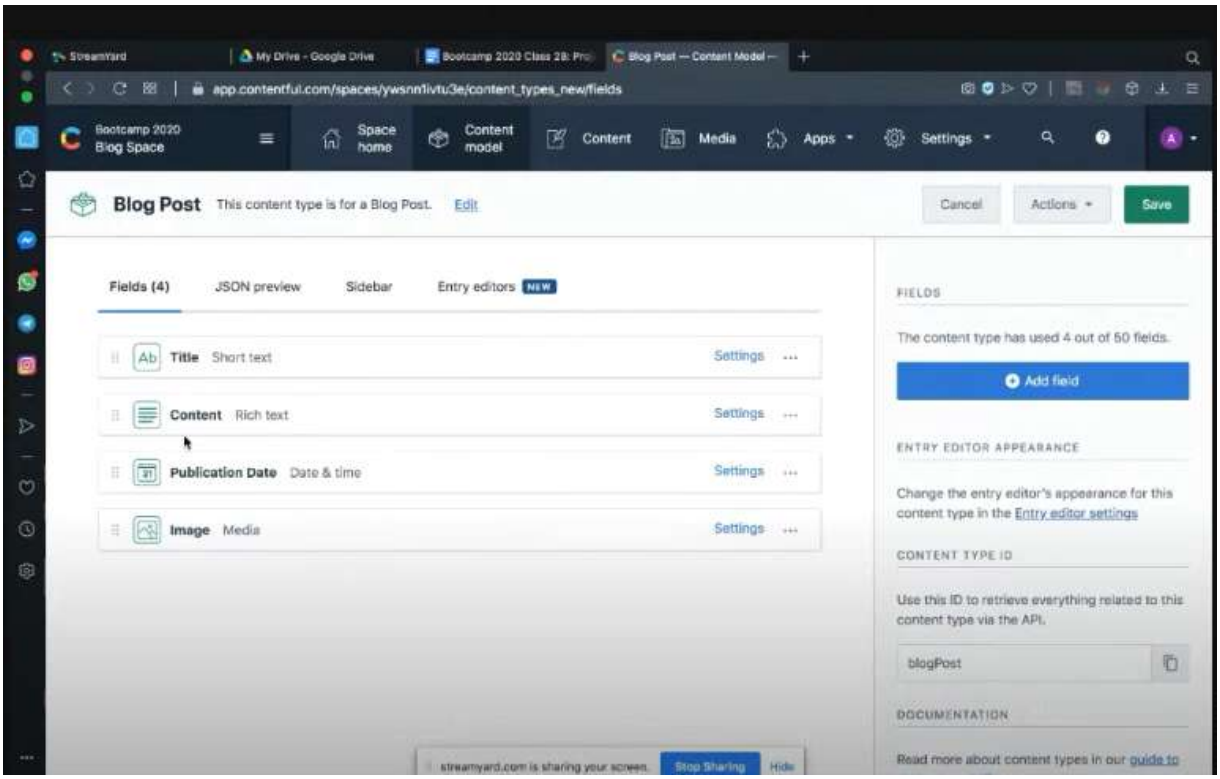
Step 10

Fetch Richest Contentful

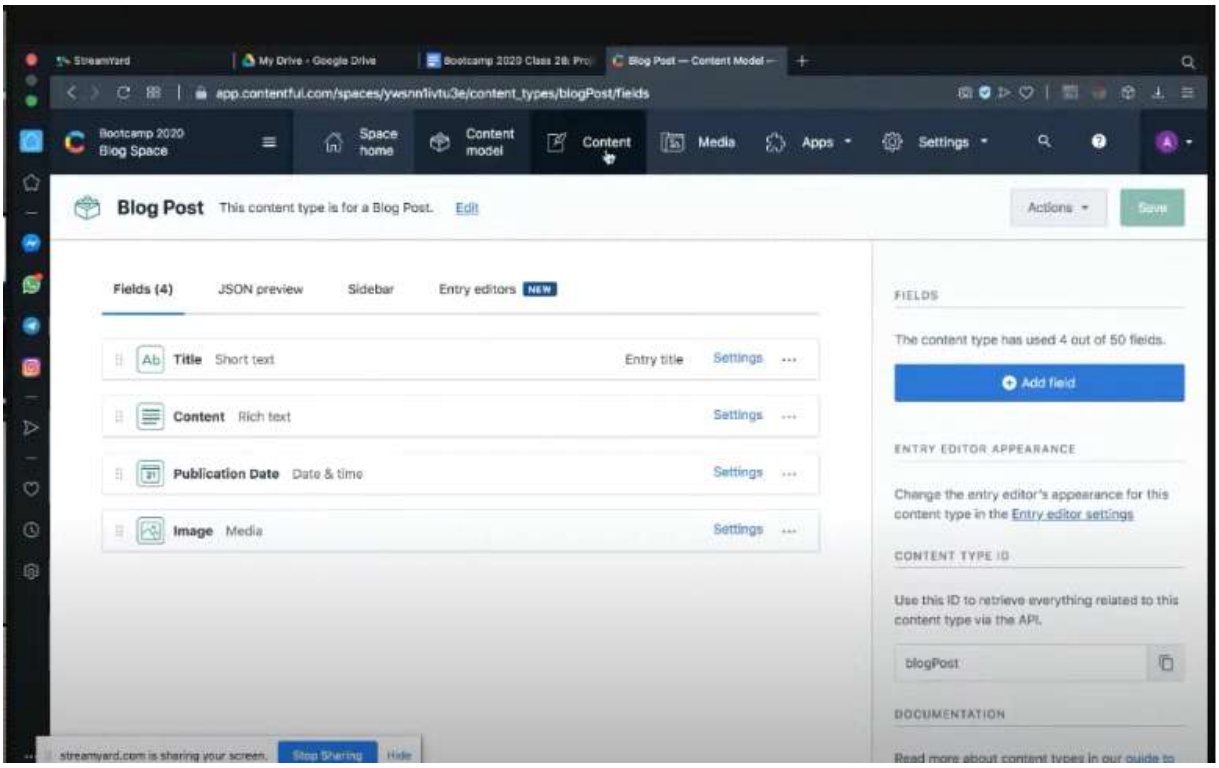
- Go to rich text



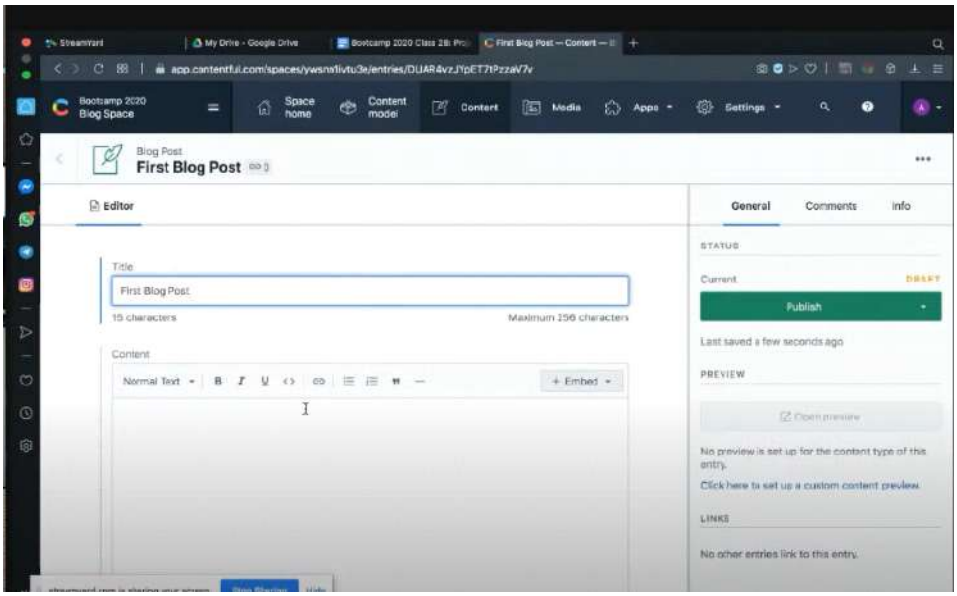
2. Create rich text
3. Then add a new file date and time
4. Add media
5. Every Blog post have this type of structure



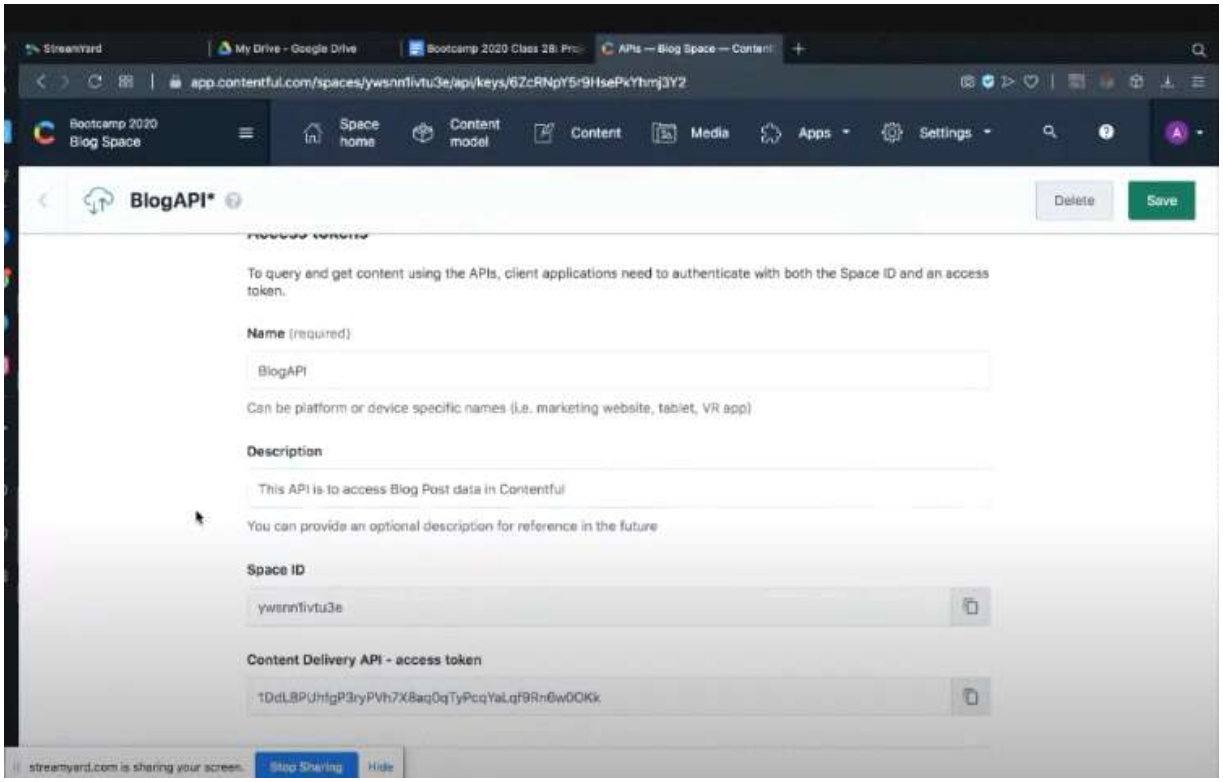
6. Save this structure
7. Now go to the content



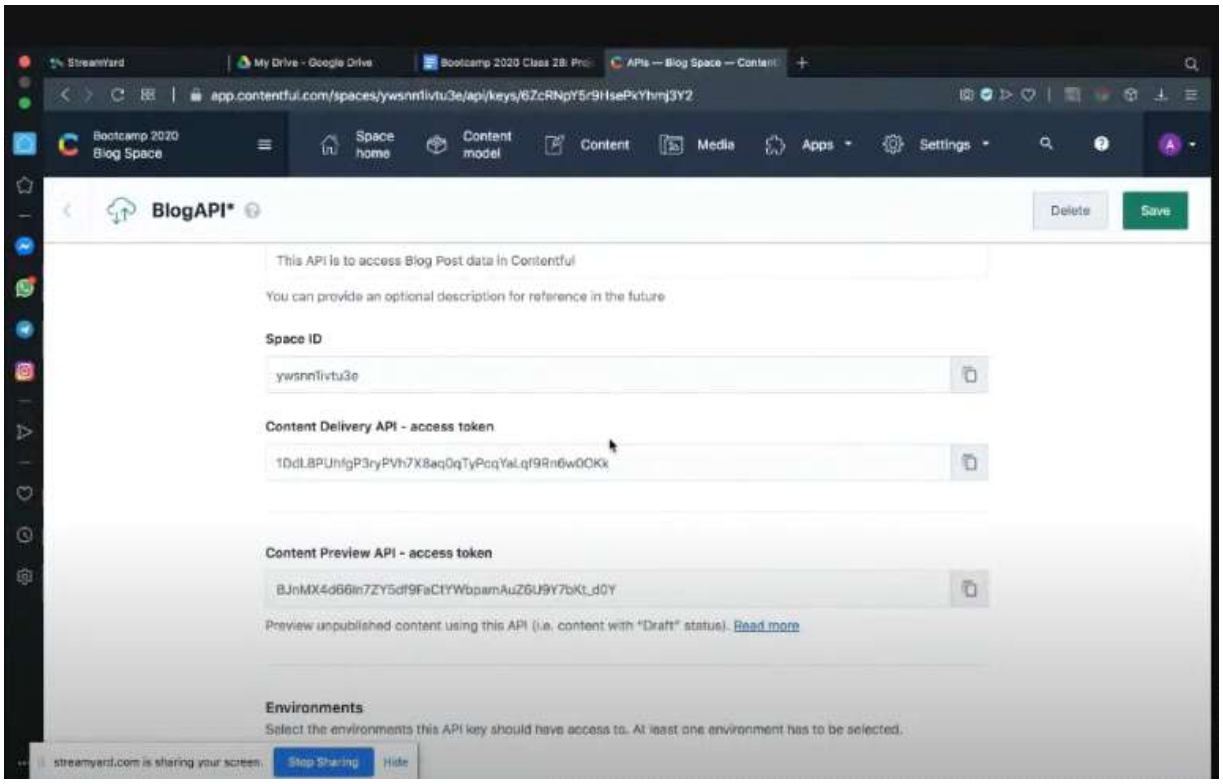
8. Add a blog post
9. Fill these forums



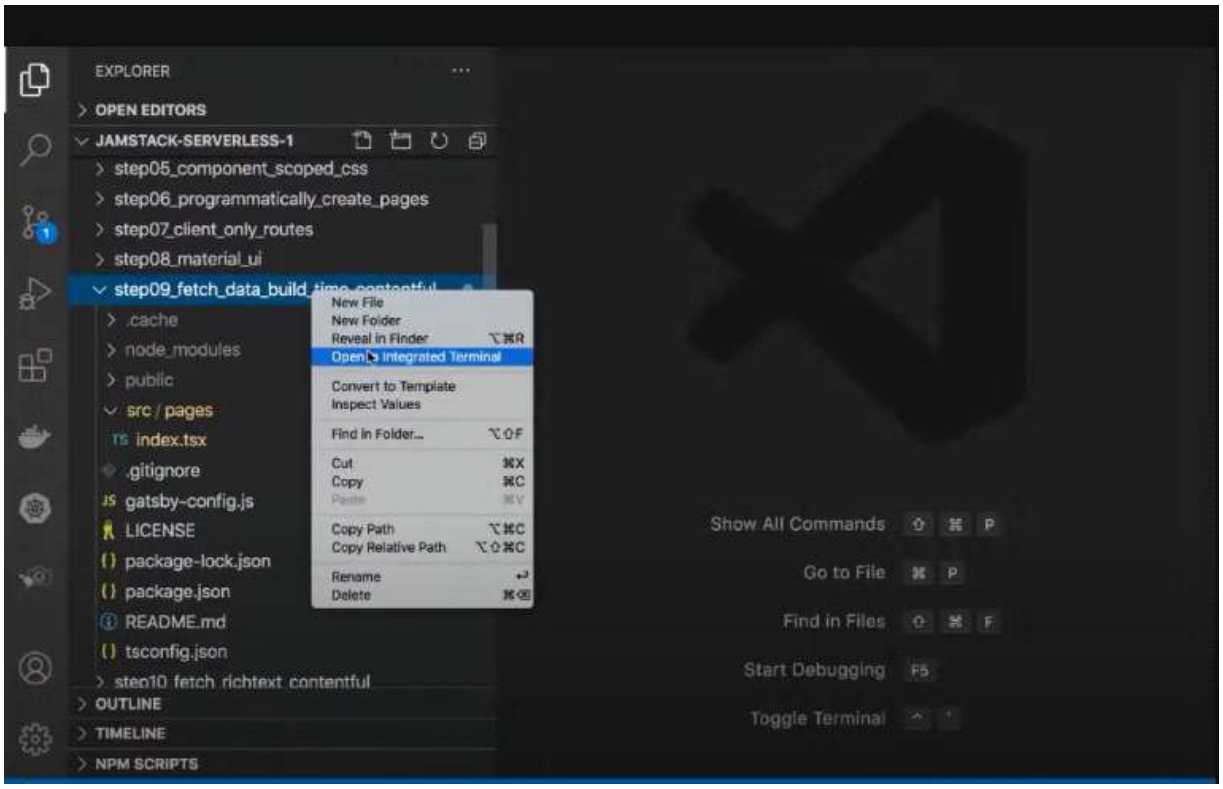
10. Publish the content
11. You can able to add another blog post using the same method
12. Go to settings
13. Go to API key and add a new API KEY
14. Create a new API key to access on Gatsby



15. Save the space id and content delivery API- access token

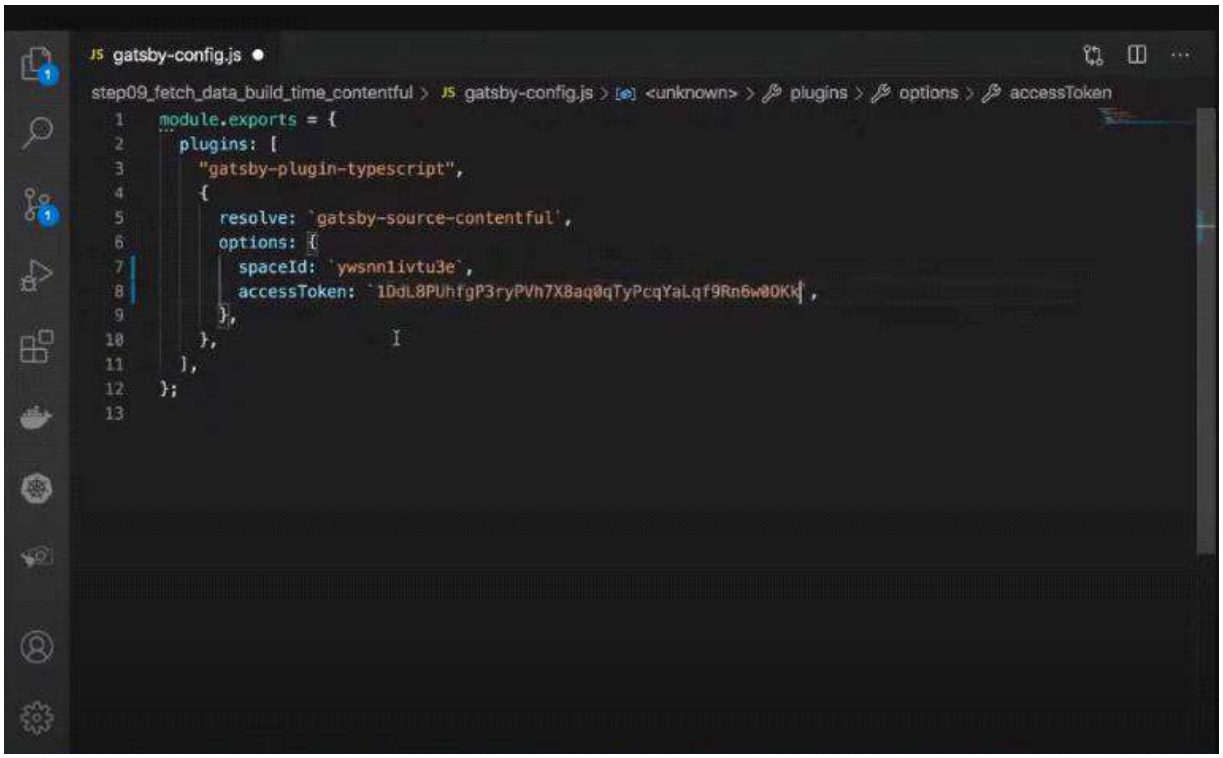


16. Save the blog API
17. Go to step no.9 in Gatsby and add contentful

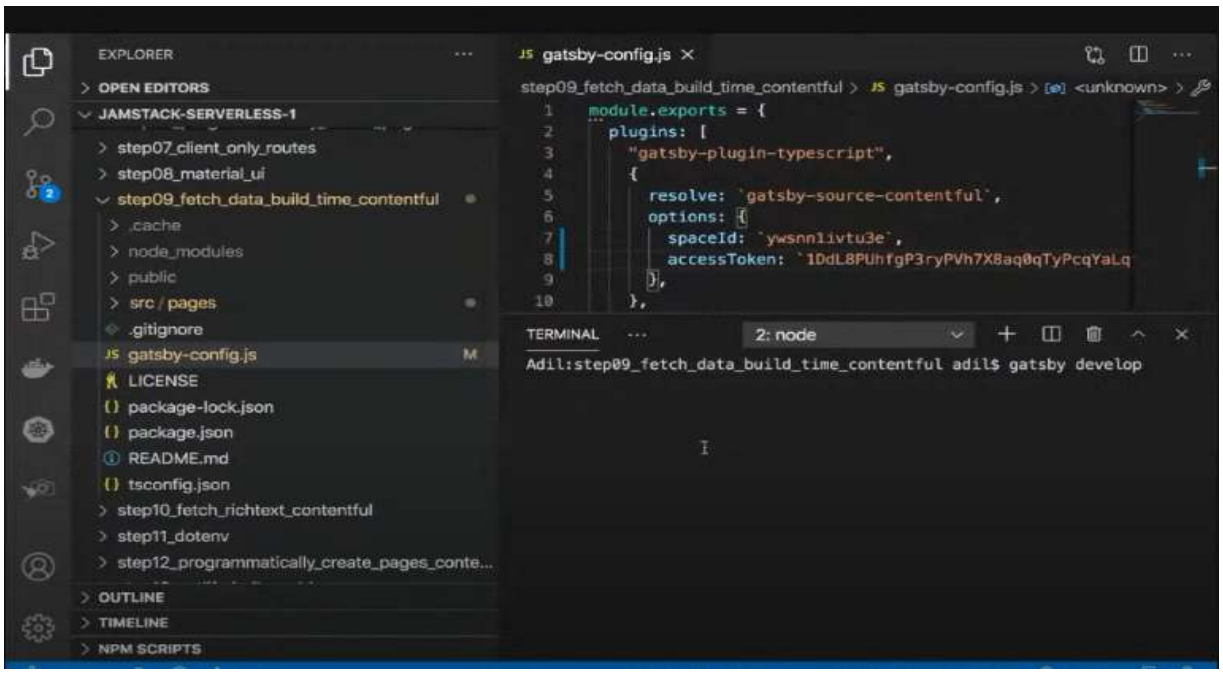


18. Go to gatsby config.js

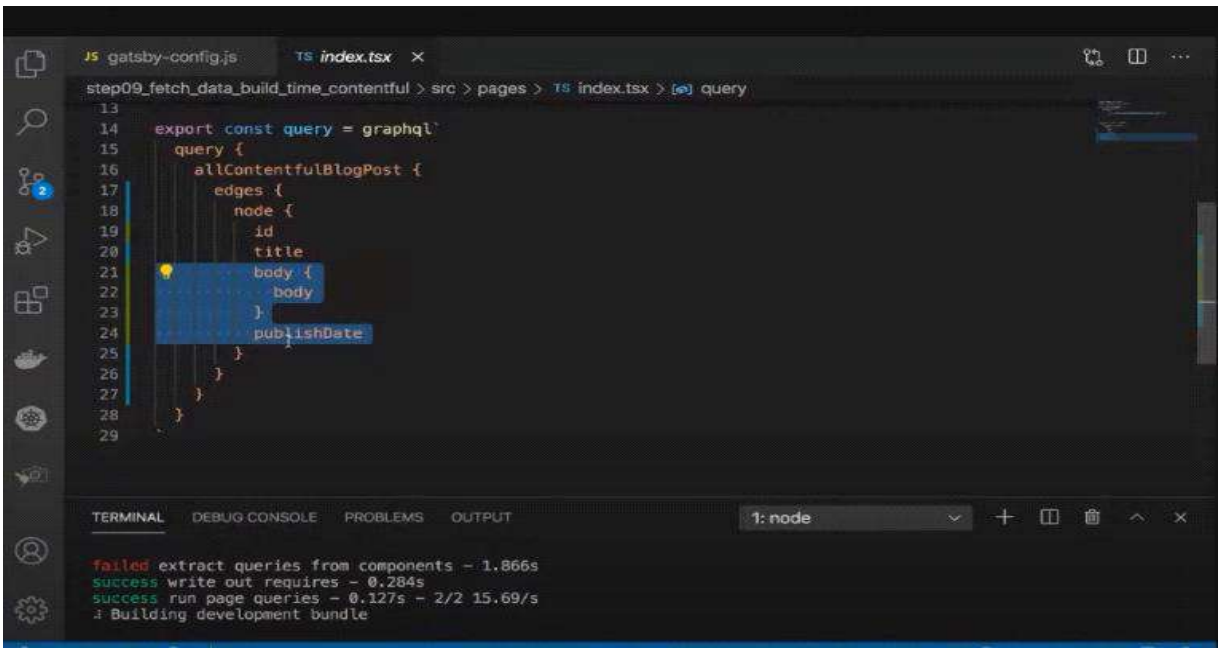
19. Add saved space id and access token in gatsby-conifg.js



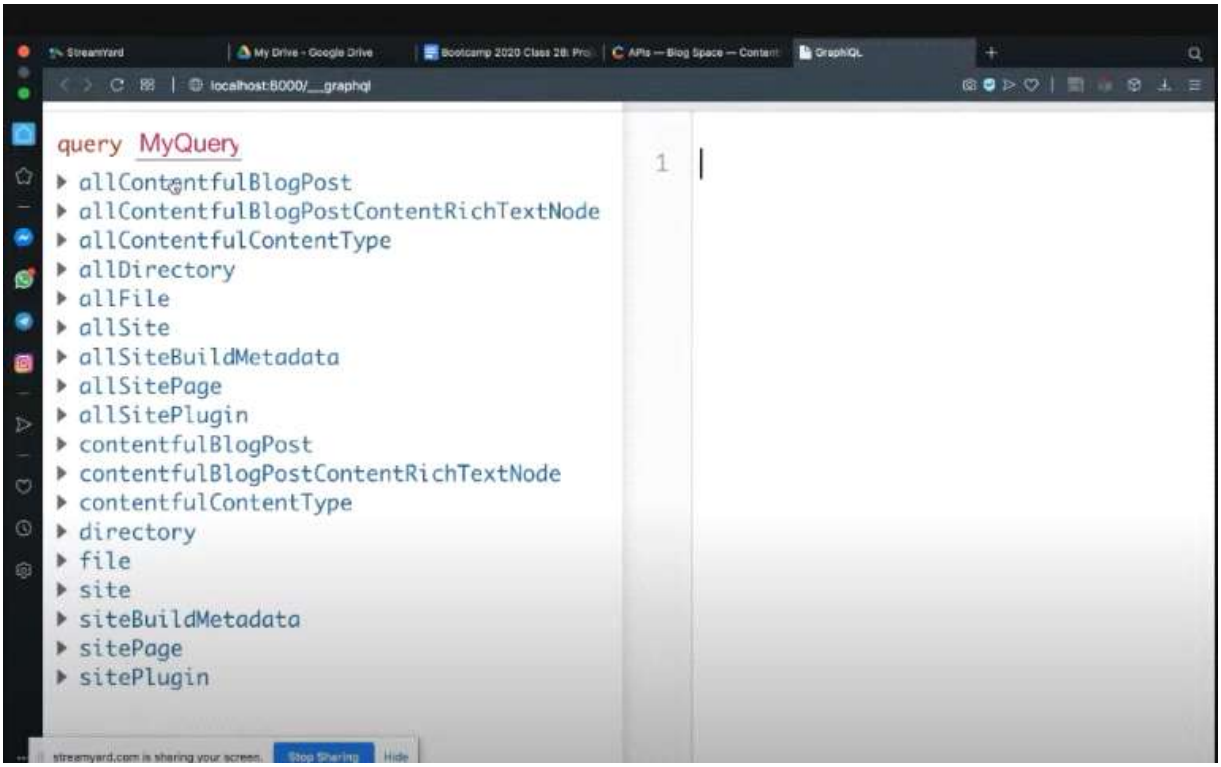
20. Run gatsby develop in terminal



21. Go to index.TSX page
22. Remove these codes

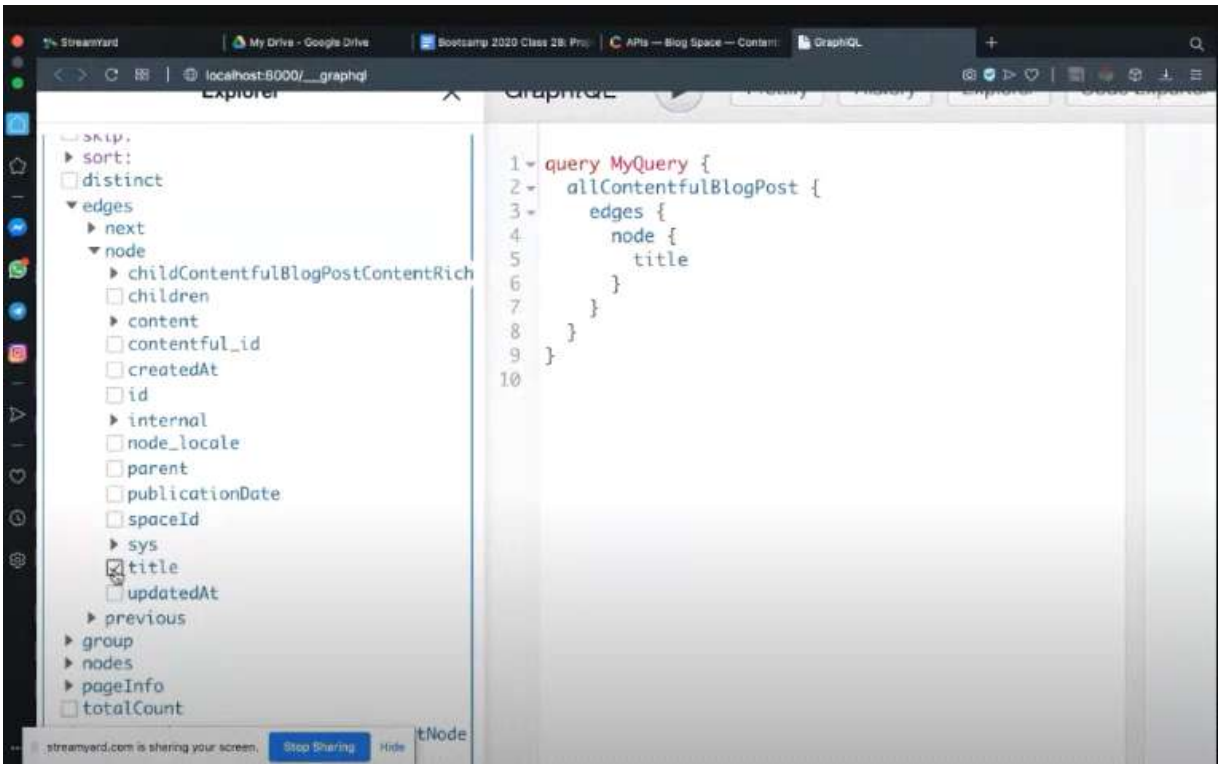


23. Go to localhost8888
24. Go with allcontentful blogpost query

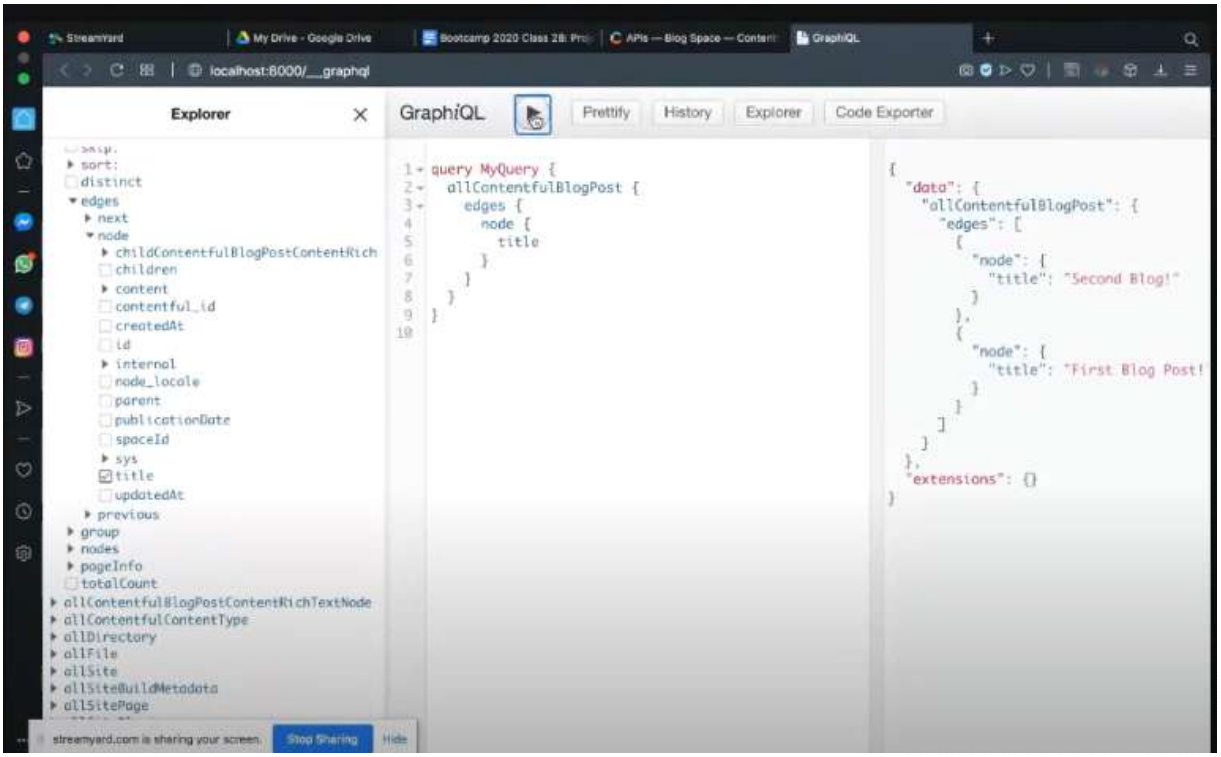


25. Go to edges

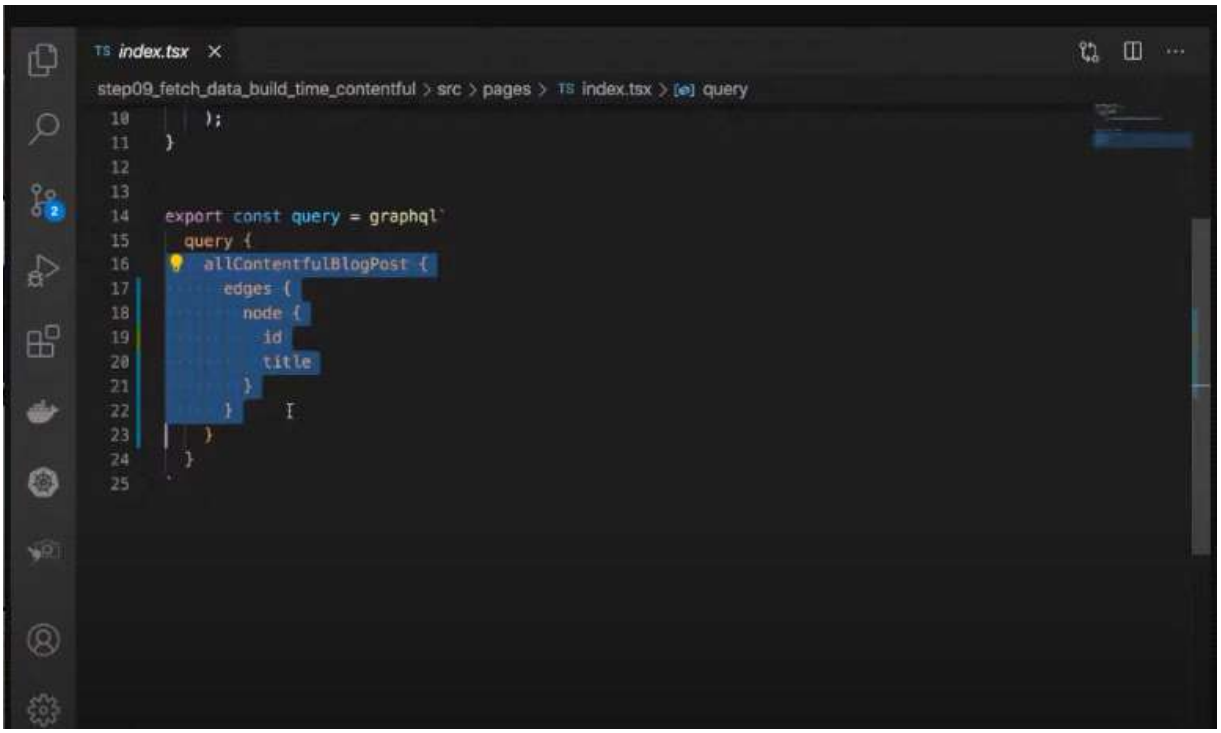
26. Go to nodes
27. Give access to titles



28. Run this



29. You can run all of the structures that you made in the blogpost
30. Copy the query
31. Go to an index page in Gatsby
32. Remove this



33. Paste the query code over this removed code
34. Go to the actual page “localhost8000”
35. If you add the publication date

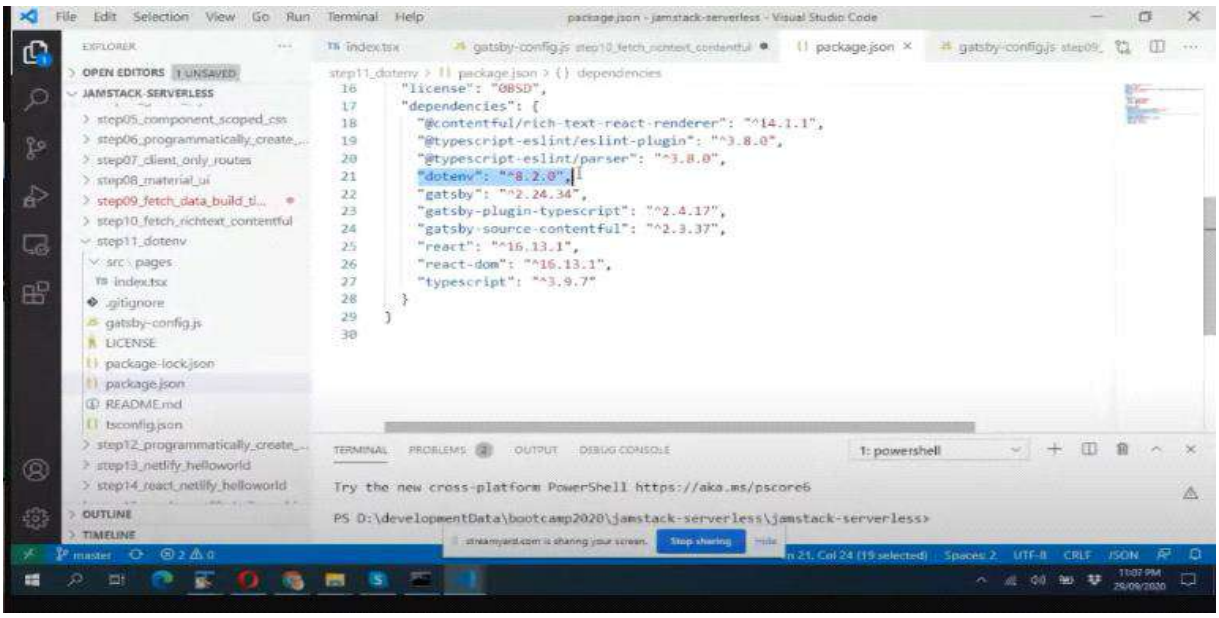
```
import React from "react";
import { raphql } from "gatsby";

export default ({data)} => {
    console.log(data);
    return (
        <div>
            <div>{data.allcontentfulBlogpost.edes[0].node.title}</div>
            <div>{data.allcontentfulBlogpost.edes[0].node.publication date}</div>

              <div>{data.allcontentfulBlogpost.edes[0].node.title}</div>
       <div>

);
```

36. Then if you add all of your created structures

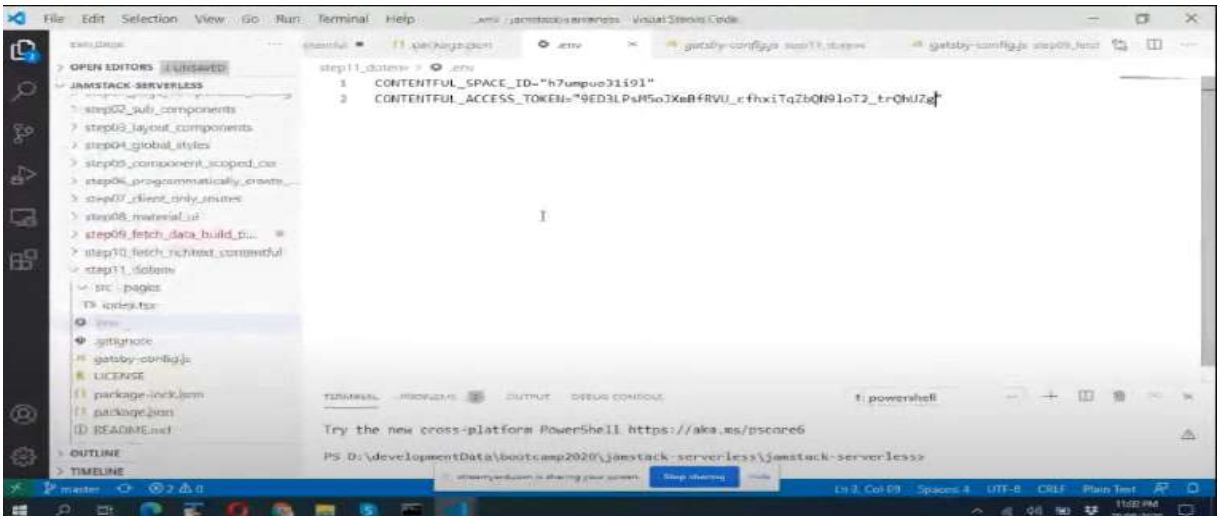
Step 11

Dotenv

- Install "dotenv"



- Create a File name ".env"
- Inside the new file, you have to add values like this

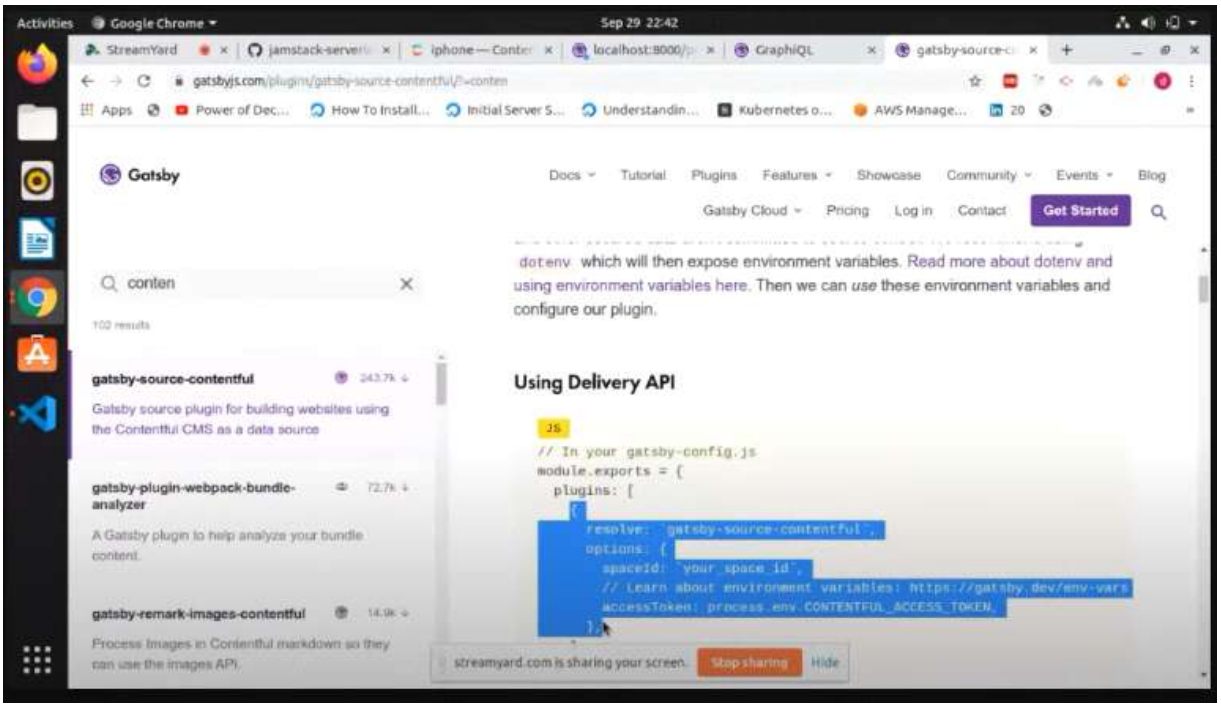


Note: Dotenv is not a part of your repo but it is a part of your development and beneficial for your application to lookover the values

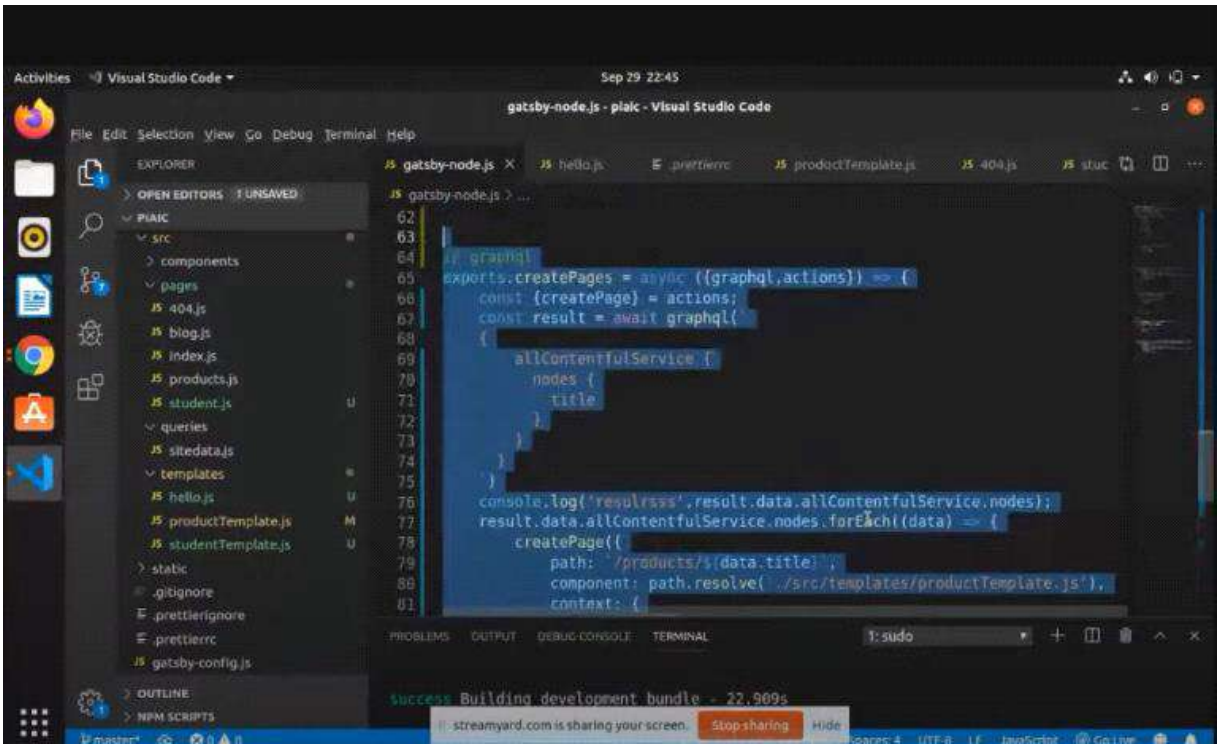
Step 12

Programmatically Create Pages

1. Go to Gatsby original site
2. Go to plugin library in Gatsby
3. Search the plugin contentful
4. Install this code



5. Go to gatsby-node.js and rewrite the code of this page



6. Rewrite the code in this format

```
exports.createpaes = async ({graphql, actions}) => {
        const  {createpae} = actions;
        const result = raphql {
          )
}
```

7. Go to the query
8. Copy the query code and paste in node.js

```
exports.createpages= async ({graphql . actions }}
  const (createpage) = actions;
  const result = graphql {
     allcontentful service {
     node {
     title {
    {
{
```

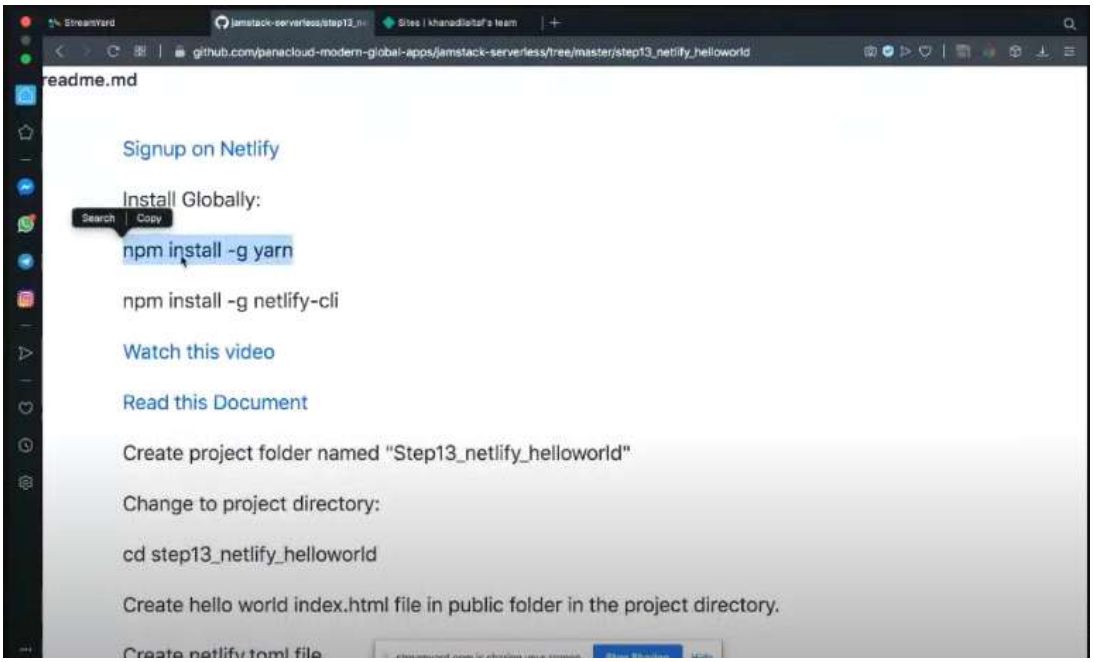
9. Finalize your code.

Session-35

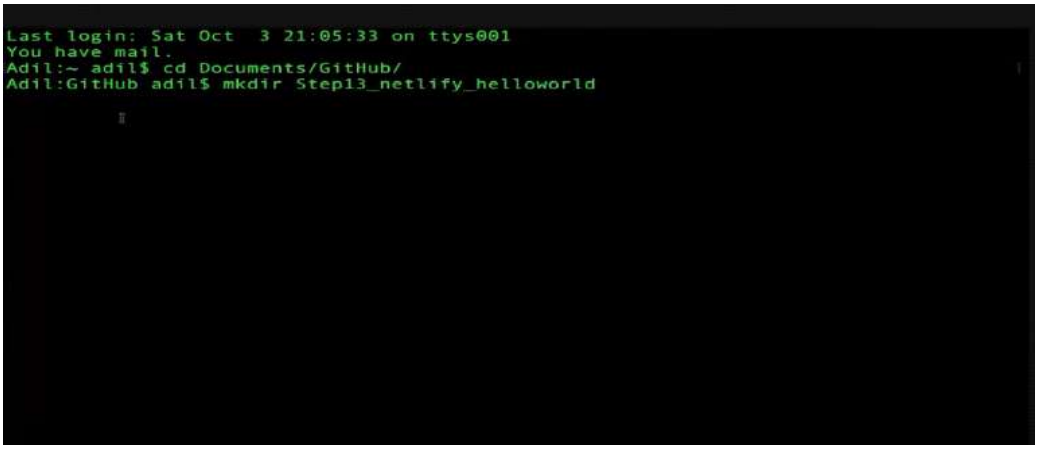
Step 13

Netlify HelloWorld

- Create a simple helloworld application using netlify
- Sign up on Netlify
- Install -g yarn on your terminal

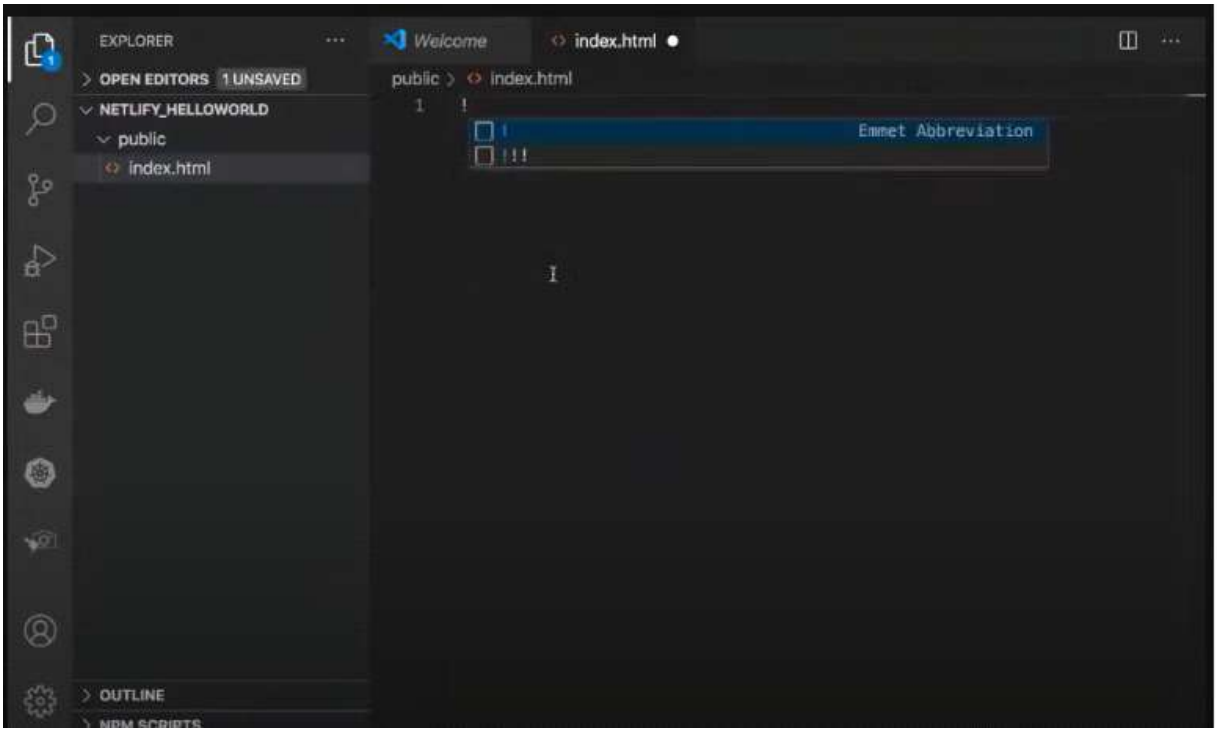


- Also install -g netlify-cli
- Create a folder in your terminal



- Open this folde

- Create an index html file
- Use Emmet in the file



- Give a title and body name

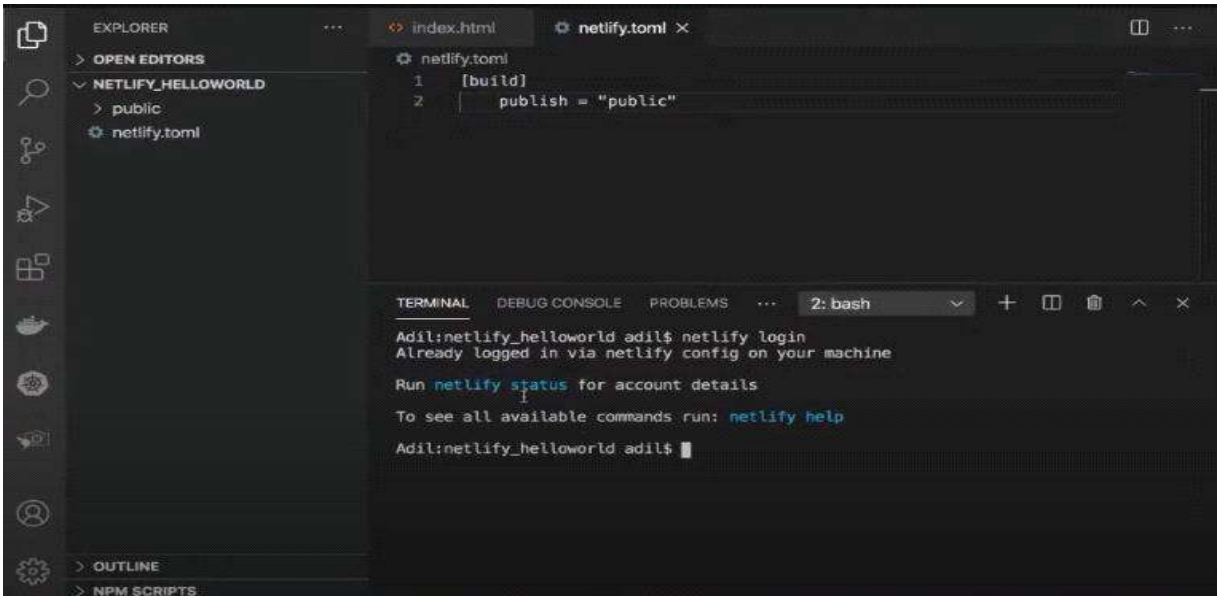
```
<!DOCTYPE html>
    <html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial scale>
         <Title>Heloo Netlify</title>
    <head>
    <body>
        Hello world from Netlify and Adil
    <body>
      <html>
```

- Creat netlify.toml file in the main page
- Create a build and public in this folder

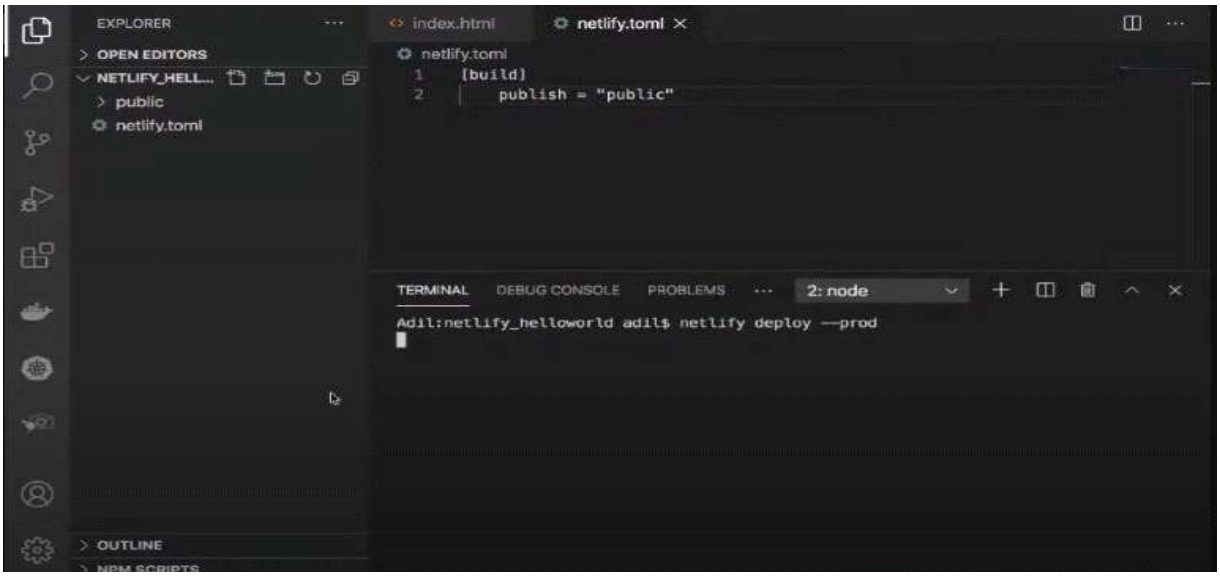
[build]
Publish = “public”

- Open the integrated terminal
- Named as netlify dev

- After startup you can access your application on "localhost8888"
- Open the new terminal to log in to the netlify
- Add Netlify login here

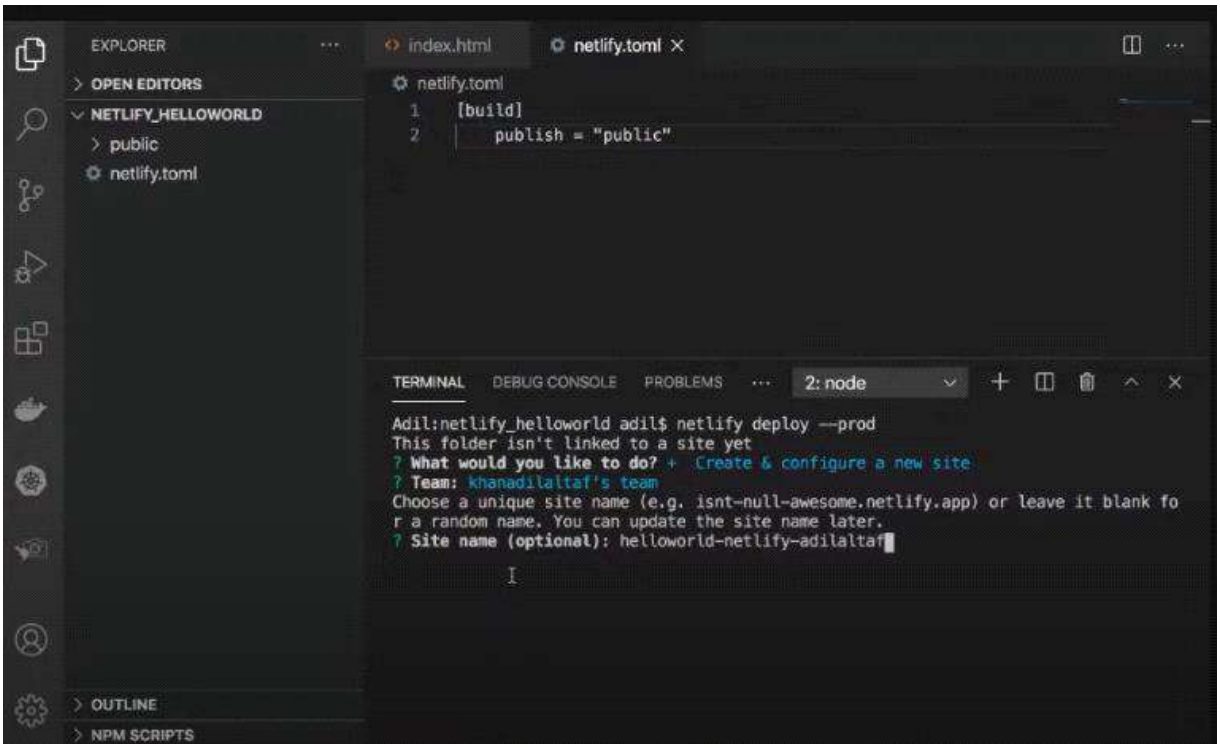


- Again open a new terminal and add netlify deploy

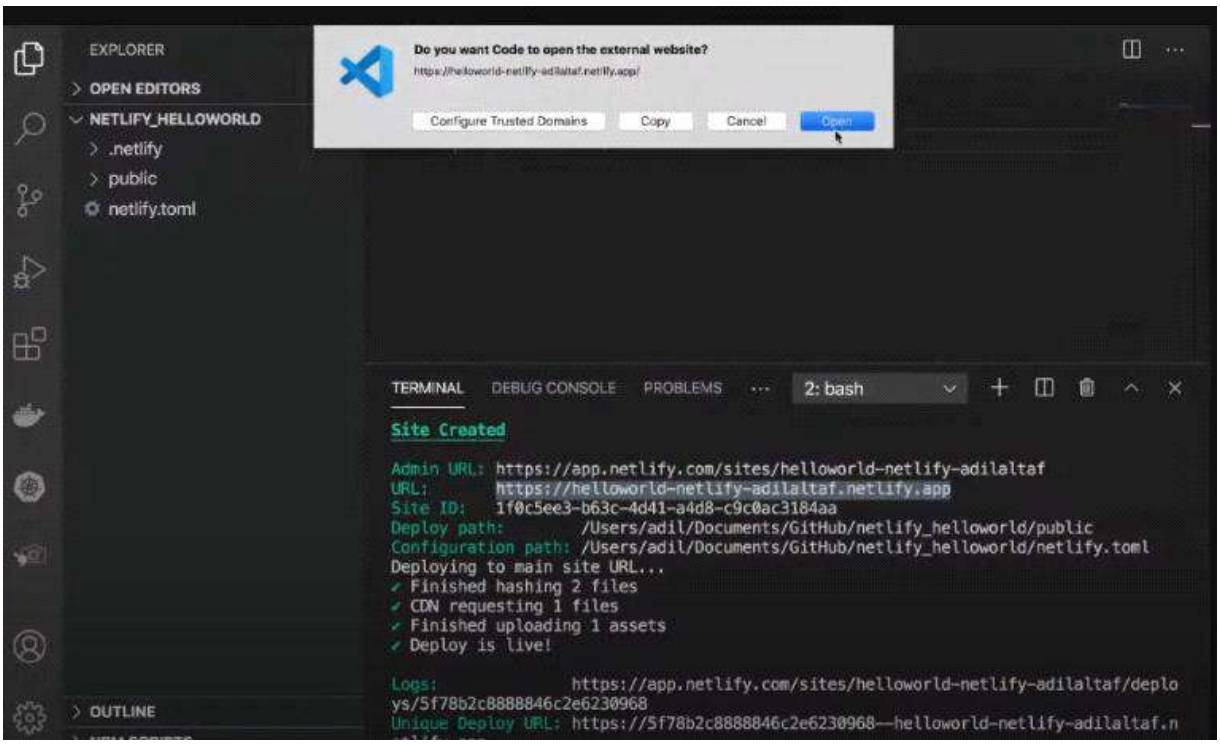


- This will ask you a few questions

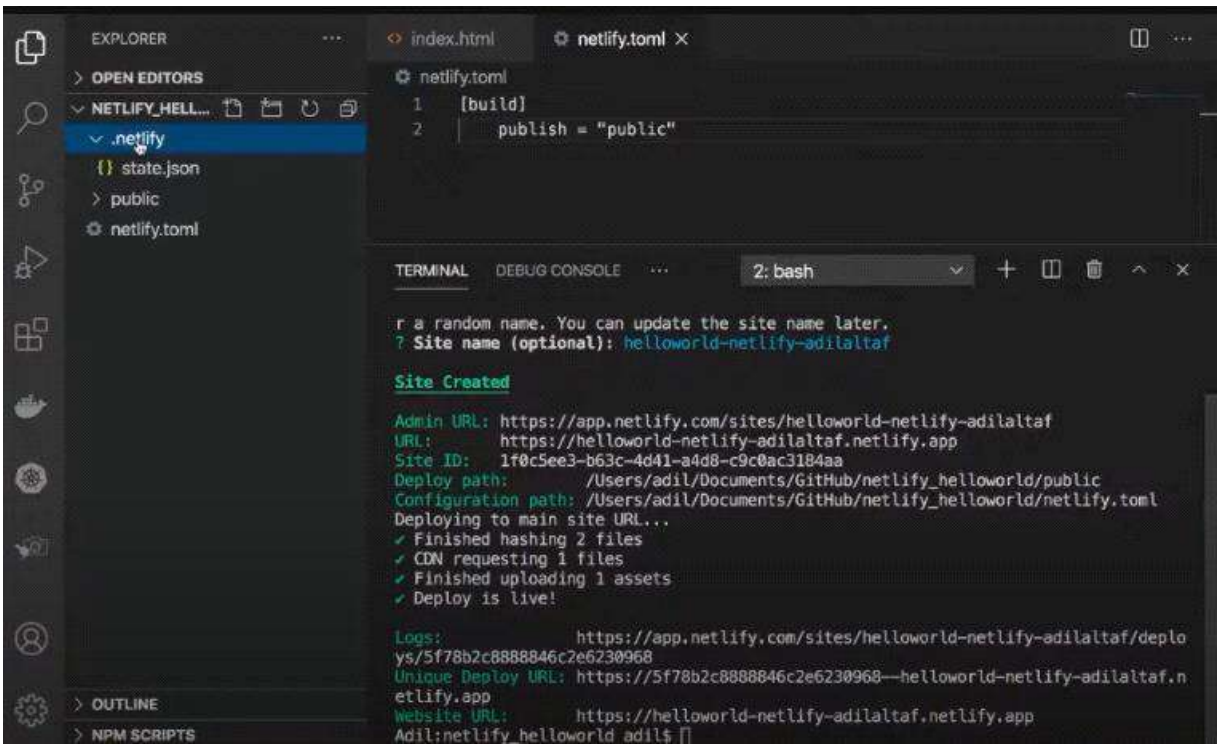
- You have to select a new site
- Then you have to select the team name
- Give a random name to the site



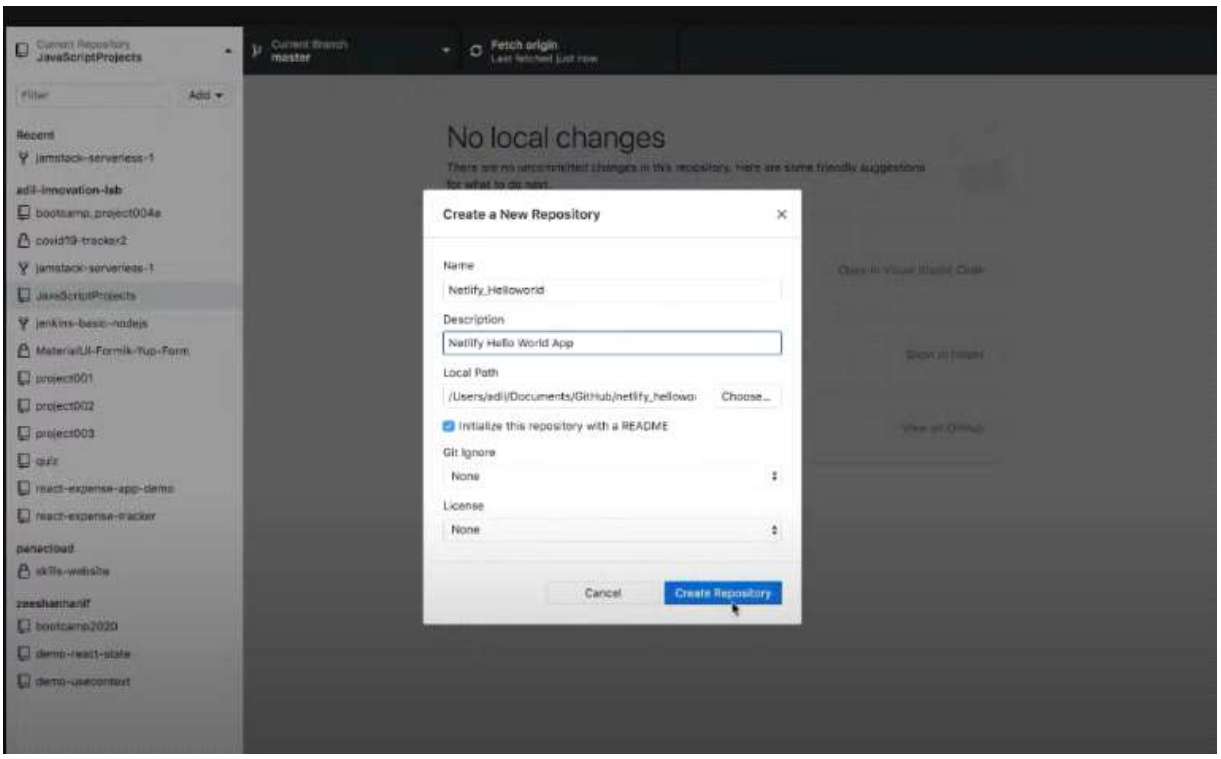
- After completion go to this link



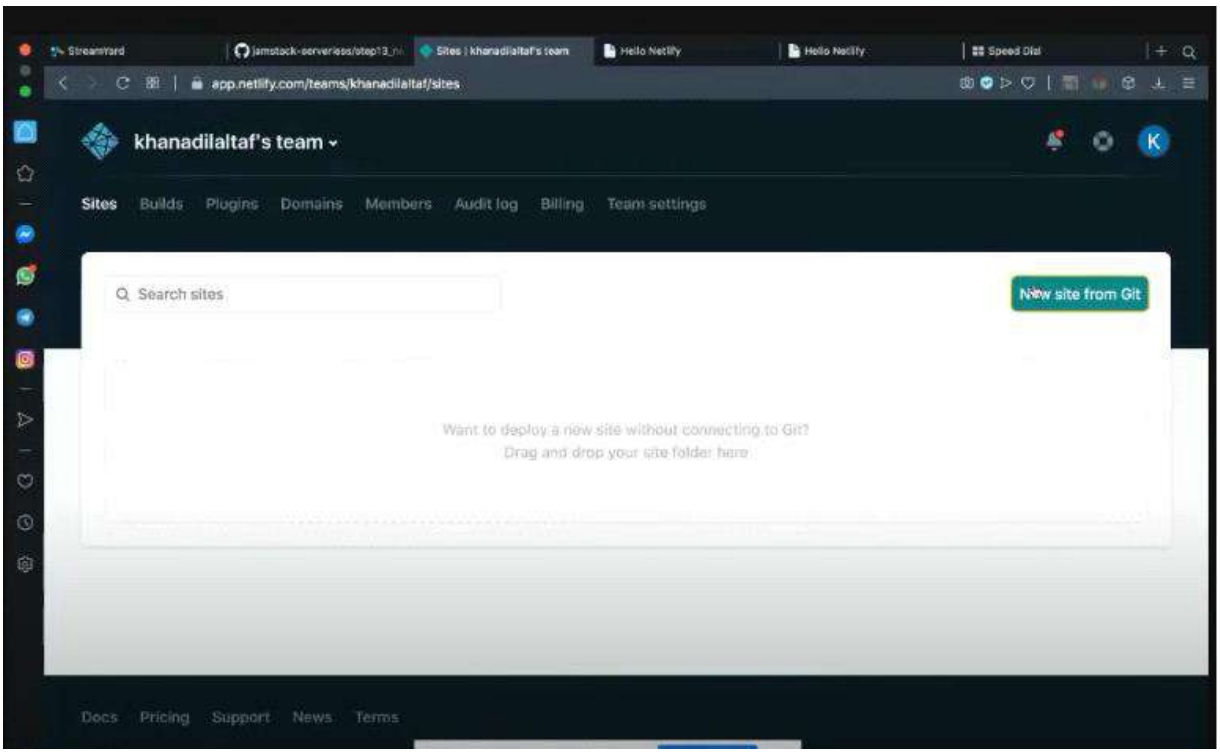
- The helloworld application is now running with netlify app
- When you deploy it in terminal, you see a folder, netlify
- Delete the netlify folder



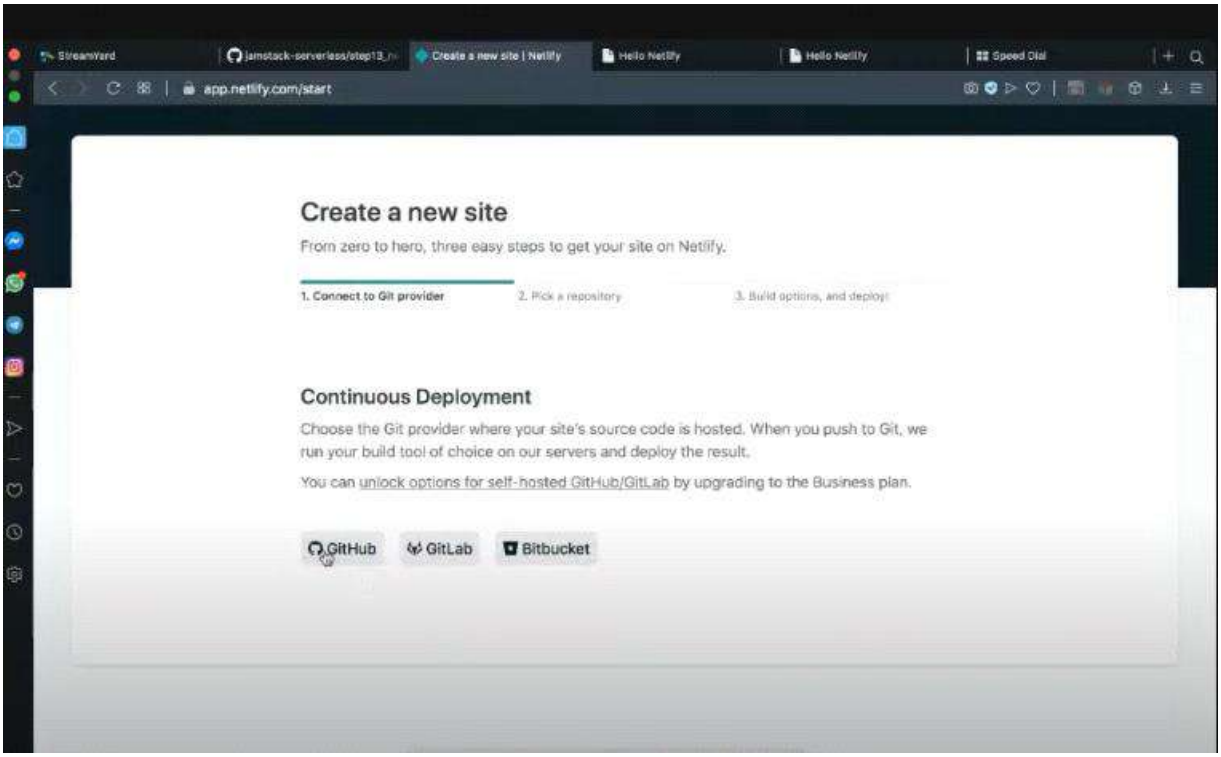
24. Create a new repository



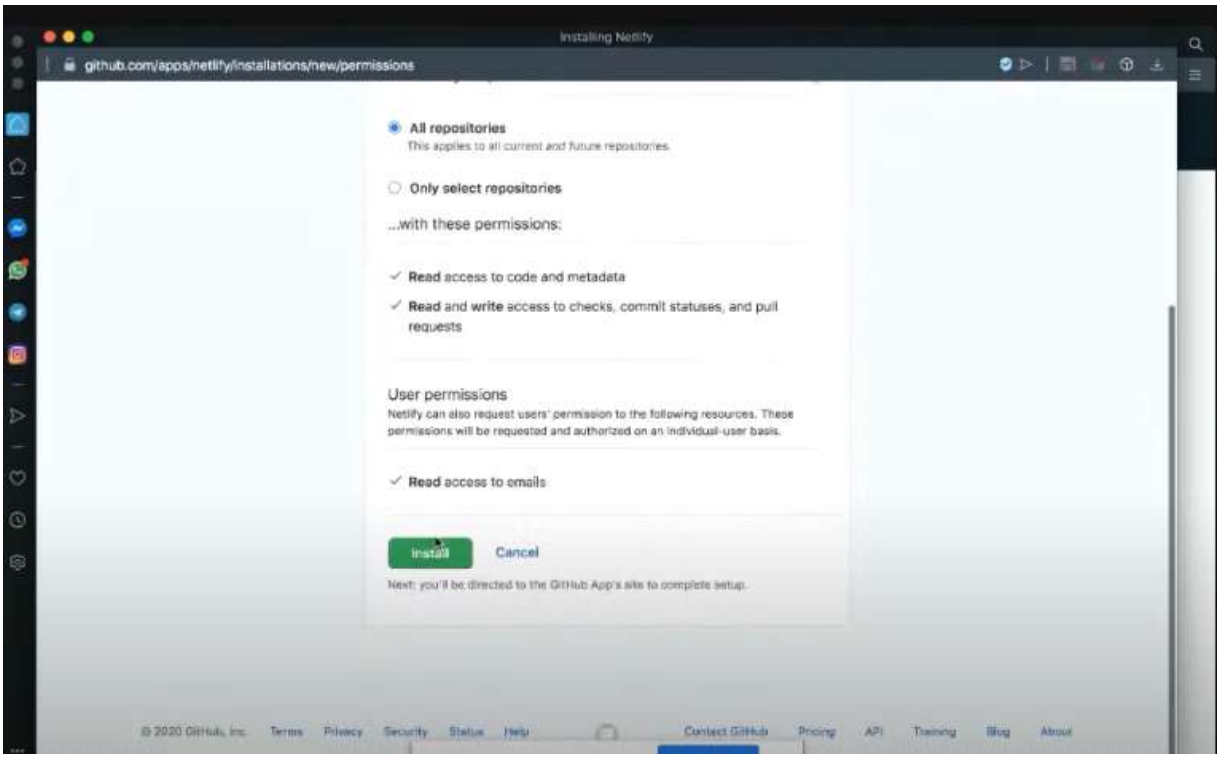
25. Publish it
26. Go to Netlify site
27. Go to the new site from GIT



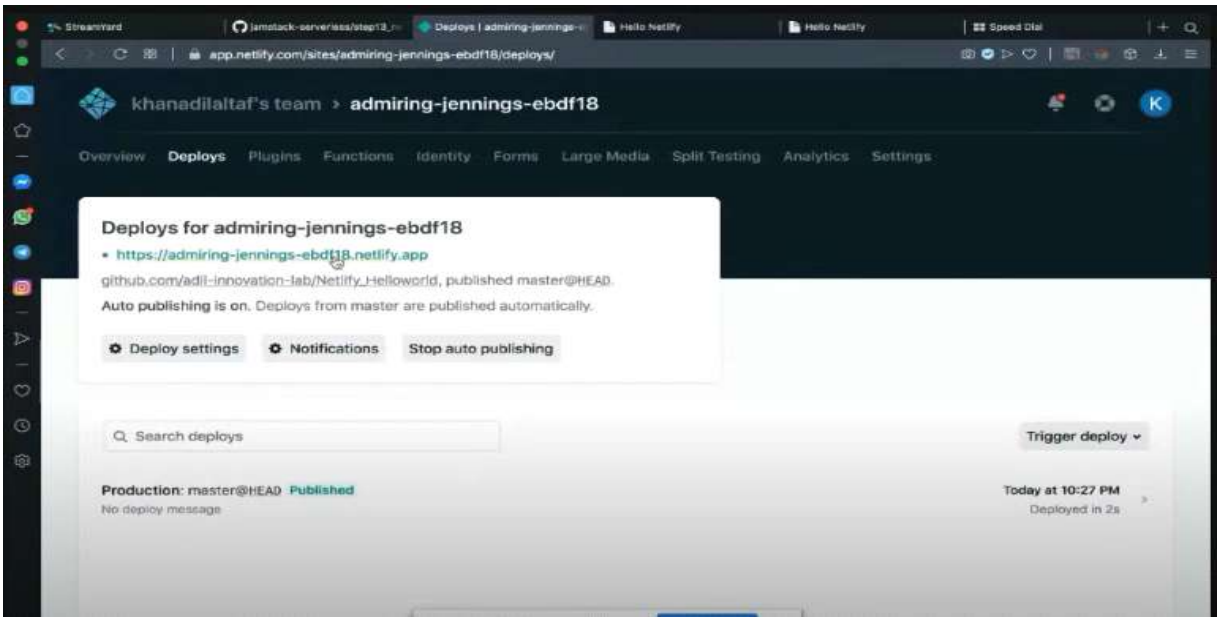
28. Connect to GITHub



29. Authorize Netlify
30. Now configure Netlify on GitHub
31. Install Netlify



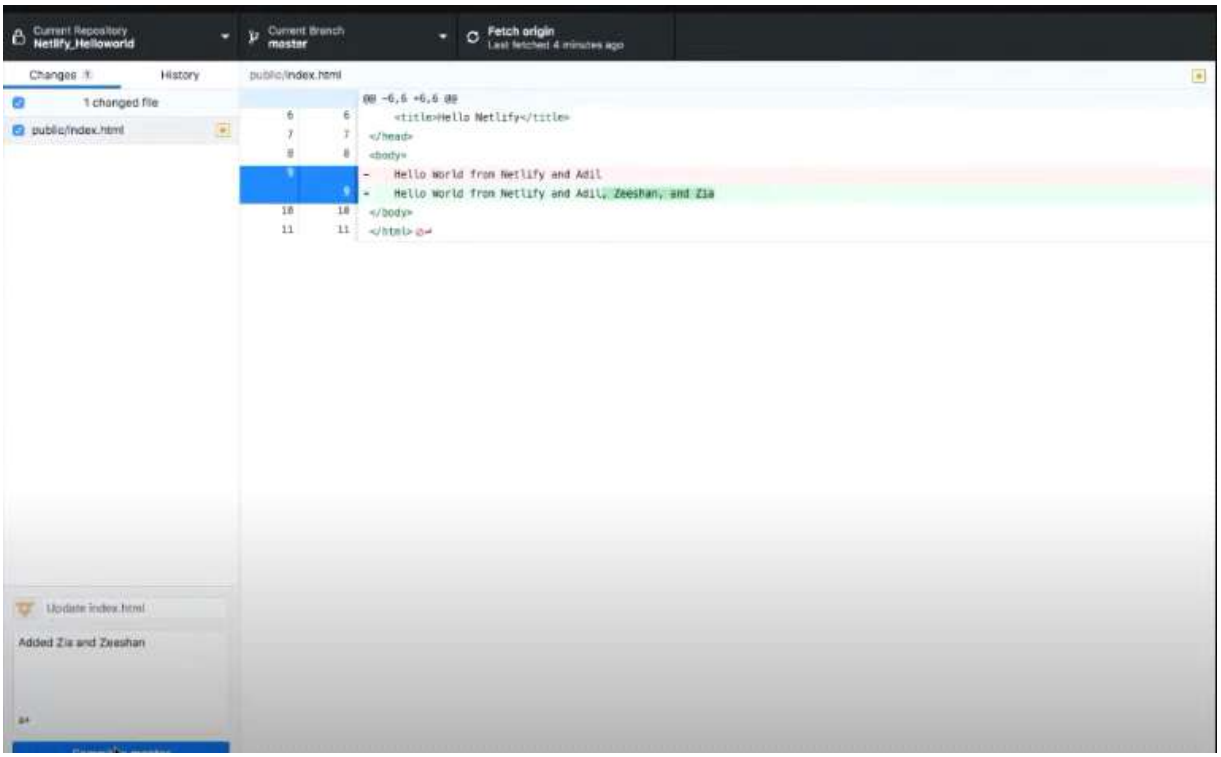
31. Go to the page that you created
32. Then deploy the site
33. After deploy go to this link



34. Make some changes in your code

```
<!DOCTYPE html>
<html lang="en">
<head>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial scale>
       <Title>Heloo Netlify</title>
 <head>
 <body>
     Hello world from Netlify and Adil
 <body>
    <html>
```

36. Then add this to master

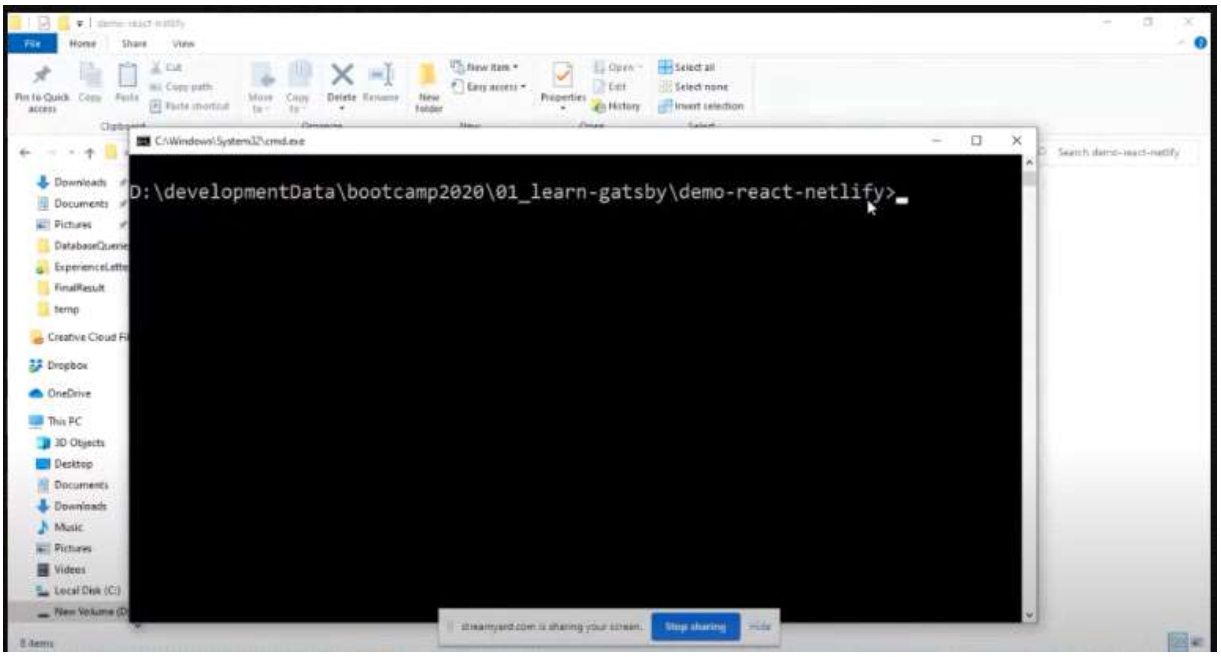


37. Then push origin

Step 14

React Netlify Helloworld

- Go to Create app
- Create Netlfiy demo app

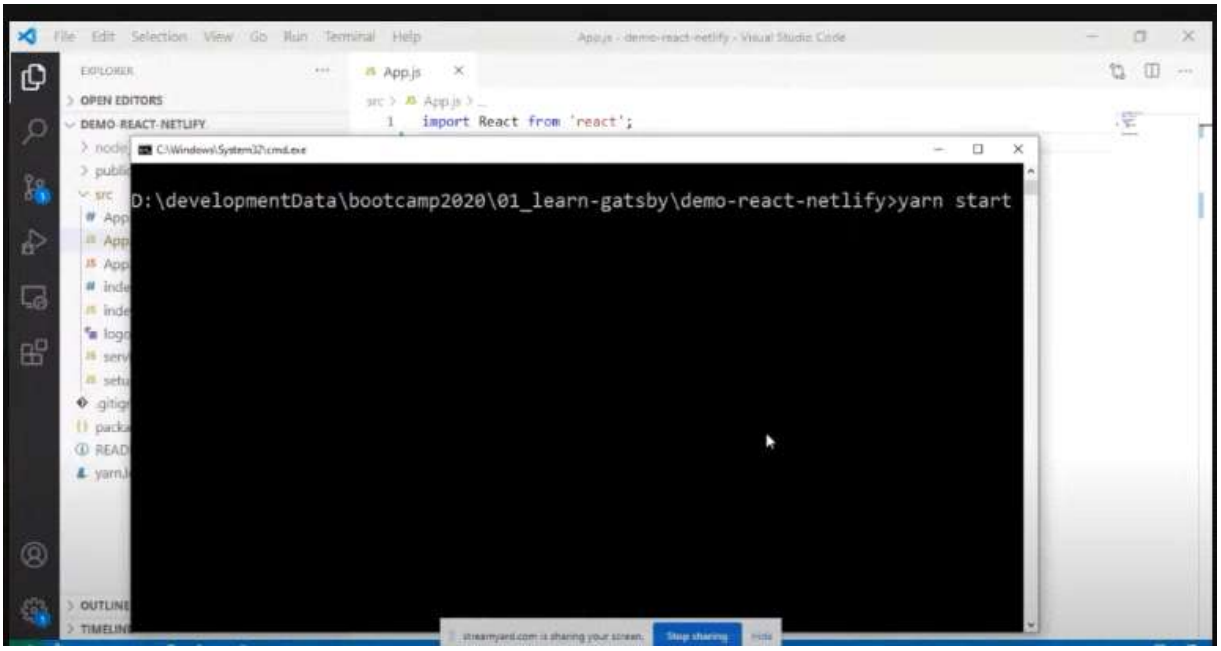


- Go to app.js and change this code

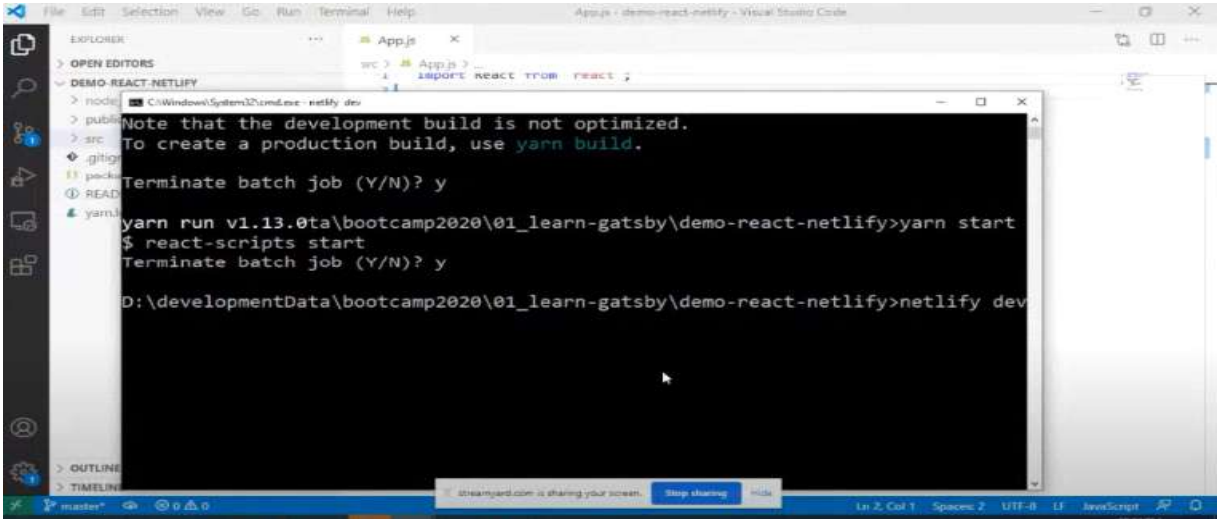
```
import React from ‘react’;
import logo from ‘./logo.svg’;
import ‘./app.css’;

  function App () {
  return {
    <div className=”App”>
        <header classNmae=”App-header”>
<log src=(logo) className=:App-logo”alt=”logo”/>
<p>
<a
  classNmae-“App-link”
```

```
  href=https://reactjs.org
target="blank"
rel="noopener noreferrer"
>
  Learn React
</a>
</header>
</div>
 );
)
```

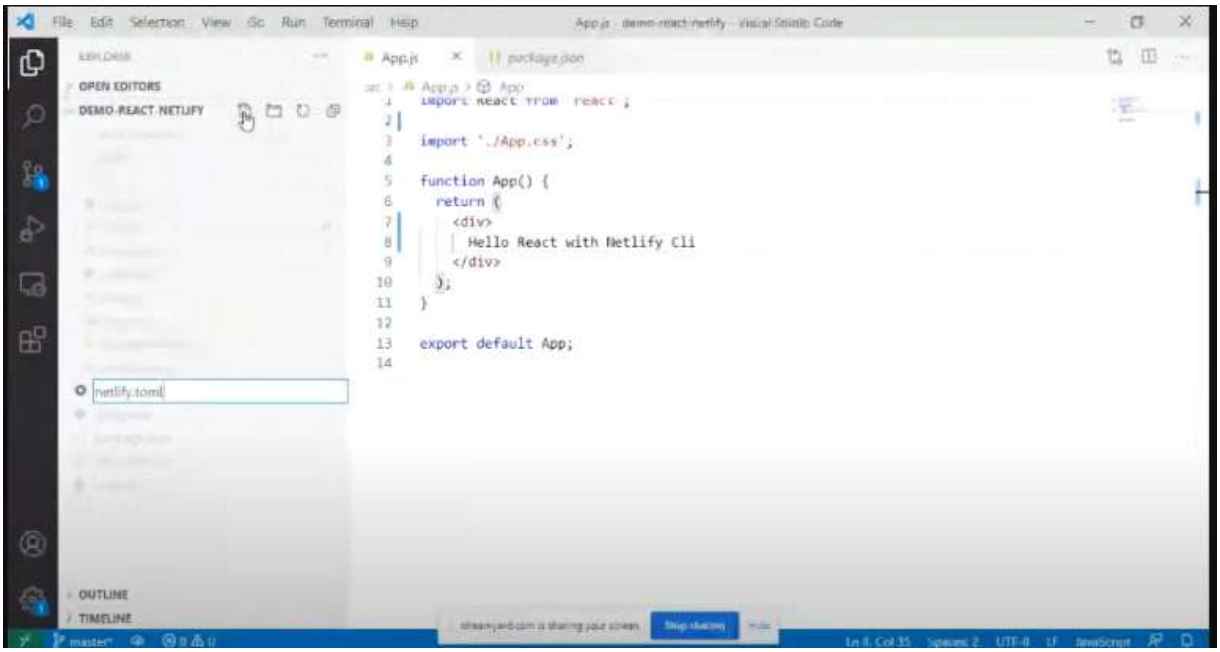
- To run this you have to add yarn start



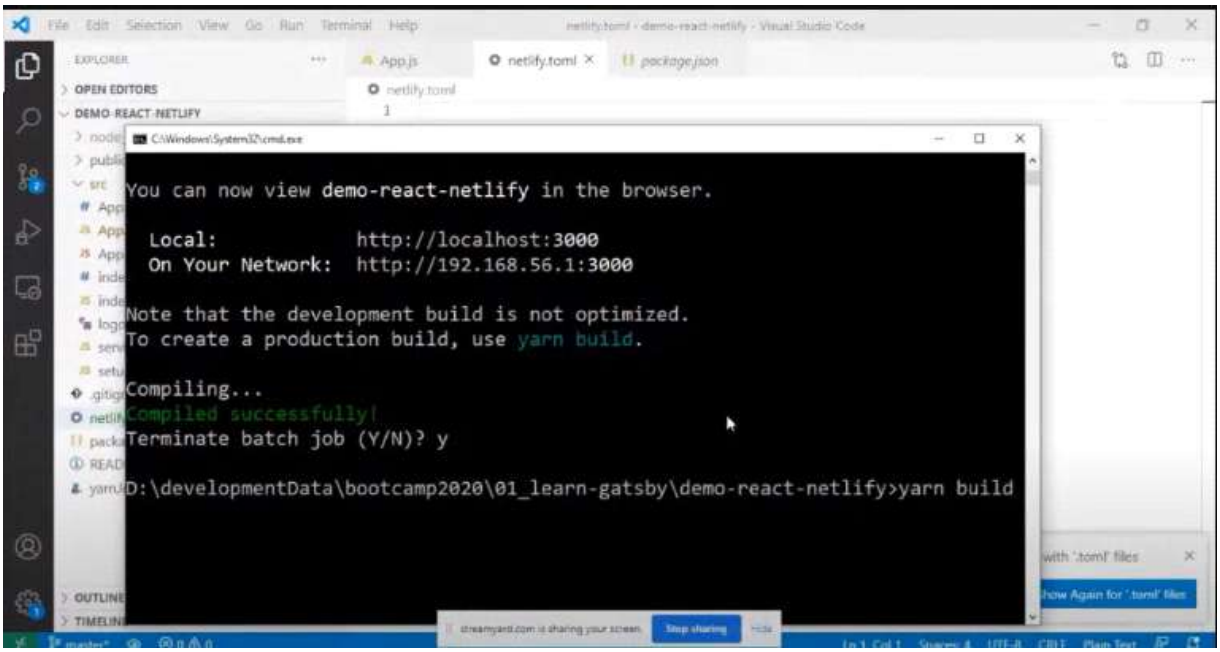
- After yarn start you have to add netlify dev



- You have the server running at "localhost8888"
- If you make a change in code it will directly shoe in sever
- Create a new file as netlify.toml



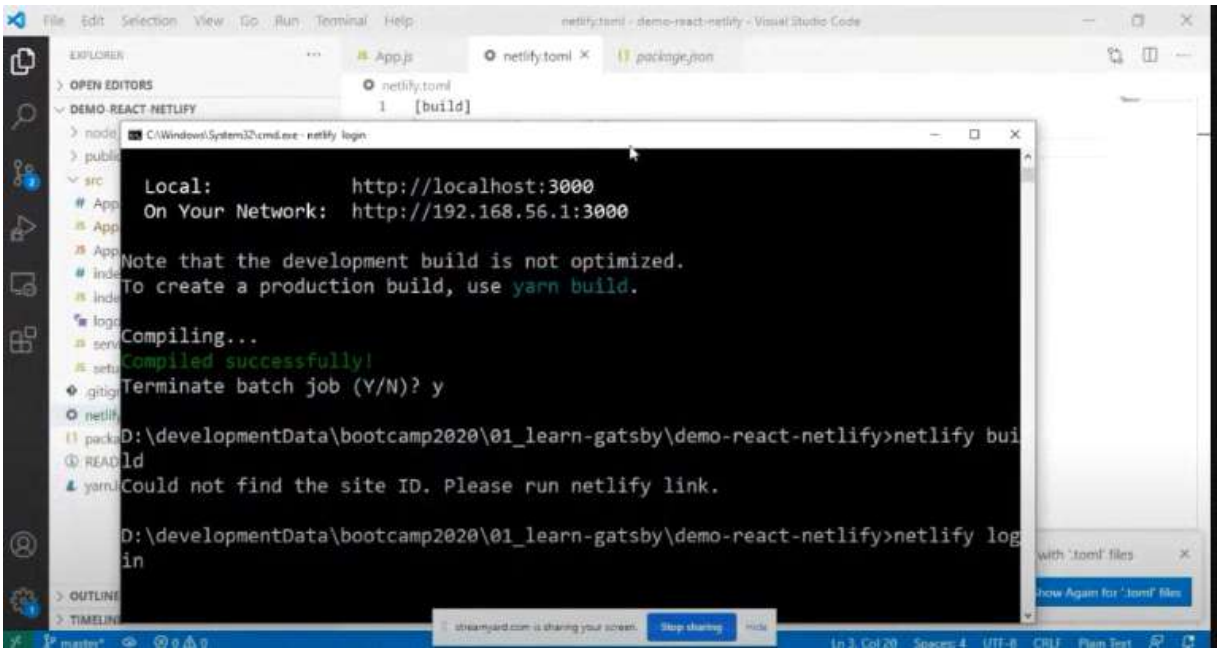
- Run a command yarn build in react app



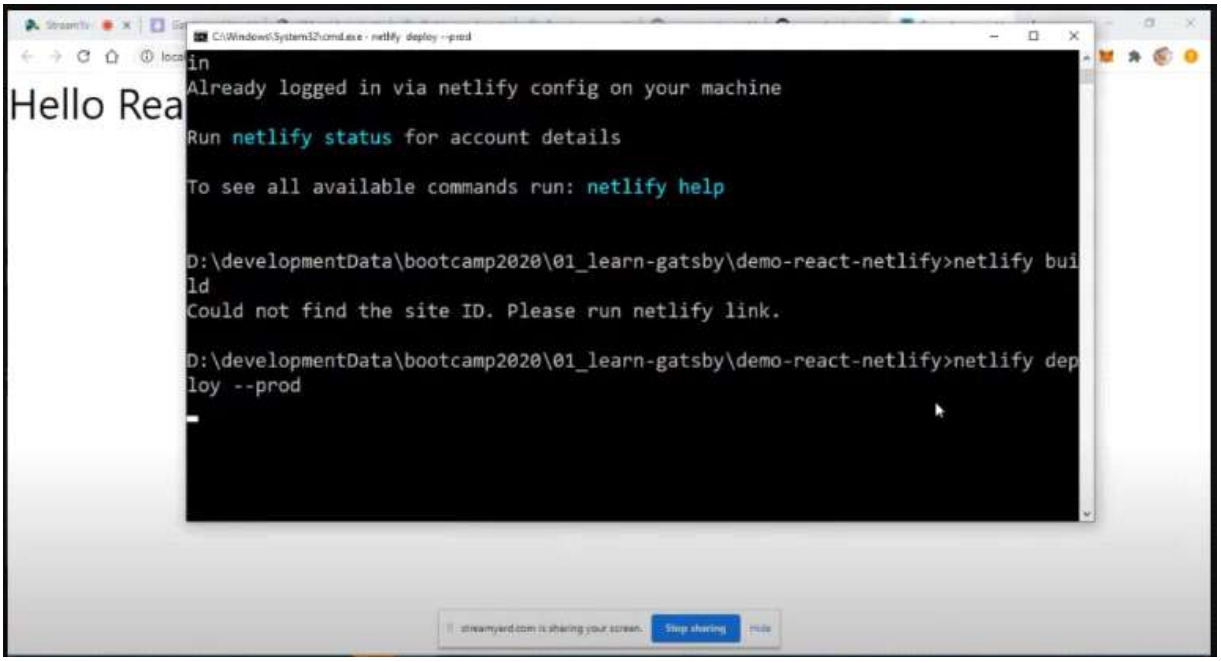
- Create a structure in netlify.toml

```
[build]
    command="yarn build"
    publish= "build/"
```

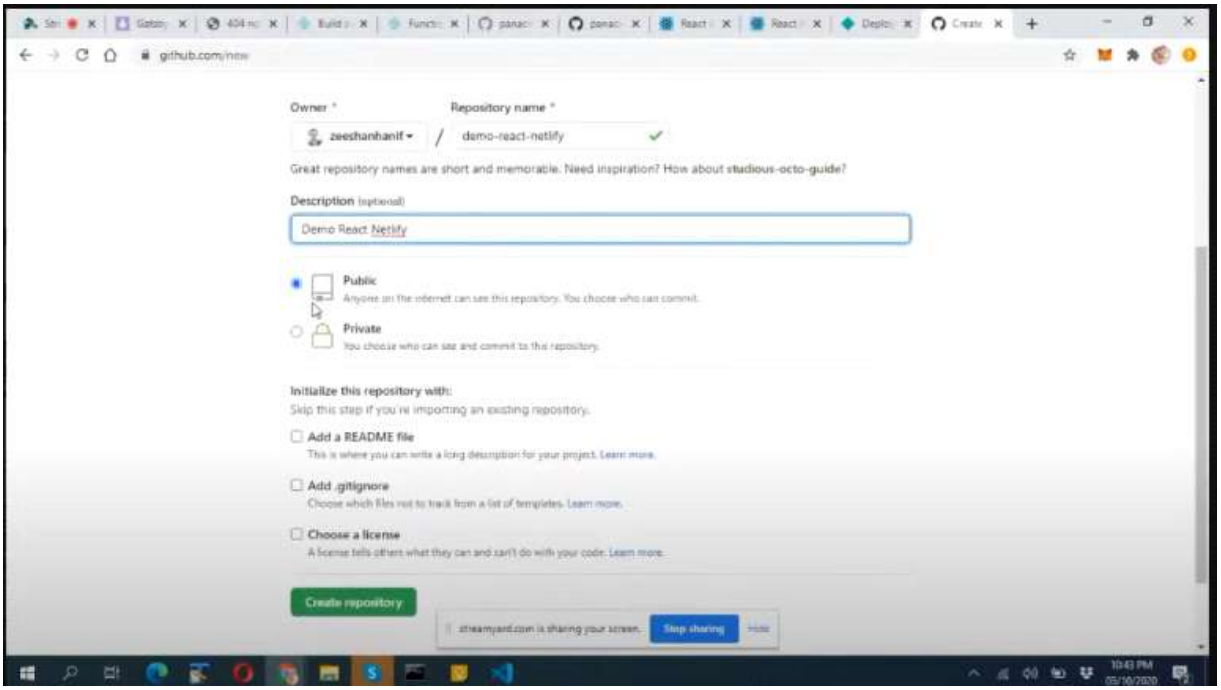
- Then add netlify login in react app



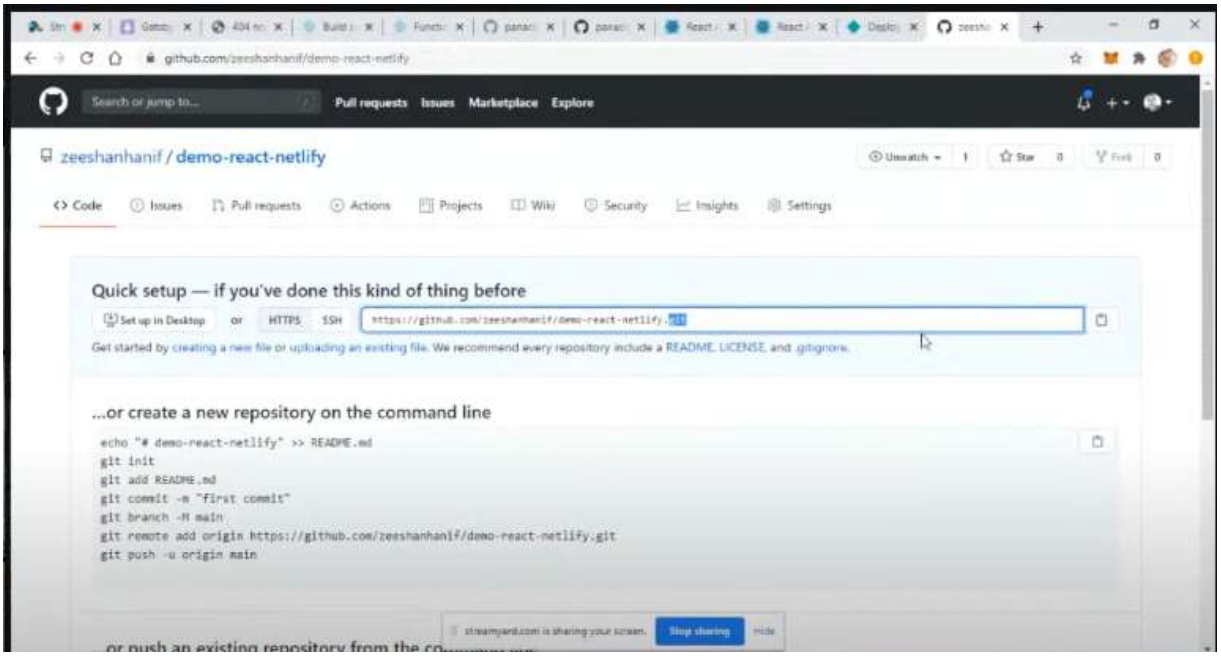
- Add netlify deploy



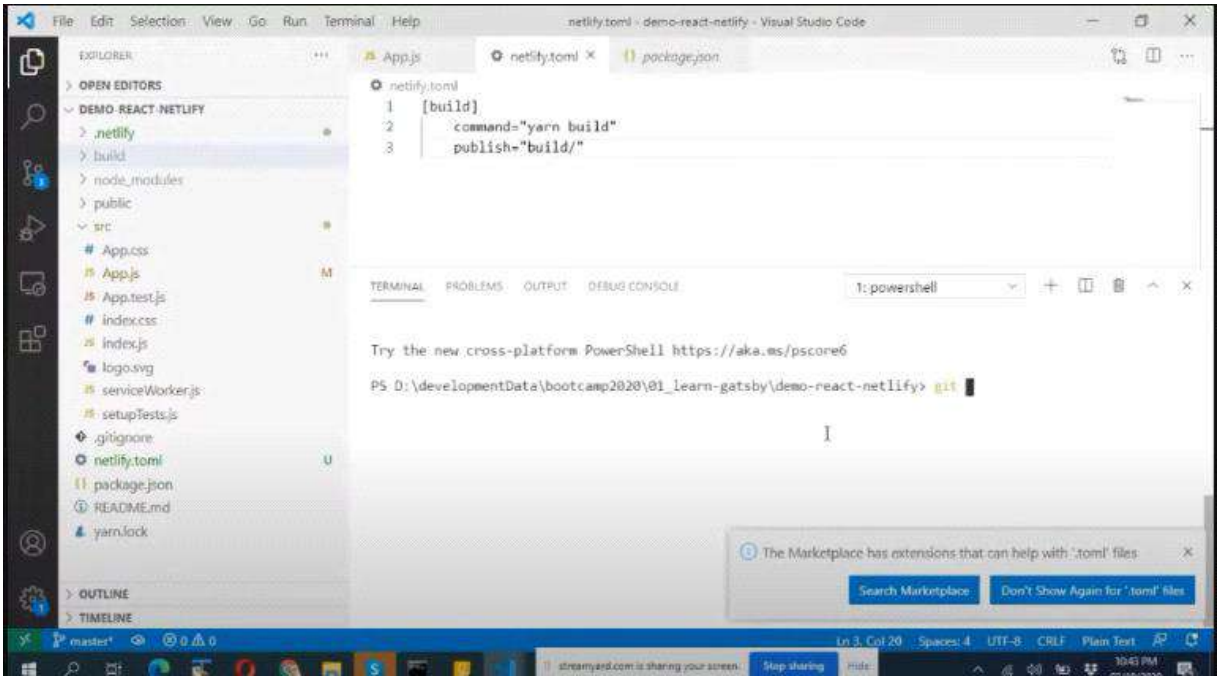
- Go to site and you can see the deployed site
- Go to Git hub
- Create a respository according to your site title



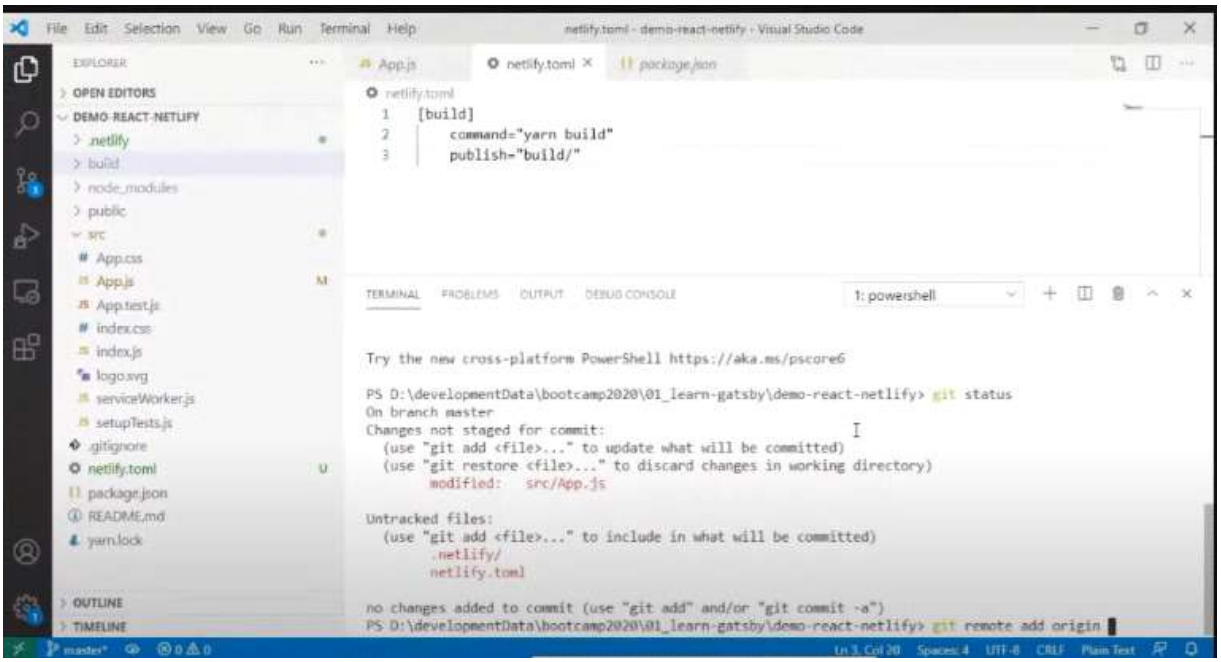
- Copy the Url



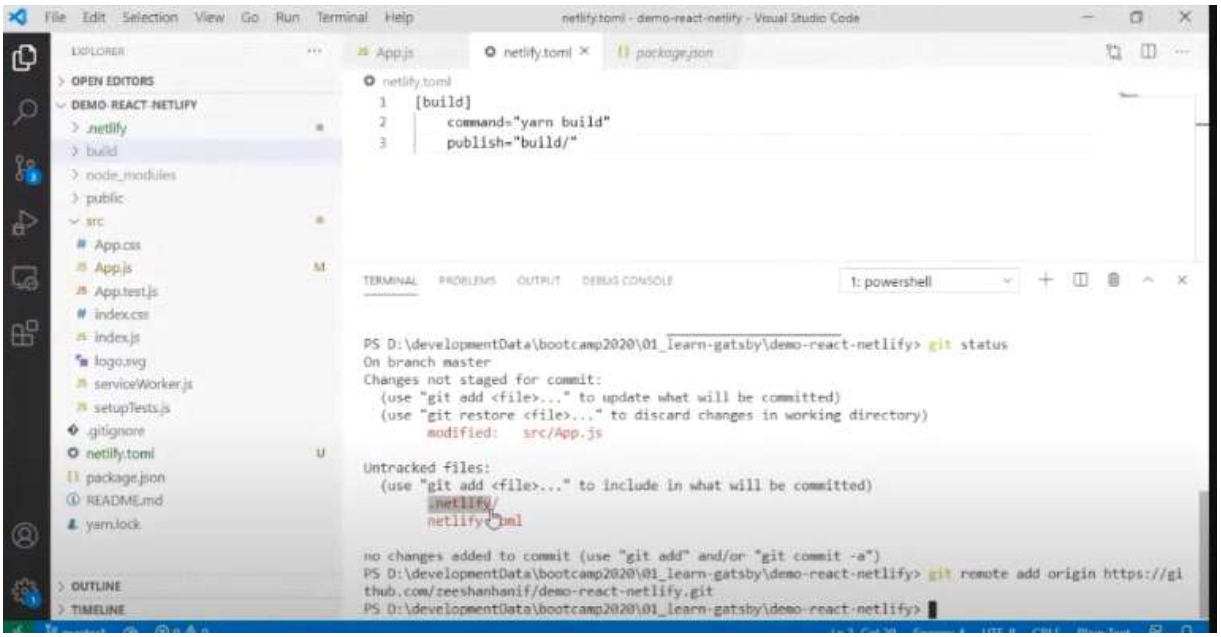
- Paste it in netlify.toml terminal



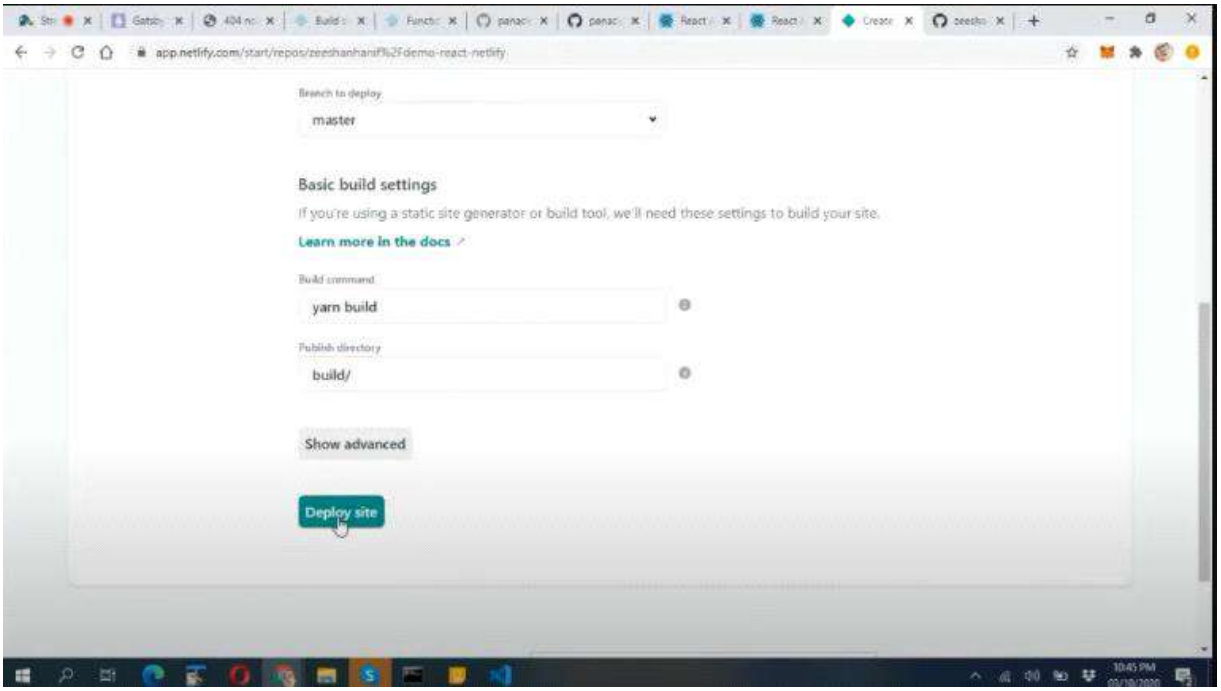
- Now add git remote add origin



- Copy .netlify/



- Paste it in gitignore
- Then add git push origin
- Go to Netlify site
- Then go to create a new site and search your site
- It will give you the yarn build and build command
- Then deploy the site



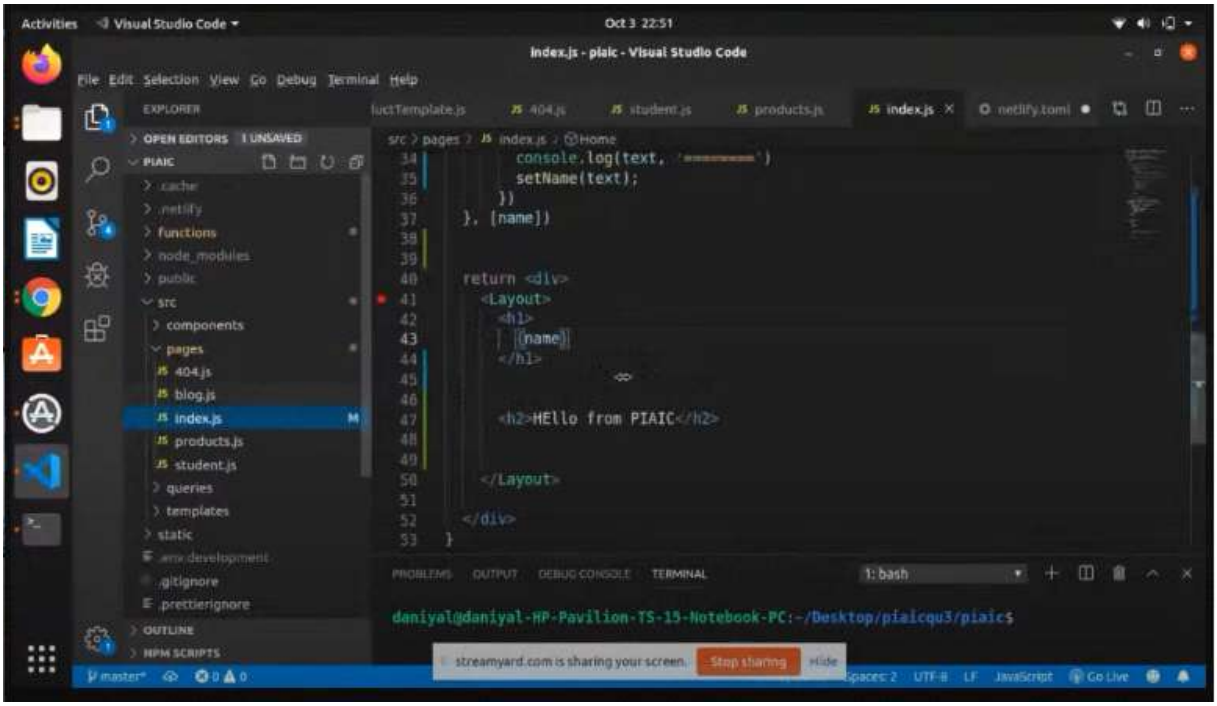
Step 15

Gatsby Develop

- Go to Netlify.toml and change the publish name

```
[build]
        command="yarn build"
        publish= "public"
```

- Then go to index.js

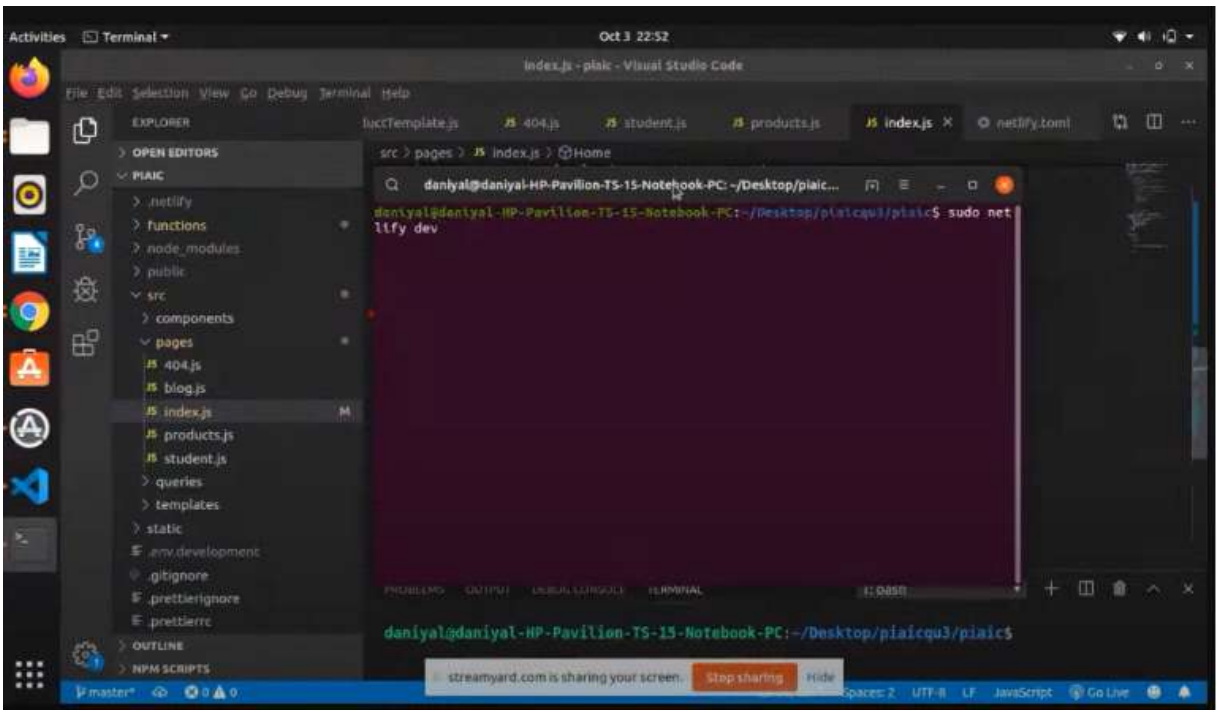


- Just change the content like this

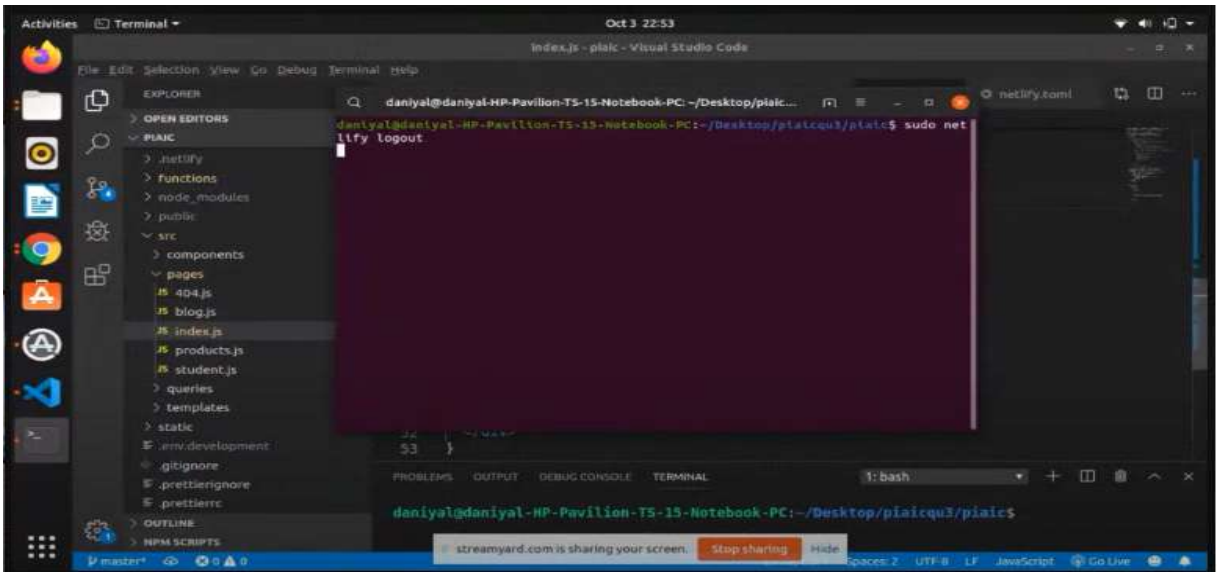
```
Return <div>
 <layout>
   {/* <h1>
      {name}
```

```
 </hq>  */}
<h2> Hello from PIAIC, danyal nagori</h2>

  </layout>
 </div>
}
```

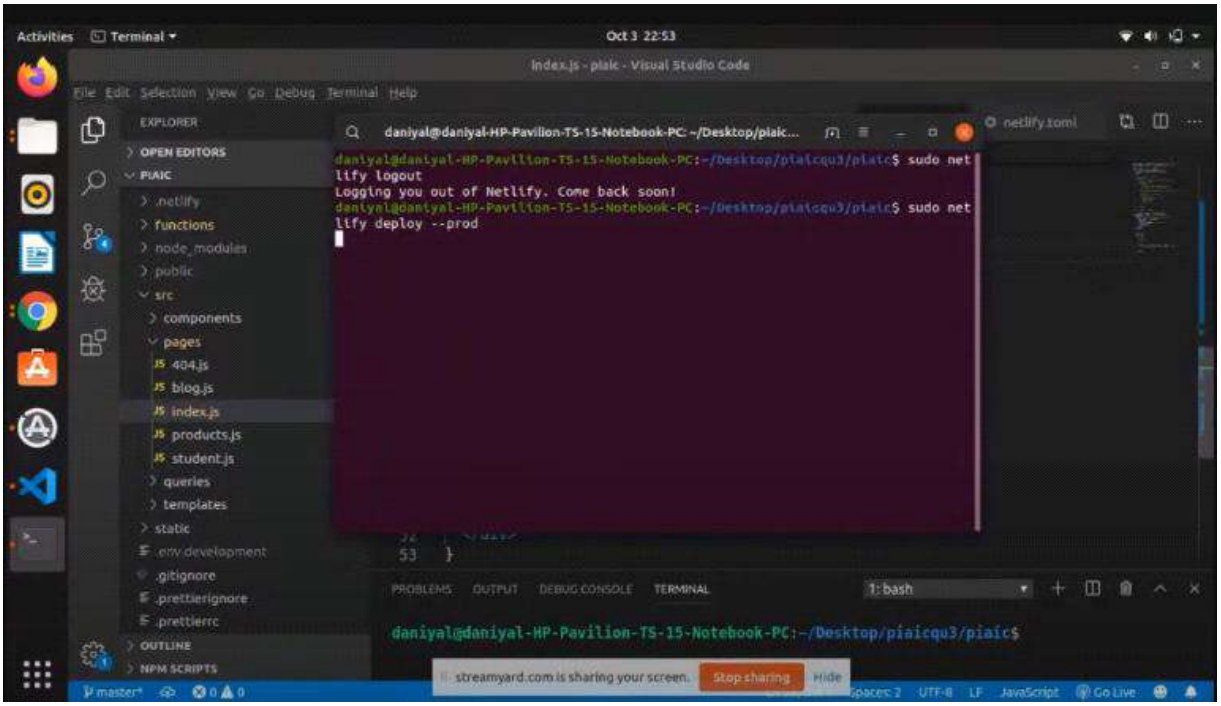
- You have to run the local command netlify dev



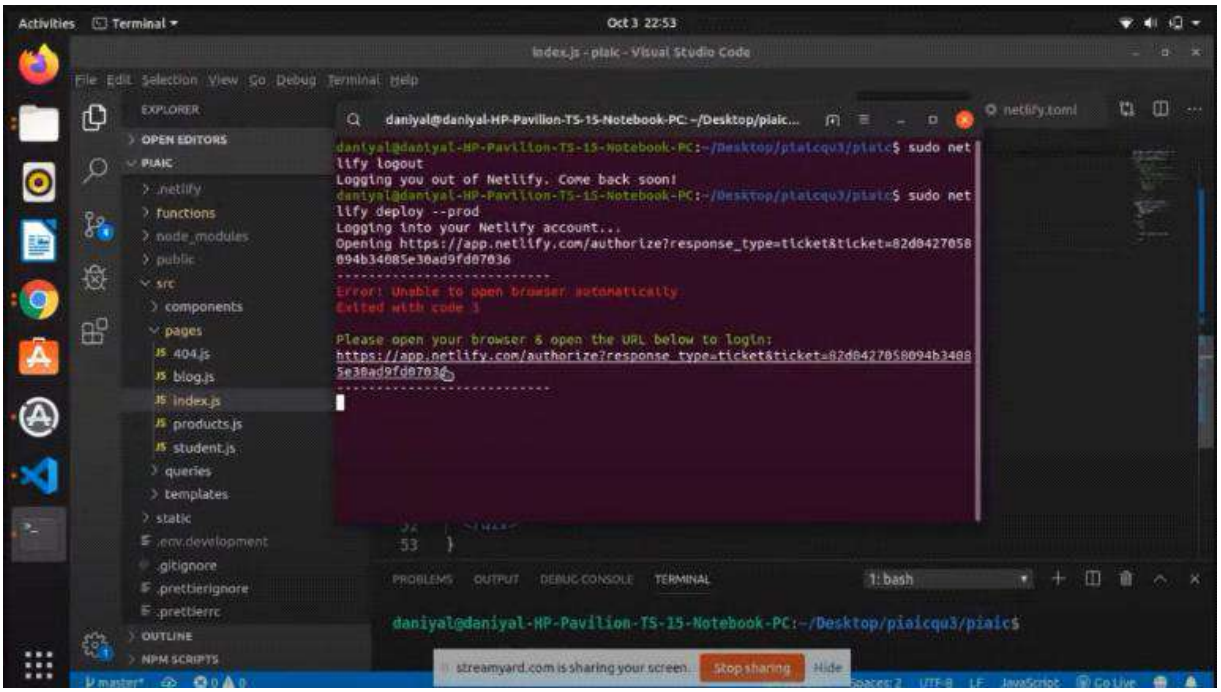
- Instead of netlify dev you have to run the
- Then run the command sudo netlify logout



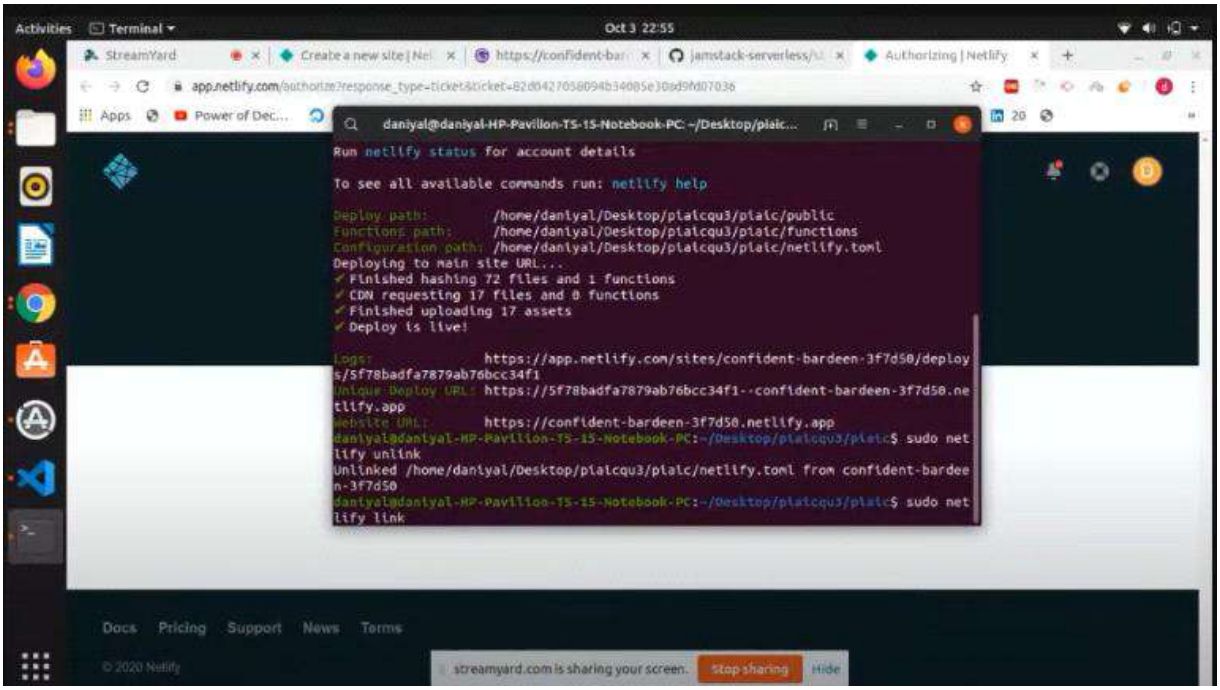
- Then run the command studo netlify develop



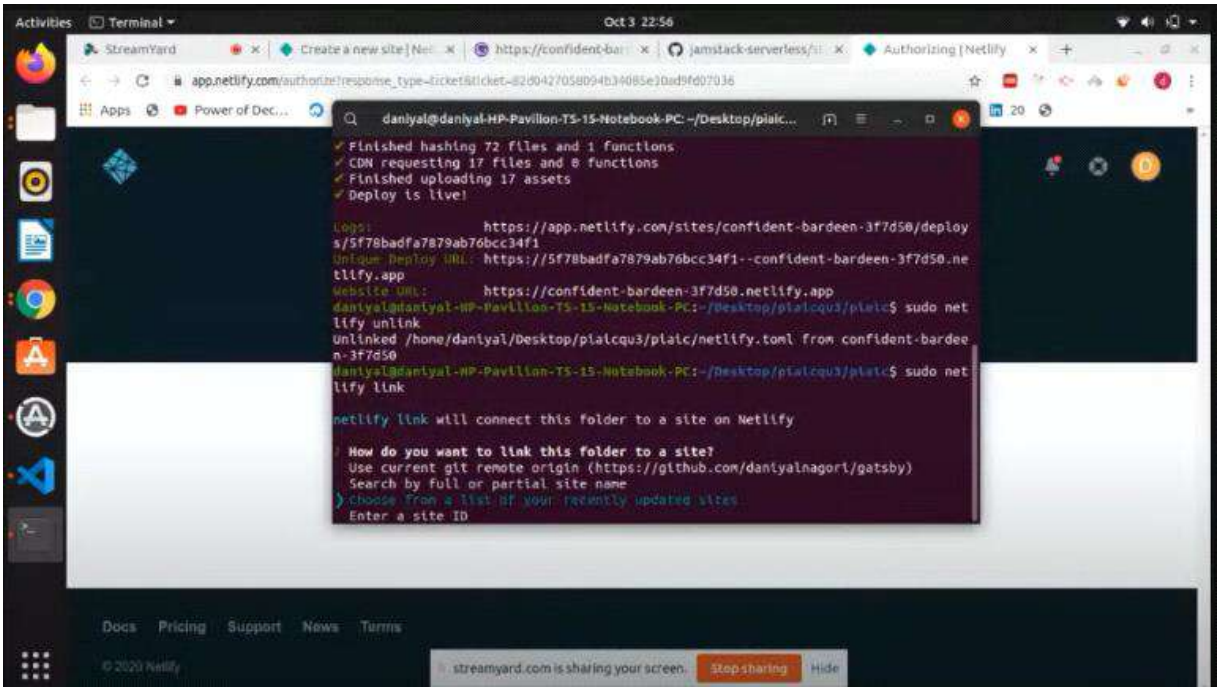
- Then copy this link



- Then run the command studio netlify link



- Select from the list

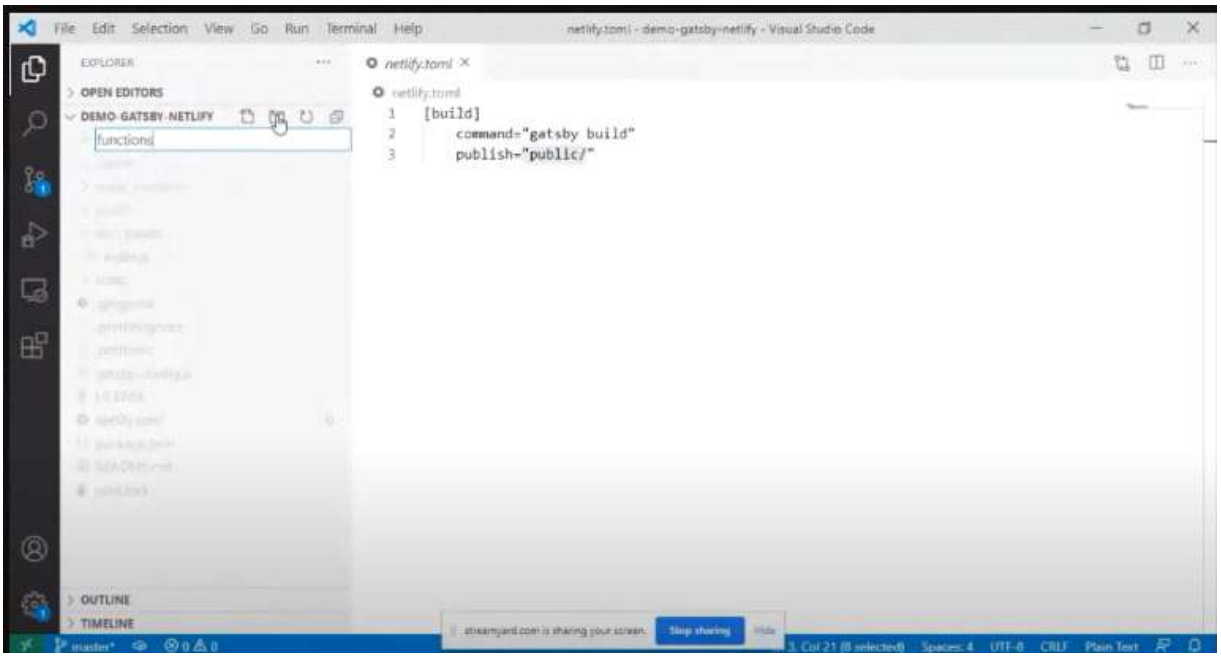


- Select one of your site in Netlify
- Then it will link from your Netlify account
- Then you have to add sudo netlify develop in the cli
- Then go to the site that you select in netlify command

Step 16

React Netlify Functions Helloworld

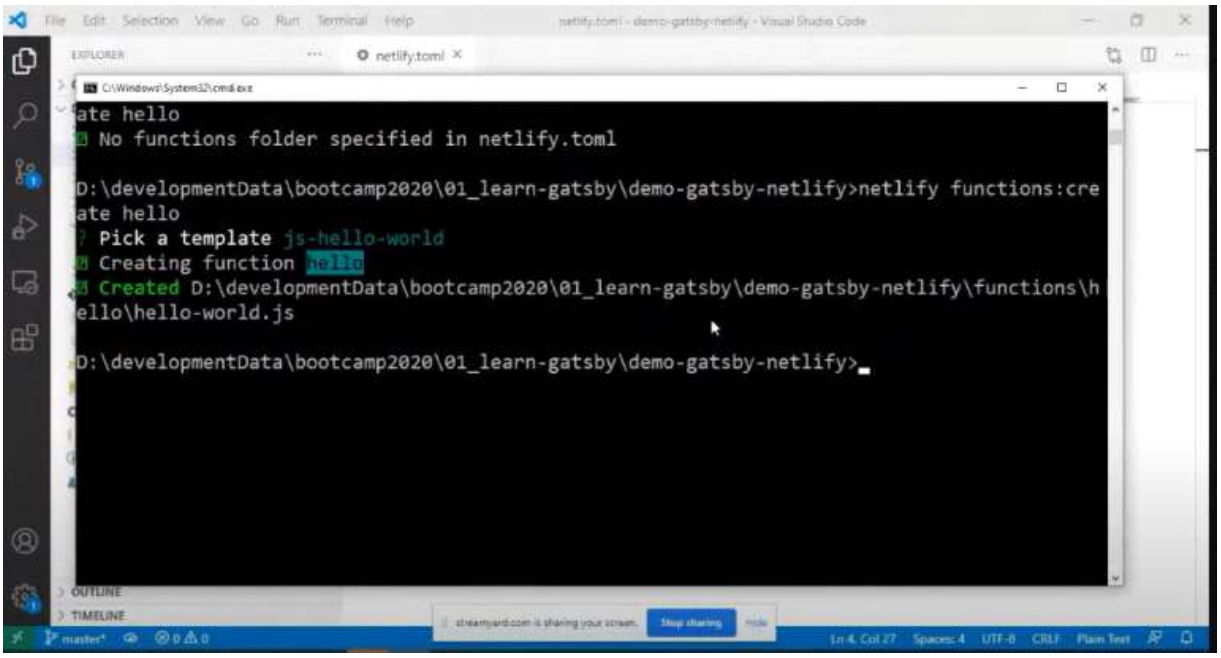
- First add the folder name “Functions”



- Then add Netlify functions: create hello
- Then go to netlify.toml and add the function

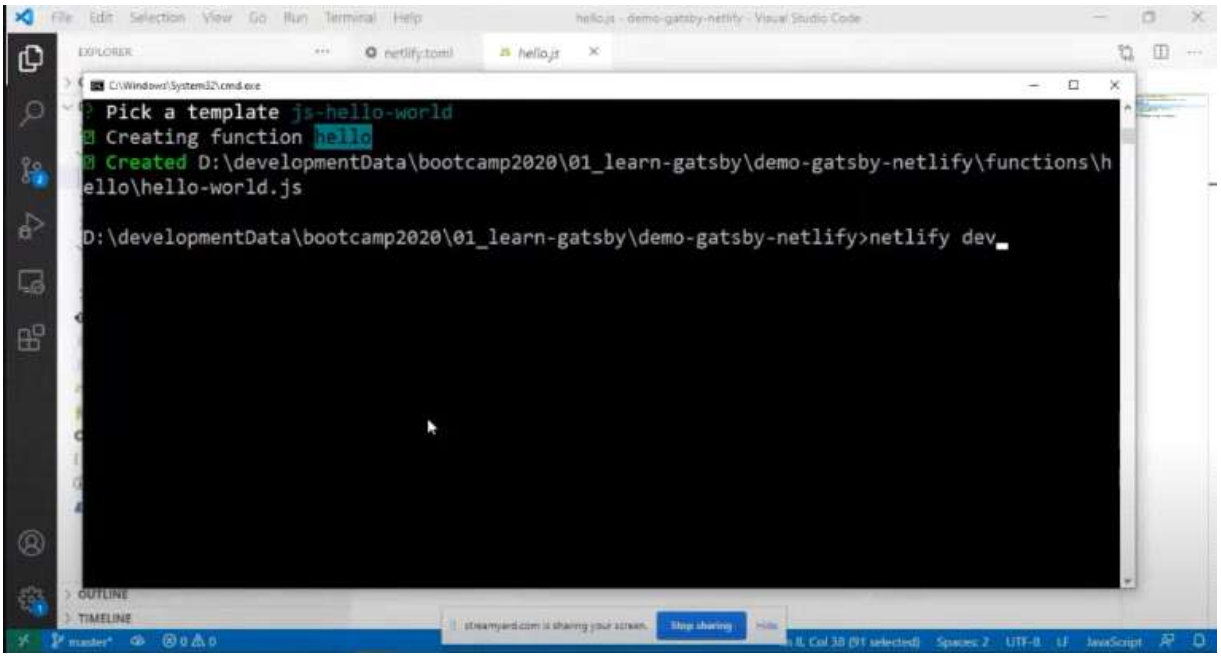
```
[build]
        command="yarn build"
        publish= "public/"
        functions:"function/"
```

- Then again add netlify functions: create hello to run the command
- Select the hello word



- It will create the hello.js folder

- To deploy the server add the netlify dev

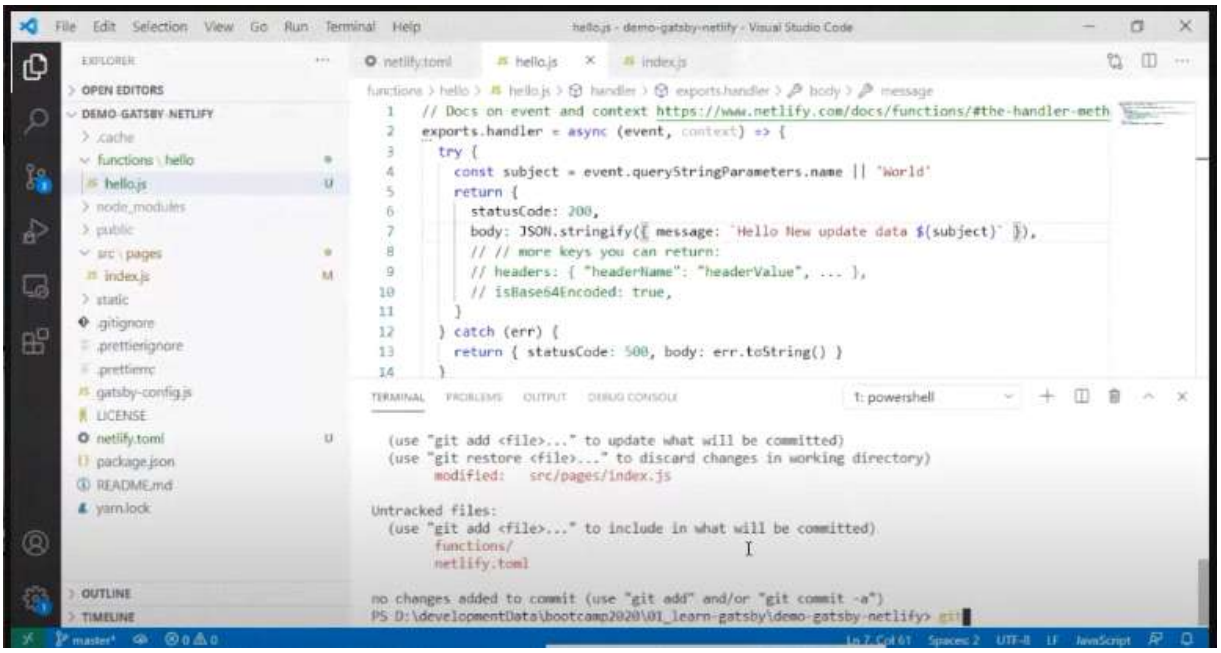


- When you hit the URL “localhost8888/netlify/functions/hello” you can see your development on the site
- If you change the code and add something like hello new change you can see this changing in localhost8888
- Then go to index.js and add some code in this format

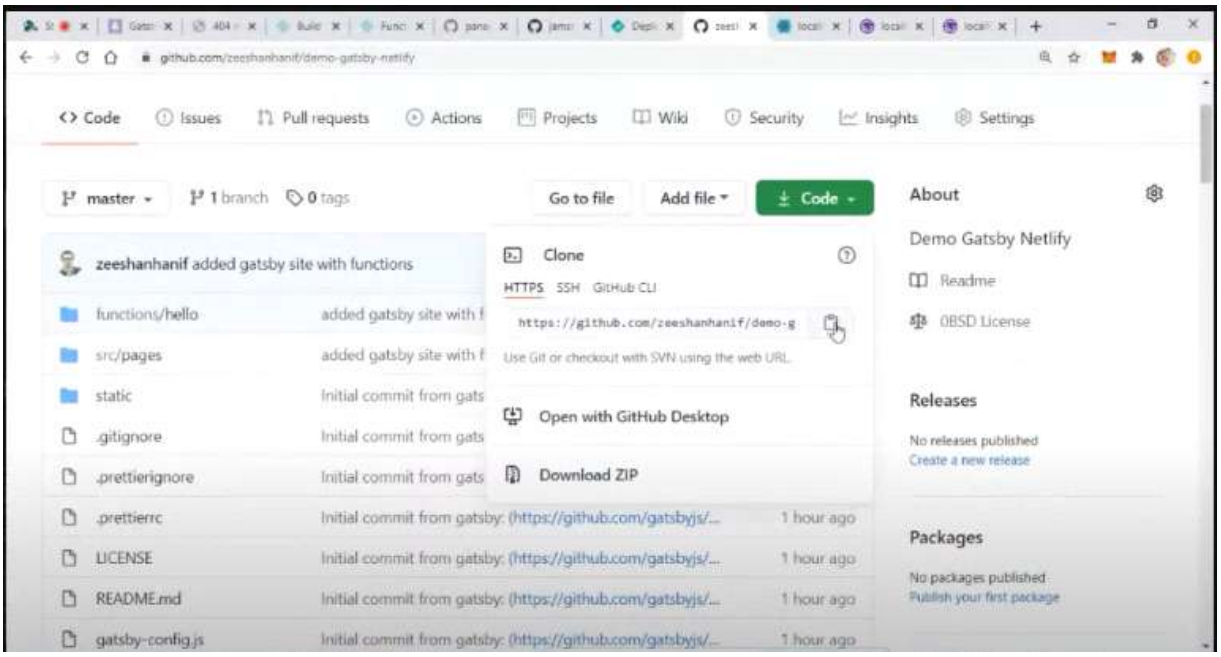
```
import React, ( useEffect, useState ) from "react"

export default functio Home  () {
```

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

 useEffect(()=>{
   (async()=>(
        const response = await fetch
('.netlify/functions/hello');
        const tempData = await response.json();
        setData(temData);
}
)();
}};


return (<div>
  HelloWorld!
 <div>(data.message)</div>
 </div>)

}
```

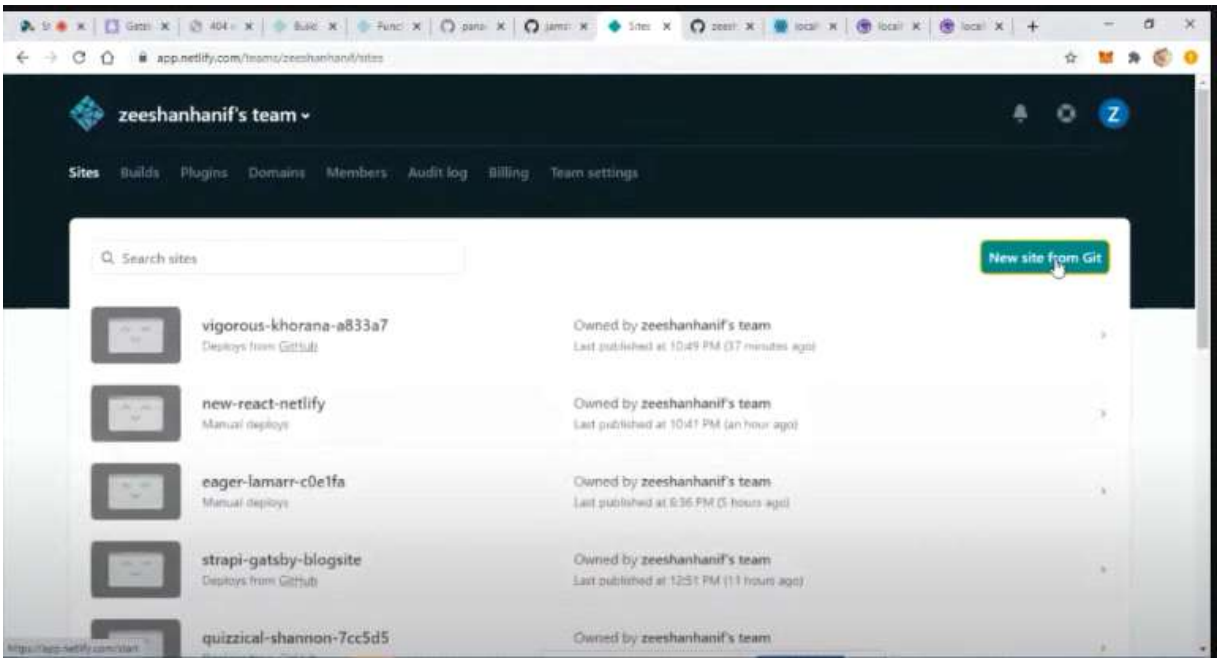
- Then go to “localhost8888” and check your results
- Go to repository and create a repository, if you already created then use this
- Go to hello.js terminal and run the command



- Go to repository and copy the code



- Then go to Netlify site and create a new site from Git



- Deploy the site
- After deploy just go to the link and check your results
- Go to index.js and add the code that help you not to face any problems

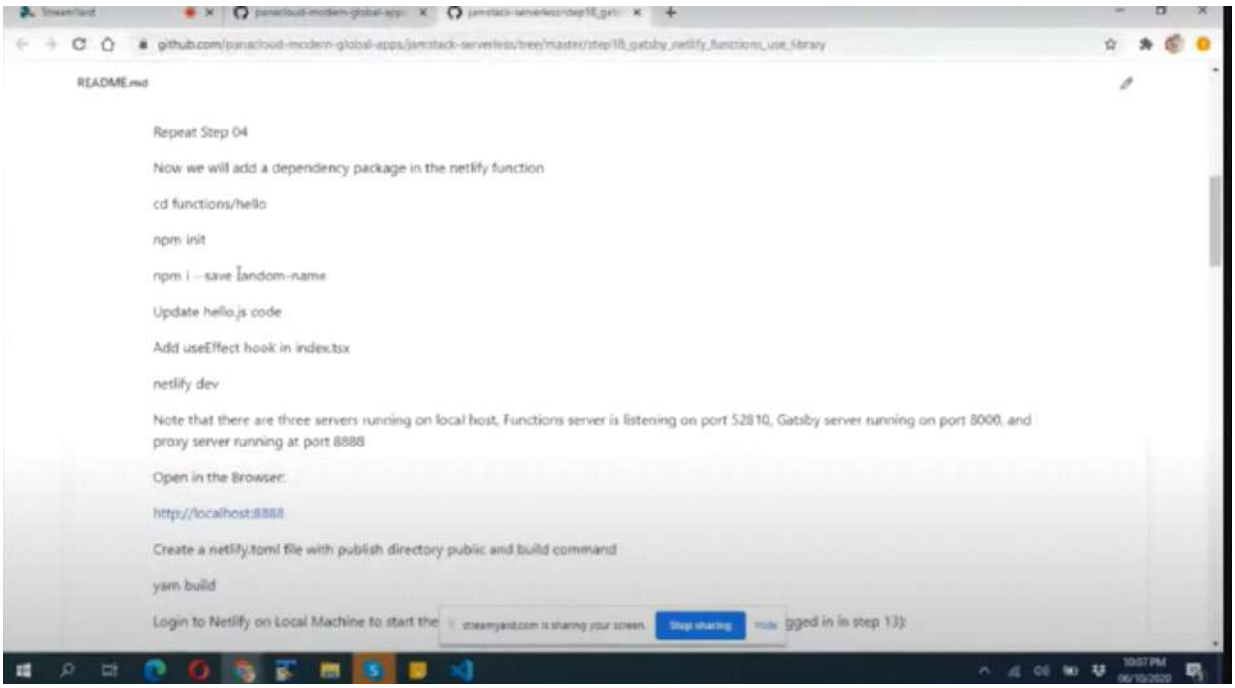
Session-36

Bootcamp

- We can create function with netlifier and those functions will be deployed with aws lambda and the good thing about the netlifer function is that they are server less so there is no limitation and scalability

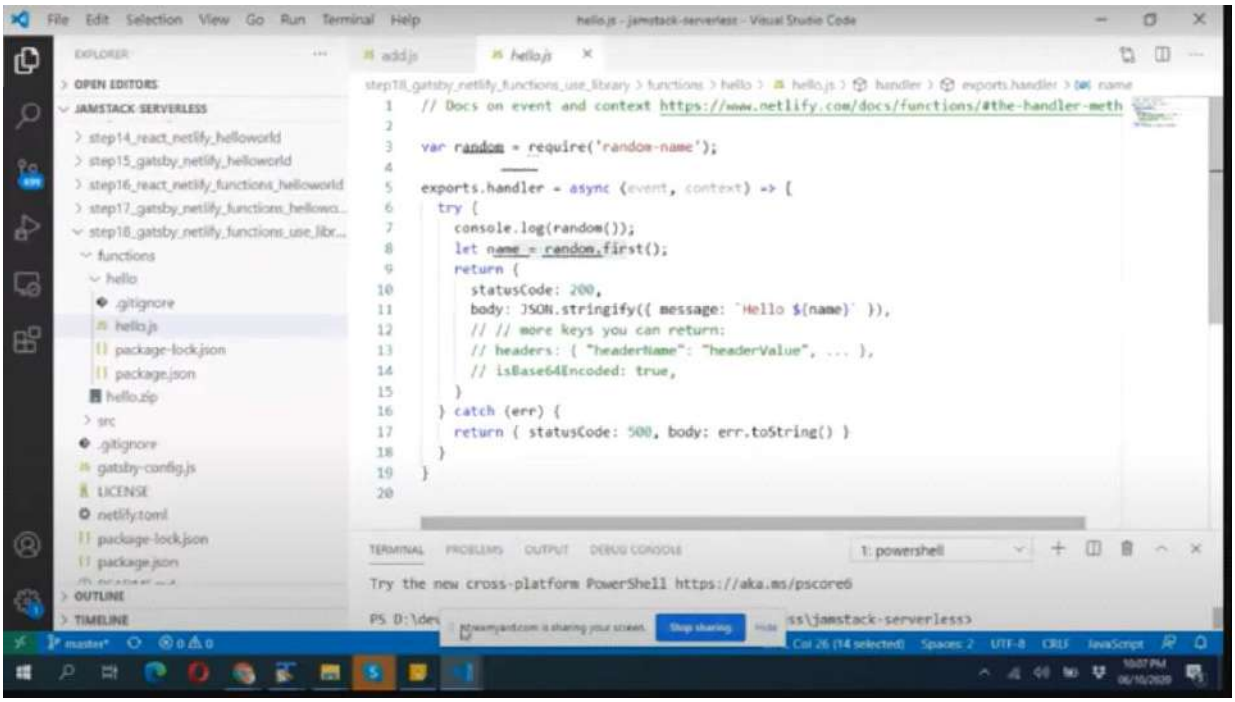
Step 18

- There is an additional one thing that in function as its a server set function so we have to use other libraries. If you want to use other libraries then how will be using that in a server as a function.

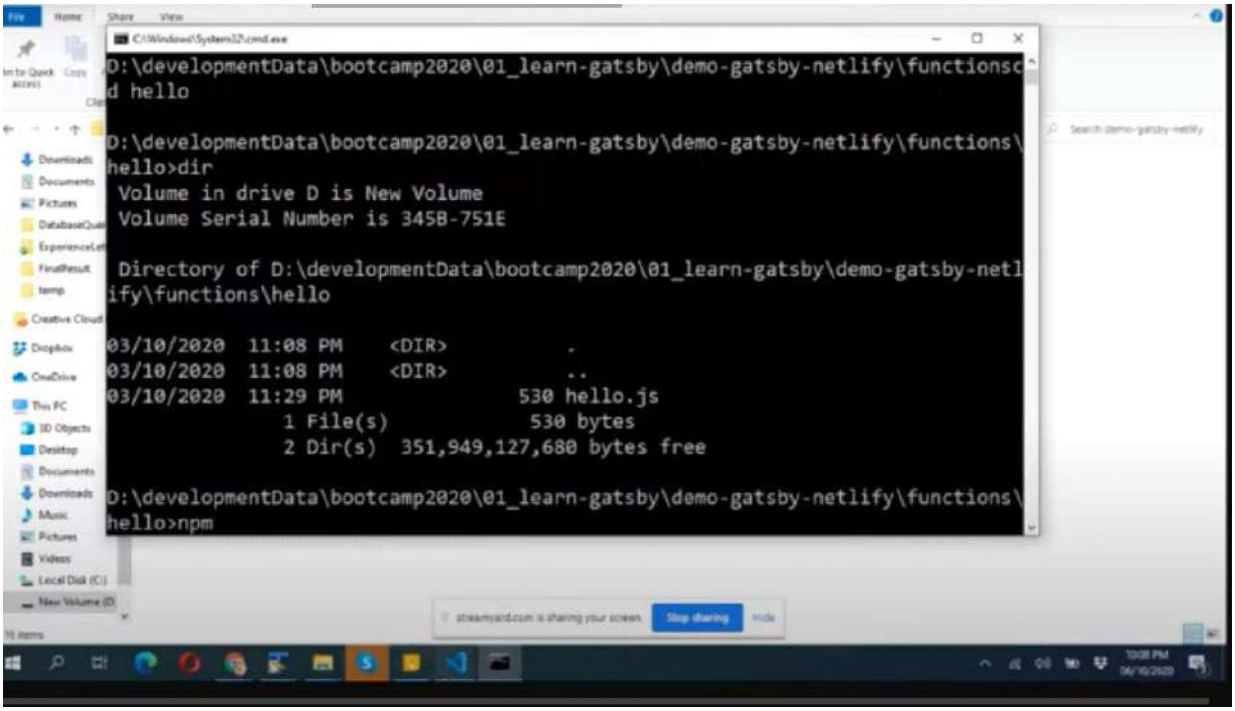


- We just want to generate the random name for a message to respond to the customer or a client.

- We import the library and then we install the library and then we call the required and after the require we call the random library to generate any random name and then we will send the name to the client.



- Use the previous code. So further go to command home and call the function



- We need to understand is the folder we have hello.js is not an npm project.to install any library and keep in packet json as we do with getsby.js same we need to do with the server site code so we must have an packet .json file to have an json file so we can create directly or we can npm command and it will generate packet json for us.so we call npm init.

```
D:\developmentData\bootcamp2020\01_learn-gatsby\demo-gatsby-netlify\functionsc
d hello

D:\developmentData\bootcamp2020\01_learn-gatsby\demo-gatsby-netlify\functions\
hello>dir
 Volume in drive D is New Volume
 Volume Serial Number is 345B-751E

 Directory of D:\developmentData\bootcamp2020\01_learn-gatsby\demo-gatsby-netl
ify\functions\hello

03/10/2020  11:08 PM    <DIR>          .
03/10/2020  11:08 PM    <DIR>          ..
03/10/2020  11:29 PM               530 hello.js
               1 File(s)            530 bytes
               2 Dir(s)  351,949,127,680 bytes free

D:\developmentData\bootcamp2020\01_learn-gatsby\demo-gatsby-netlify\functions\
hello>npm init
```

- So you can see it can generate packet.json file.

```
"name" : "hello"
"version": "1.0.0",
"description"; " ",
"mian": "hello.js"
  Debug
"scripts": {
   "test": "echo/ "Error: no test specfind/" && exit 1"
},
"author": " ",
"license": "ISC"
}
```

- Any library with the save command it will appear in dependency just like we did it get we have seen we have a dependency like we have here. So same will be there for this case.
- Install npm

```
{
  "name": "hello",
  "version": "1.0.0",
  "description": "",
  "main": "hello.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}


Is this OK? (yes) yes

D:\developmentData\bootcamp2020\01_learn-gatsby\demo-gatsby-netlify\functions\
hello>npm install --save random-name
```

- After installation it shows all modules.

- Now run the command

```
C:\Windows\System32\cmd.exe - netlify dev
Microsoft Windows [Version 10.0.18363.1082]
(c) 2019 Microsoft Corporation. All rights reserved.

D:\developmentData\bootcamp2020\01_learn-gatsby\demo-gatsby-netlify>netlify de
v
◈ Netlify Dev ◈
◈ Functions server is listening on 50016
◈ Starting Netlify Dev with gatsby
yarn run v1.13.0
$ gatsby develop

    ◈ Server now ready on http://localhost:8888
```

- Go to "localhost8888" and check your results

How to add multiple libraries?

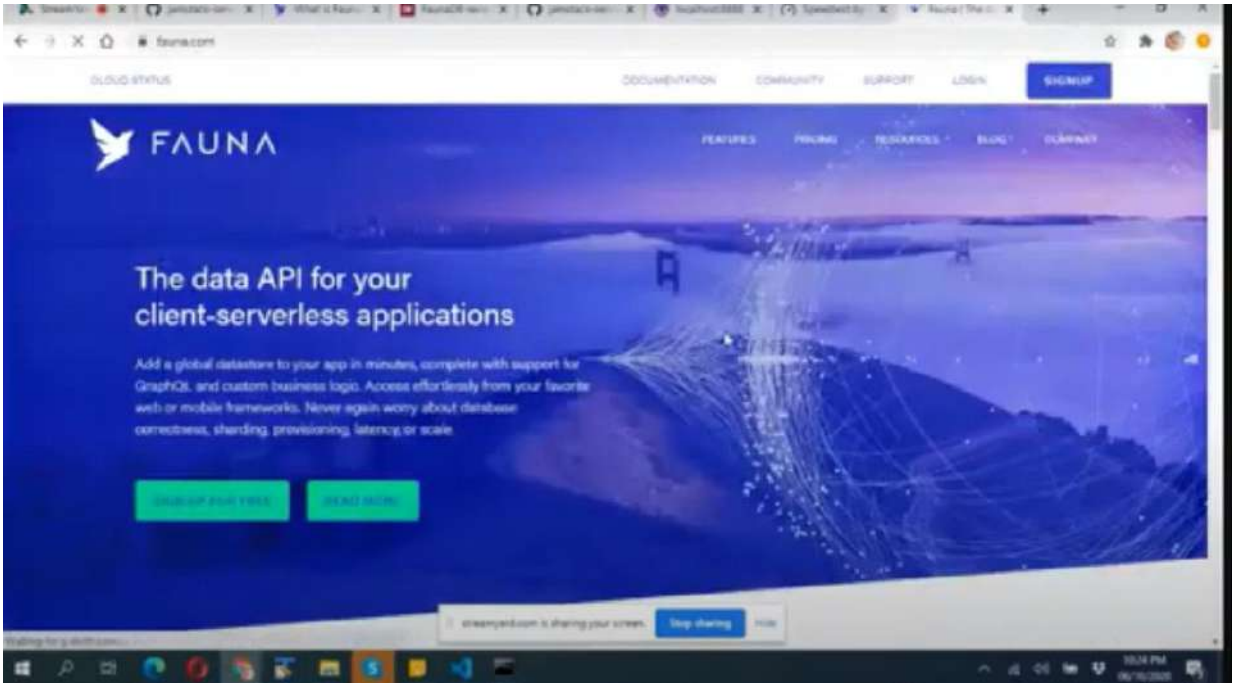
With their function show a function requires to use some other library any any other repository on library or utility for which we need to use inside a function to for that will be using package JSON show will start npm init function will become function folder will become what you say npm project and then we can whatever laboratory want to use it will install it and then we can call the library is it just like we will have done with the any client library similar to that for the server as well only difference is it will be running in aws lambda.

Step 19

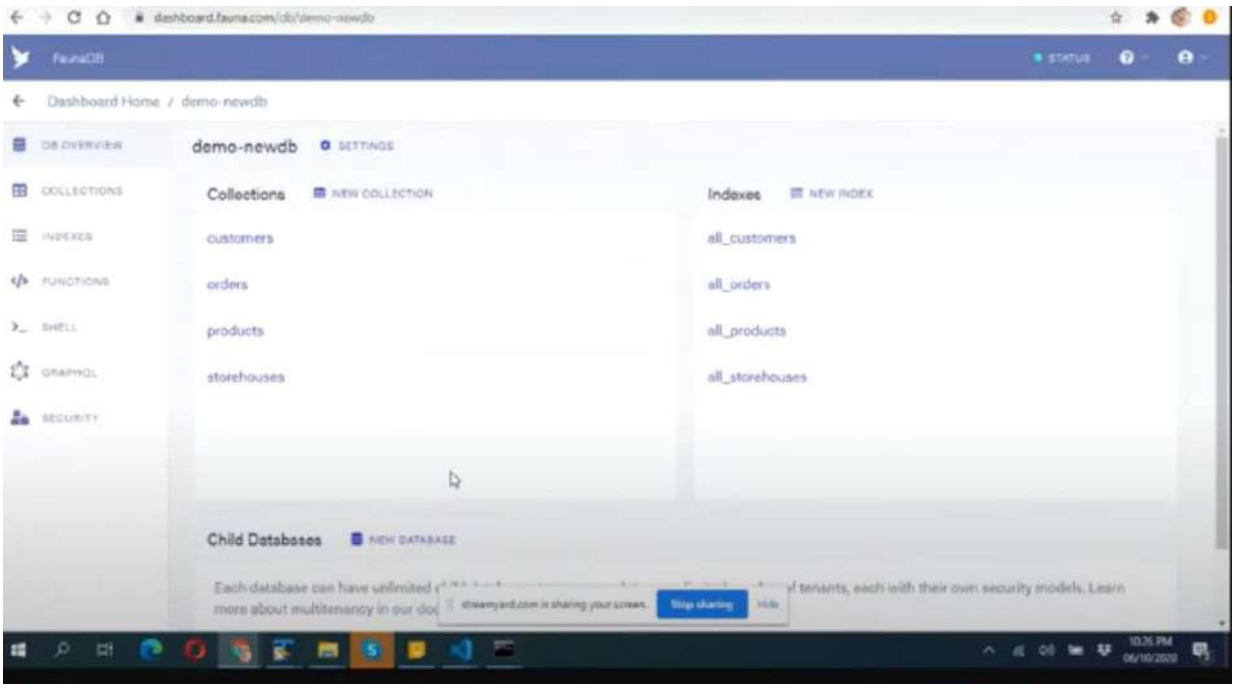
FAUNADB

What is FaunaDb?

FaunaDB is a cloud-native OLTP database with distributed ACID transactions using the Calvin protocol. It includes support for enterprise features like configurable data retention and hierarchal multi-tenancy. FaunaDB is available as a managed cloud service or as a downloadable JAR, machine image, or container.



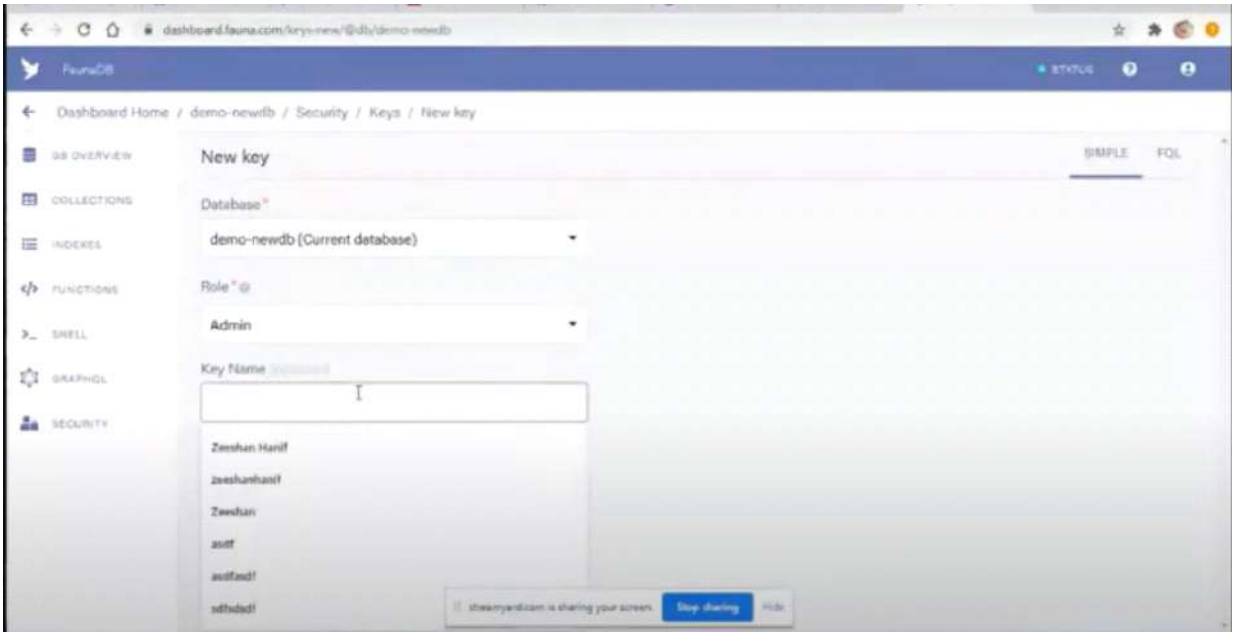
- Use MangoDb



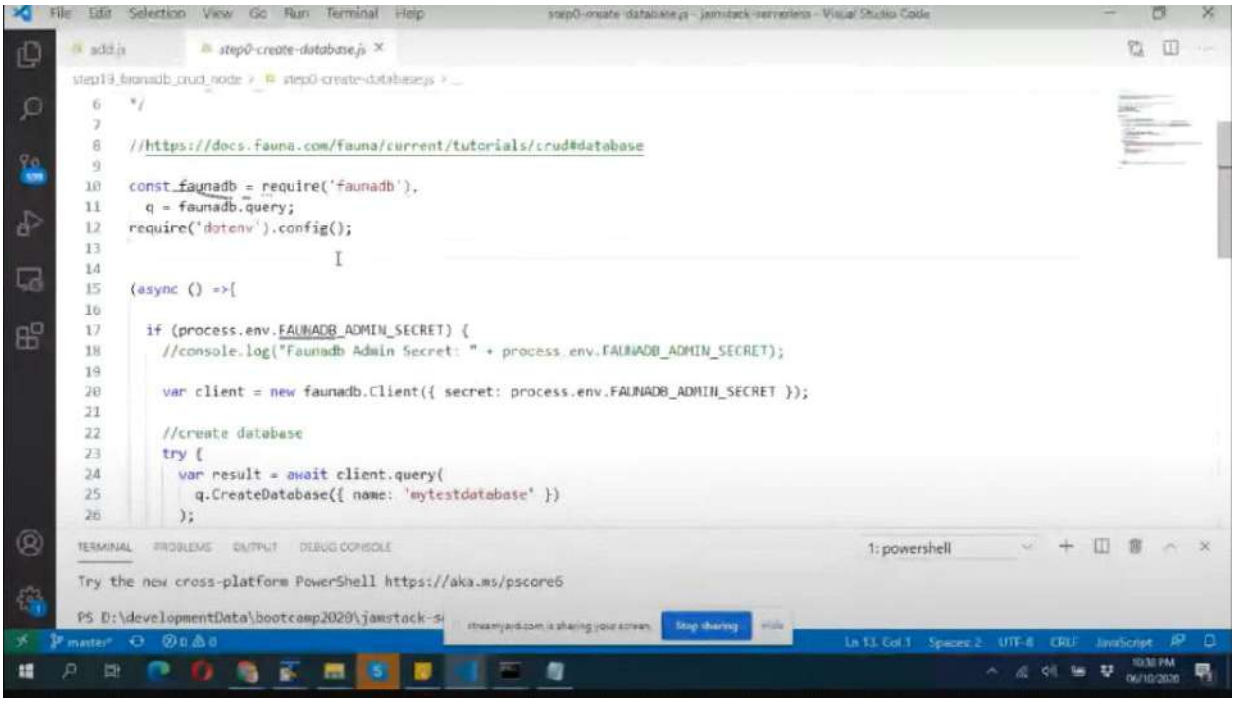
What is interface ?

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

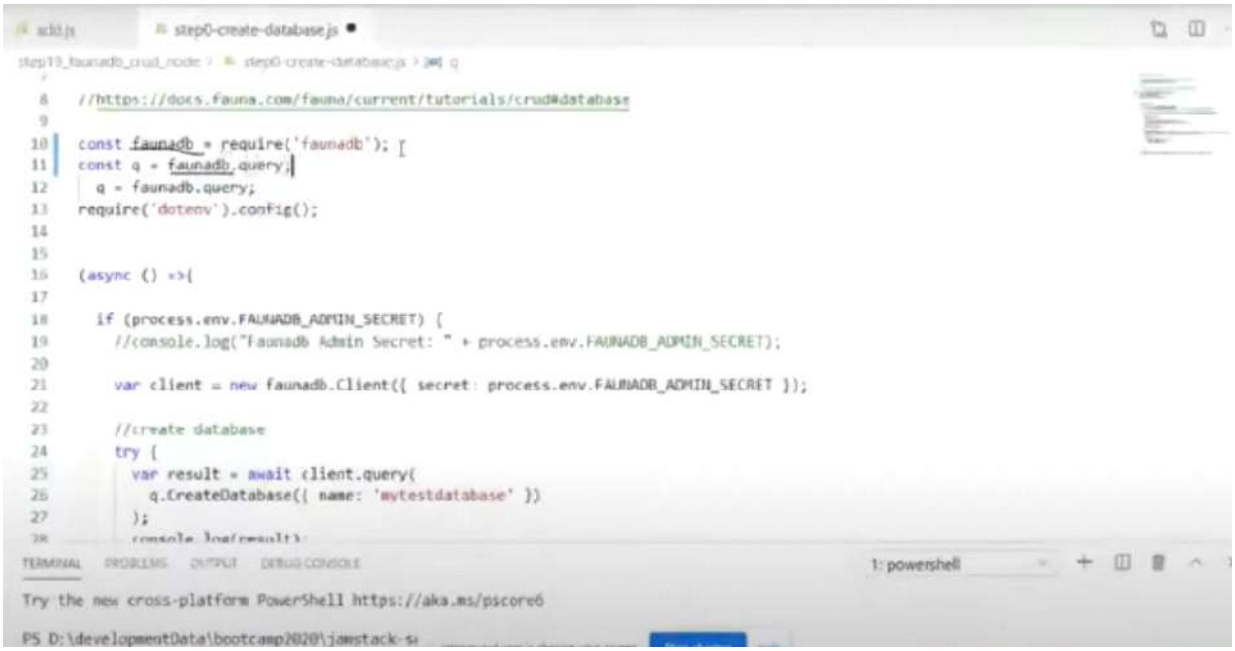
Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.



- Copy the created key
- Instsall FaunaDb and check the reults
- Add async function



- Const faunaDB = require and then we write const q = faunaDB. Query;



- Create a Database

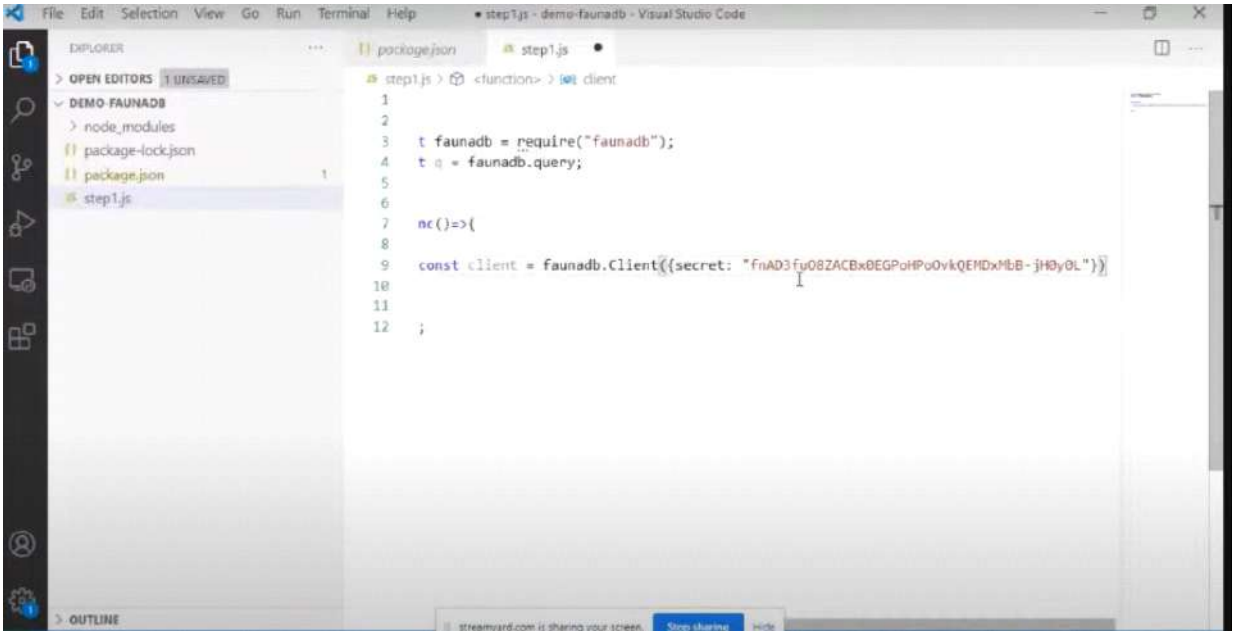
- Create FaunaDB

```
“name” : “hello”
“version”: “1.0.0”,
“description”; “ “,
“mian”: “hello.js”
  Debug
“scripts”: {
   “test”: “echo/ “Error: no test specfind/” && exit 1”
},
“author”: “ “,
“license”: “ISC”
“dependicies”: {
“dotenv”: “8.2.0:,
“faunadv”: “3.0.1”
}
```

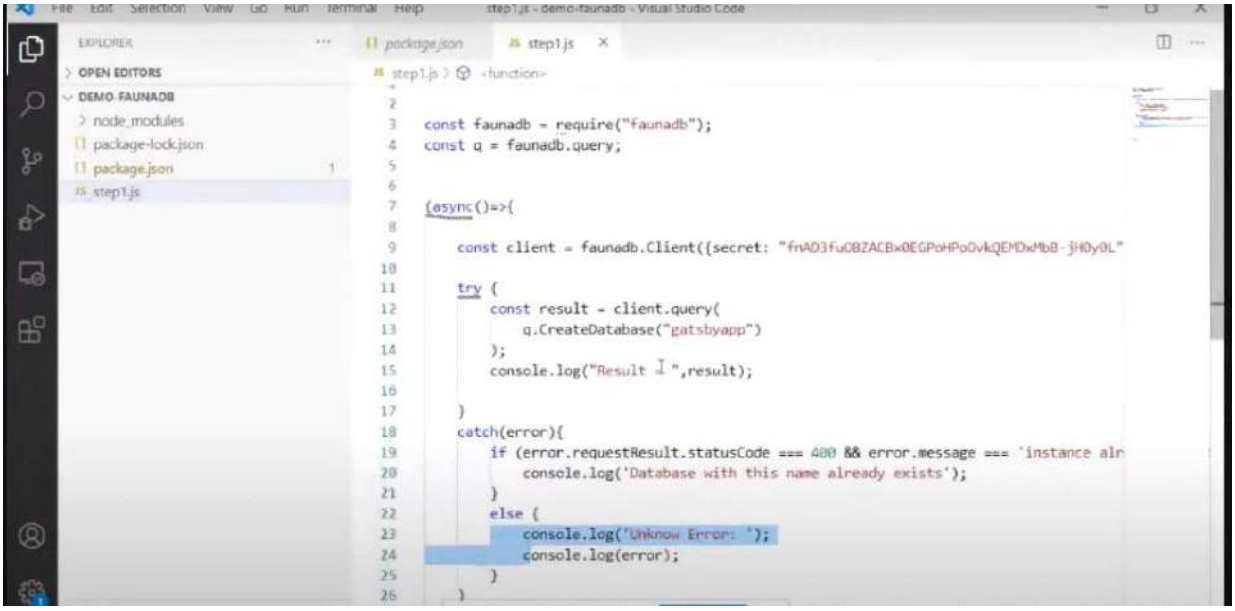
So this is basically package.JSON file and it has. Envy and faunaDB as a library and then I will create a JavaScript file search in simple file it will call step 1 or anything inside that what I need to do i need to require the faunaDB.

```
Const faunadb = require ( “faunadb”);
Const = faunadb.querry;
```

We need to call the await so we call the async function. Its a self calling function.
First we create a client and then pass the secret. Now i will pass the static value directly hard coded value later we will change it.



The client what we need to do we need to do all the query and whatever will be doing it will return as a result. What why we need try catch because if you are using async await instead of promises if there isn't any error it will crash your it will throw error so you need to use it try catch for that if you are using async await.



Once you are done as and mention that it's not a client-side application it's also not a server-side application is just a node.js code so what we need to do is i will just call it from the the terminal, so node step1.js.once i will do that i will hit that what it will do is it will execute this function.and everything inside it.

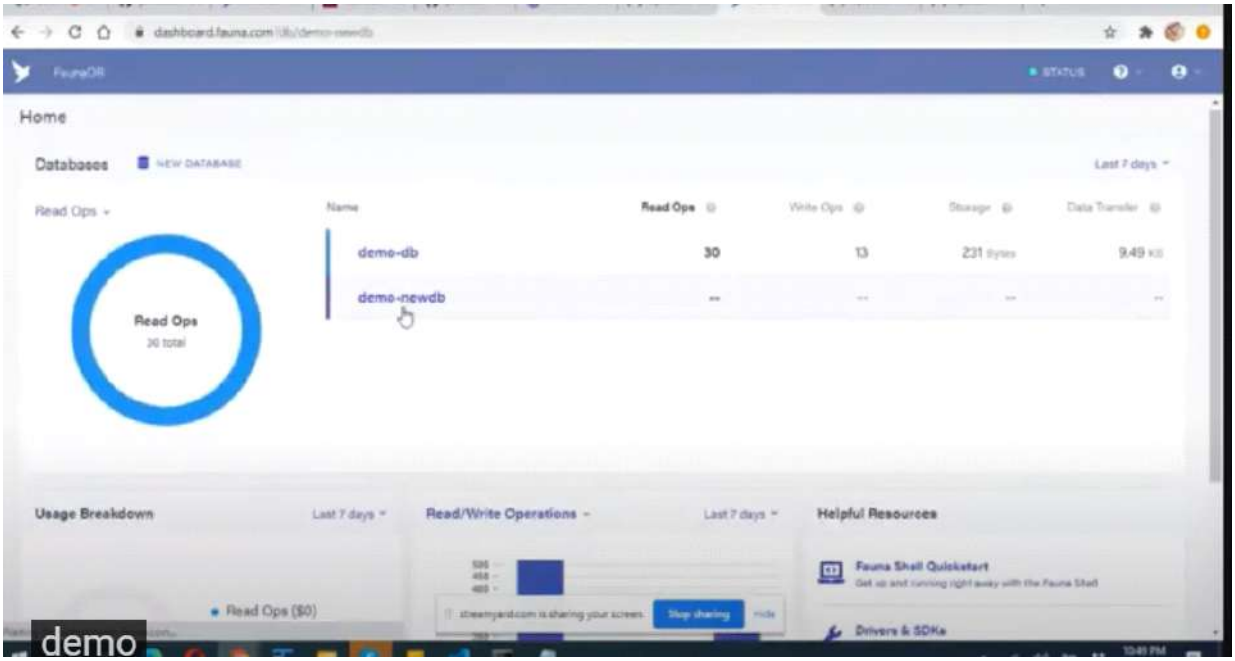
```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.18363.1082]
(c) 2019 Microsoft Corporation. All rights reserved.

D:\developmentData\bootcamp2020\jamstack-serverless\learn-funadb\demo-faunadb
D:\developmentData\bootcamp2020\jamstack-serverless\learn-funadb\demo-faunadb>node step1.js
```

```
File Edit Selection View Go Run Terminal Help        step1.js - demo-faunadb - Visual Studio Code
EXPLORER
> OPEN EDITORS
DEMO-FAUNADB
> node_modules
package-lock.json
package.json
step1.js
package.json   step1.js
step1.js > <function> > client > secret
const faunadb = require('faunadb');
const q = faunadb.query;

(async()=>{

    const client = faunadb.Client({secret: "fnAD3fu0gZACBx0EGPoHPoOvkQEMDxMbB-jH0yOL"

    try {
        const result = client.query(
            q.CreateDatabase("gatsbyapp")
        );
        console.log("Result = ",result);

    }
    catch(error){
        if (error.requestResult.statusCode === 400 && error.message === 'instance alr
            console.log('Database with this name already exists');
        }
        else {
            console.log('Unknow Error: ');
            console.log(error);
        }
    }
```

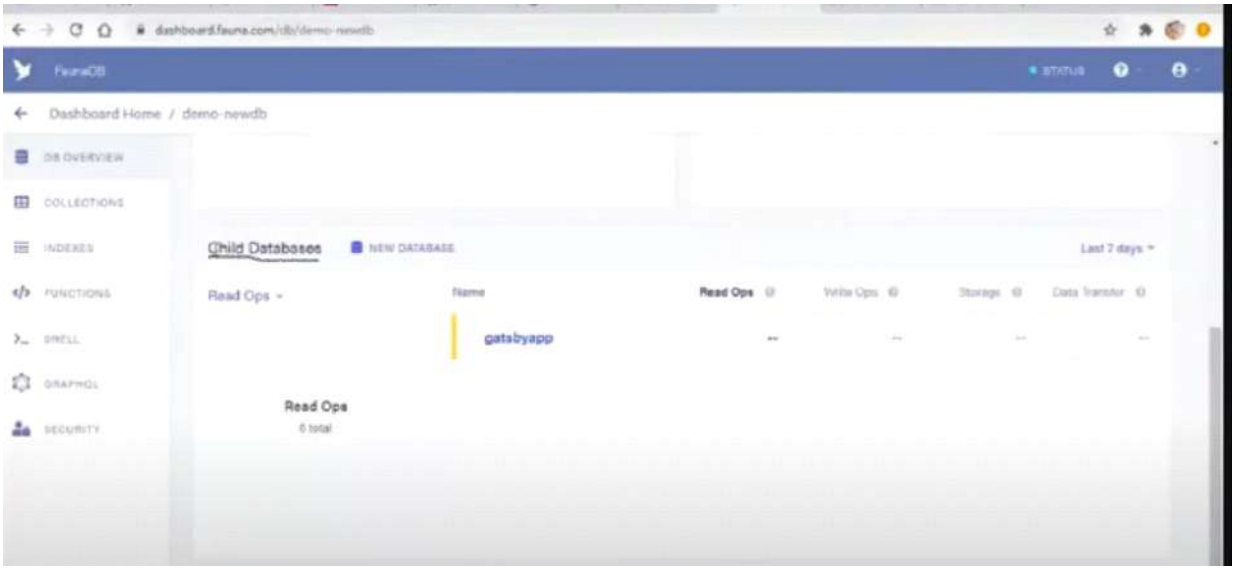
So to call the created database we need to pass an object and inside that object will be passing the key the name of the data base and then value.

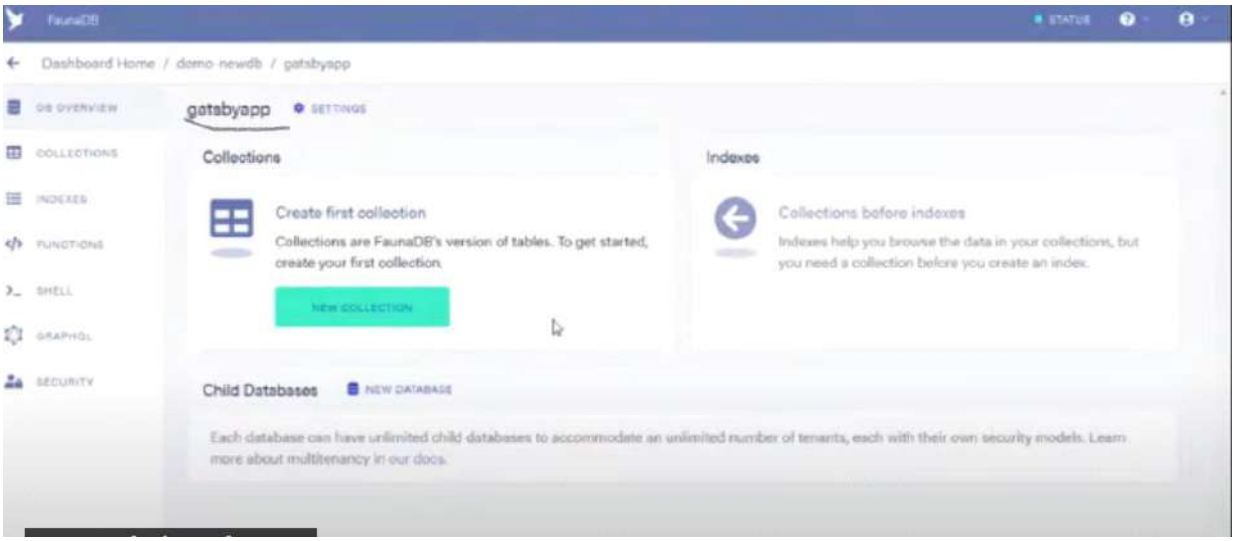
```
add.js   step0-create-database.js
C:\Windows\System32\cmd.exe
    'x-txn-retries': '0',
    'x-txn-time': '1602006528726679',
    'x-write-ops': '0'
  },
  startTime: 1602006526792,
  endTime: 1602006527755
 }
}

D:\developmentData\bootcamp2020\jamstack-serverless\learn-funadb\demo-faunadb>node step1.js
Result= {
  ref: Database("gatsbyapp"),
  ts: 1602006575190000,
  name: 'gatsbyapp',
  global_id: 'yxq93t7zaydyn'
}

D:\developmentData\bootcamp2020\jamstack-serverless\learn-funadb\demo-faunadb>_
```

Now it says results so its mean data base has been committed.and we can go to our database and see if we have this.



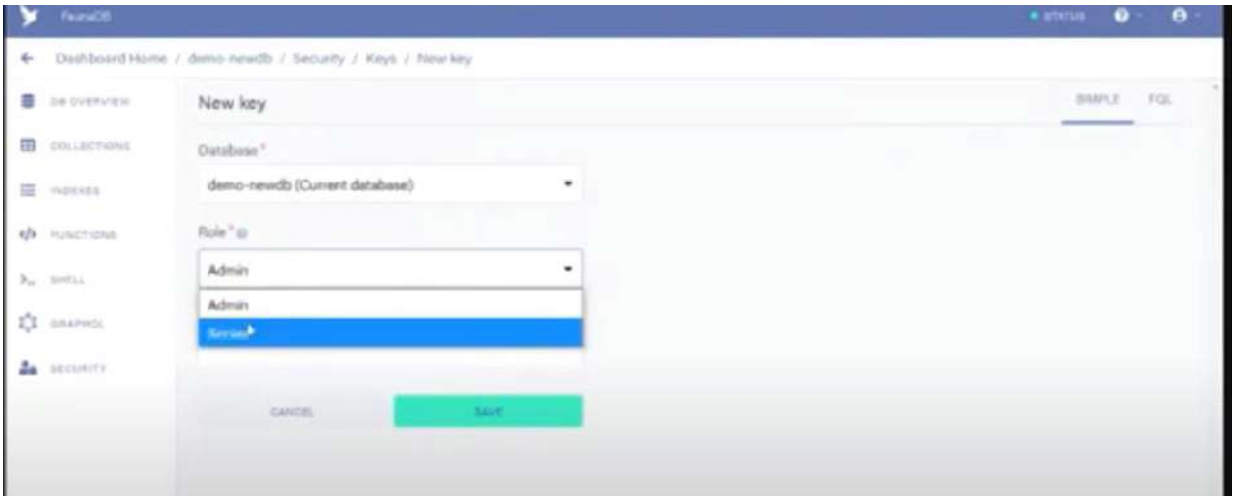
So the first step what i have done is i just call the client faunaDB. Client provided the secret. Once I have provided the secret it will request to the server and it will connect to the database and then we will call the client.query it will initiate the query and whatever query we want to do. At this moment we are creating a database and that we can call the get ,we call the data get collection.



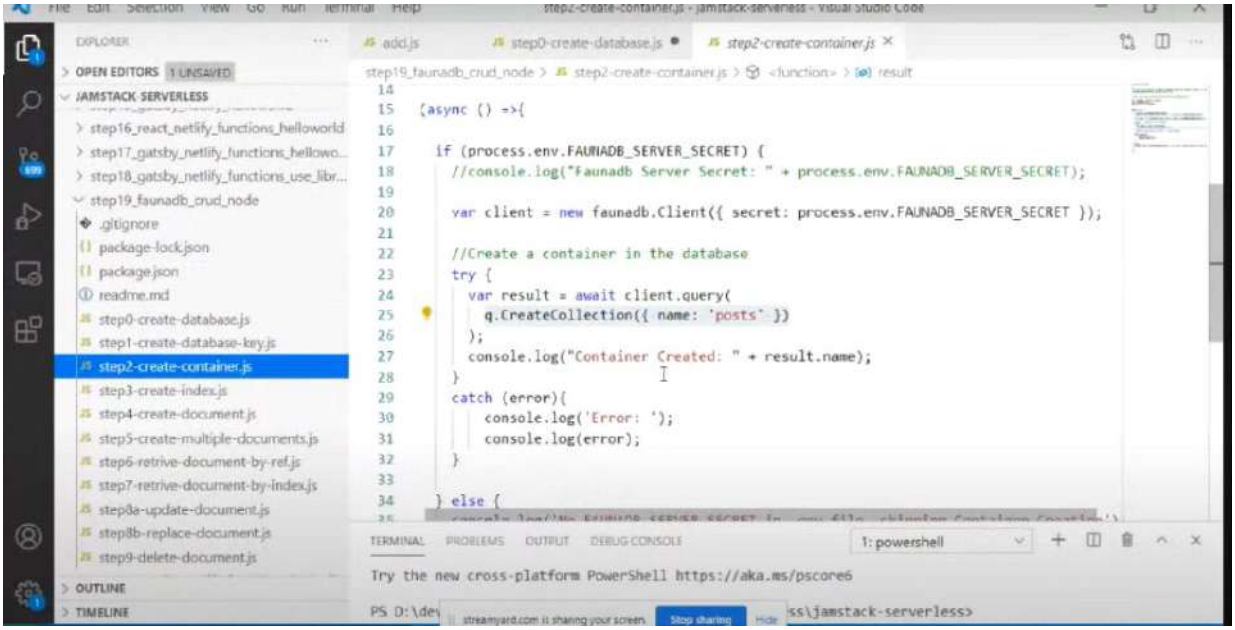


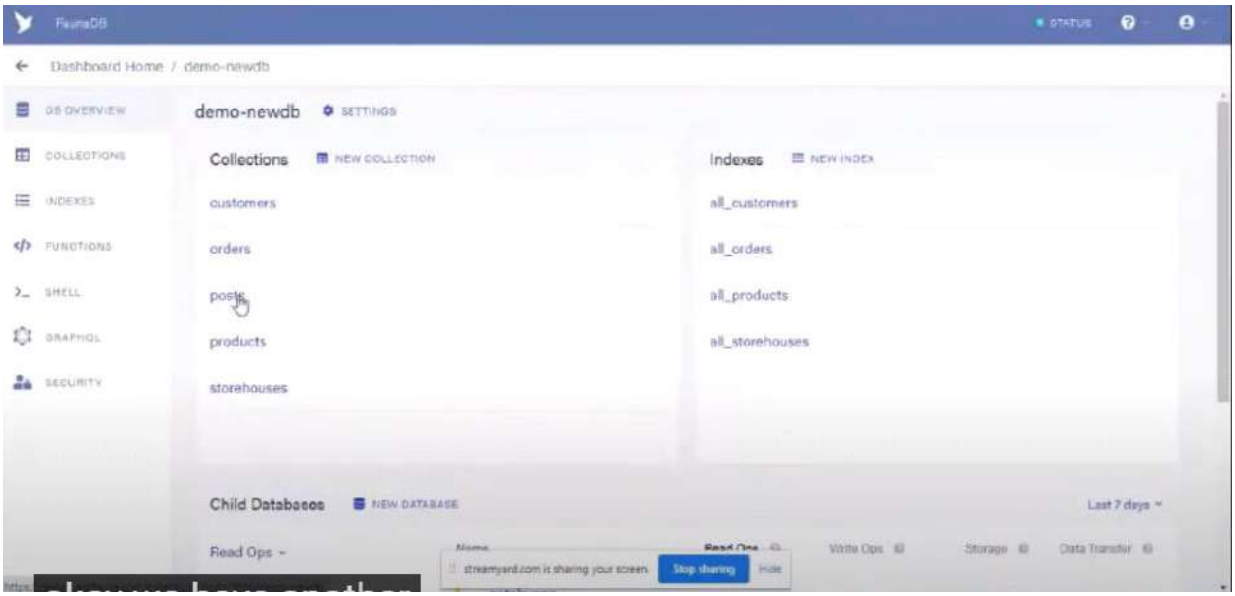
Create key

Created key from the the settings we can also create in security because we have a created key we can create key from the client side as well because we have created the database from the code so we can also called the key from the code as its mention here everything will be the same but we are just doing is again client.inside.qd we are calling q.create key and round bracket you are passing the name of the database and your passing the role as we have seen when I try to create a new key it ask me the role in admin or server which type of key you want to create you can specify the role from your code so what I do is go to a code and comment on line as shown.



At this moment this key represents the key for new demo new db so it can access this collections it can access this child databases if you want to call directly for that particular database we can use that as well ok now what will do is create container of the collection so as we have seen here once we created the database it has given us a few collections for that we can add document inside that collection like you will be using any no sql database for that exactly for that we will be using it create collection and will pass the name in the post whatever the name he want to keep.





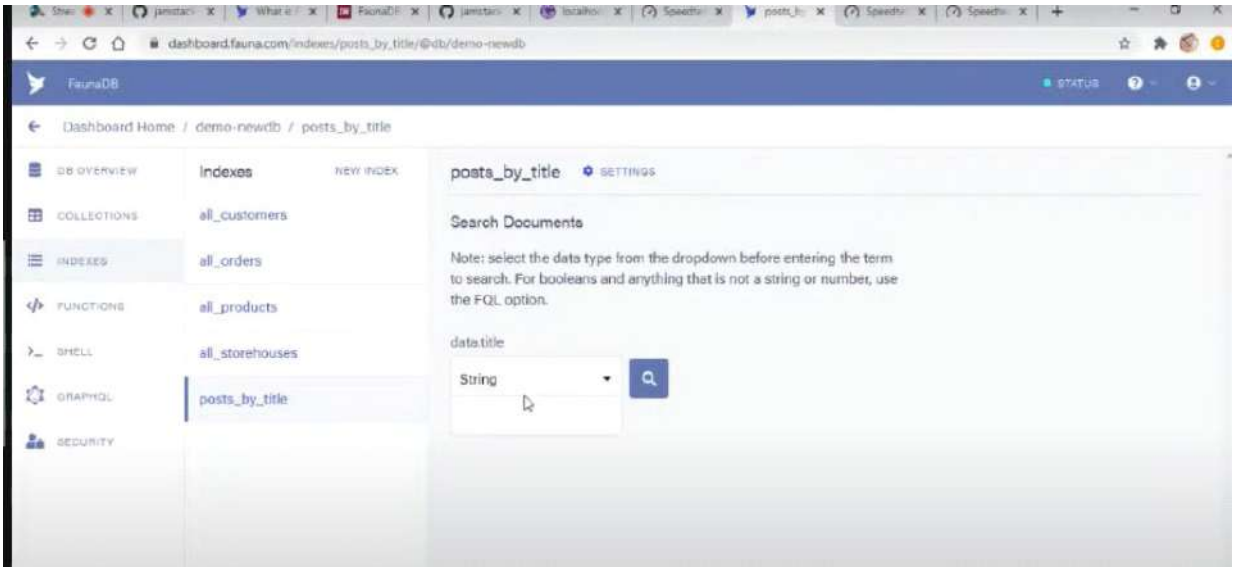
If i will change that key and use the key provided by the this step so if I use this key then my collection will be inside getsby app database at this moment we are using the parent database key the collection of name post created over there is no text document inside so let's at the document inside it but first Let's create index so we have create index function will pass the name of the index and we will passed which collection on which collection you want to create index and on what field for example name a gmail address there will be 10 20 fields for in a particular collection so on what field you want to create index

```
var client = new faunadb.Client({ secret: process.env.FAUNADB_SERVER_SECRET });

//Create a index of the container in the database
try {
  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){
    console.log('Error: ');
    console.log(error);
}

} else {
  console.log('No FAUNADB_SERVER_SECRET in .env file, skipping Index Creation');
```

- So close by title so assuming that we are creating a post and a collection of cost and inside that course we have a title property.

```
Result =  fnAD3fz55AACBzLZ4WUkm-do9kSwLqP3eZrrc0tH

D:\developmentData\bootcamp2020\jamstack-serverless\learn-funadb\demo-faunadb>node step1.js
Result =  undefined

D:\developmentData\bootcamp2020\jamstack-serverless\learn-funadb\demo-faunadb>node step1.js
Result =  {
  ref: Index("posts_by_title"),
  ts: 1602007016235000,
  active: true,
  serialized: true,
  name: 'posts_by_title',
  source: Collection("posts"),
  terms: [ { field: [Array] } ],
  partitions: 1
}

D:\developmentData\bootcamp2020\jamstack-serverless\learn-funadb\demo-faunadb>
```

- Now we check in our database.

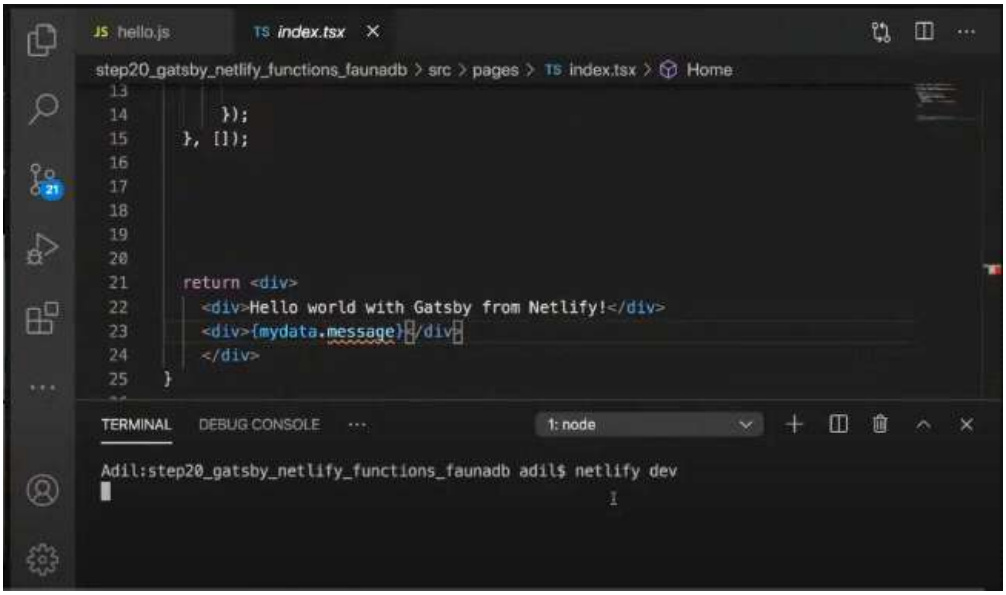

dashboard.fauna.com/indexes/posts_by_title/@db/demo-newdb
FaunaDB
STATUS
Dashboard Home / demo-newdb / posts_by_title
DB OVERVIEW
COLLECTIONS
INDEXES
FUNCTIONS
SHELL
GRAPHQL
SECURITY
Indexes
NEW INDEX
all_customers
all_orders
all_products
all_storehouses
posts_by_title
posts_by_title
SETTINGS
Search Documents
Note: select the data type from the dropdown before entering the term to search. For booleans and anything that is not a string or number, use the FQL option.
data.title
String

SESSION-37

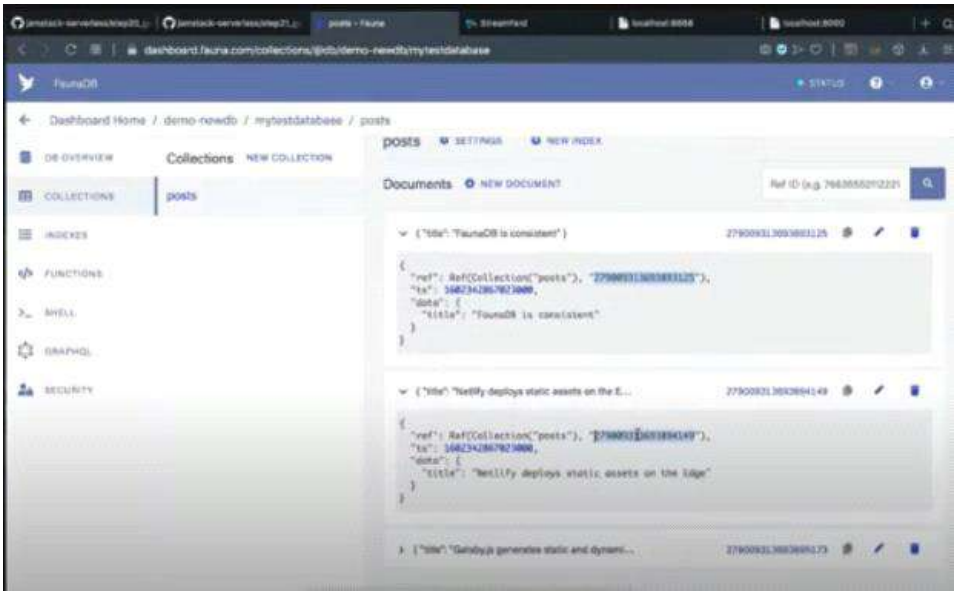
Step 20

Gatsby Netlify Function Faunadb

1.To run the command in your project, go to functions terminal and add netlify dev



2. To change the query, replace the command code with the code present in the document of local host.



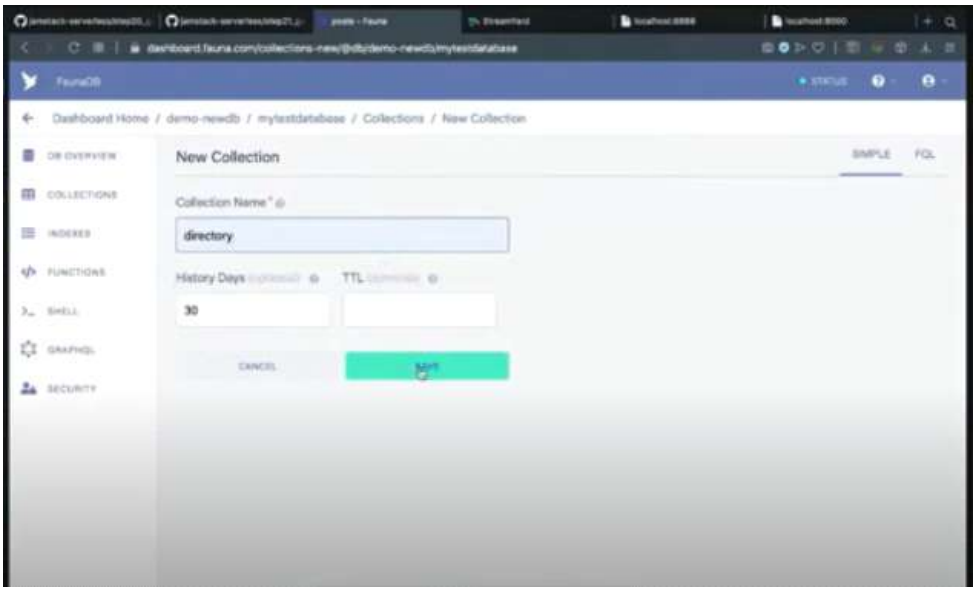
STEP 21

Gatsby Netlify Function Faunadb Formik

1.Go to step 21, open “add.js” and then make changes in this format.

```
return <div>
   <div>Directory Addition Form </div>
  <Formik initialValue= ( {
      firstName:"",
      lastName:""
     age:0
}}
onSubmit={(values)=>{
     console.lo(values)=>{
     fetch['/.netlify/functions/ass", {
    method: 'post',
   body: JSON.strongify(values)
})
.then(response => response.json()}
.then(data => {
    detData(data);
    console.log("Data: " + JSON.stringify(data)}:
```

2. Go to FaunaD and create a new collection and name it directory.



3.Make some changes in the collection named directory that we just created(in “add.js and “index.tsx”).

```
JS add.js   TS index.tsx
unadb_formik > functions > add > JS add.js > handler > exports.handler > result > data > lastName
12      }
13
14      let reqObj = JSON.parse(event.body);
15
16      var client = new faunadb.Client({ secret: process.env.FAUNADB_ADMIN_SECRET
17
18      var result = await client.query(
19        q.Create(
20          q.Collection('directory'),
21          { data: { firstName: reqObj.firstName, lastName: reqObj.lastName, age:
22        )
23      );
24
25      console.log("Entry Created and Inserted in Container: " + result.ref.id);
26
27
28      return {
29        statusCode: 200,
30        body: JSON.stringify({ id: `${result.ref.id}` }),
31        // // more keys you can return:
```

4. Duplicate a div in formik
5. Go to step 21, open terminal and add netlify div

```
JS add.js   TS index.tsx
step21_gatsby_netlify_functions_faunadb_formik > src > pages > TS index.tsx > Home
1   import React, {useState, useEffect} from "react"
2   import { useFormik, Formik, Form, ErrorMessage, Field } from 'formik';
3   import TextField from '@material-ui/core/TextField';
4
5   export default function Home() {
6     const [mydata, setData] = useState("");
7

TERMINAL   DEBUG CONSOLE   ...   1: node

Adil:step21_gatsby_netlify_functions_faunadb_formik adil$ netlify dev
```

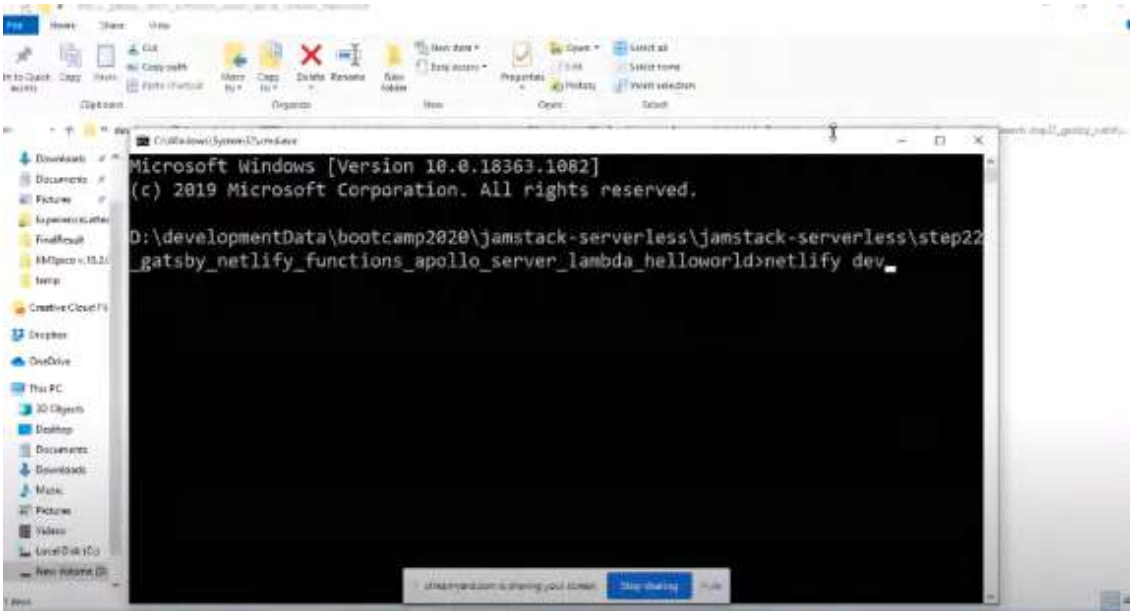
Session-38

STEP 22

Gatsby Netlify Functions Apollo Server Lambda

AWS Lambda is a service that allows users to run code without provisioning or managing servers. Cost is based on the compute time that is consumed, and there is no charge when code is not running.

- Go to create app and add netlify dev.



1. If u want to add new queries
2. Go to graphQL and add the code

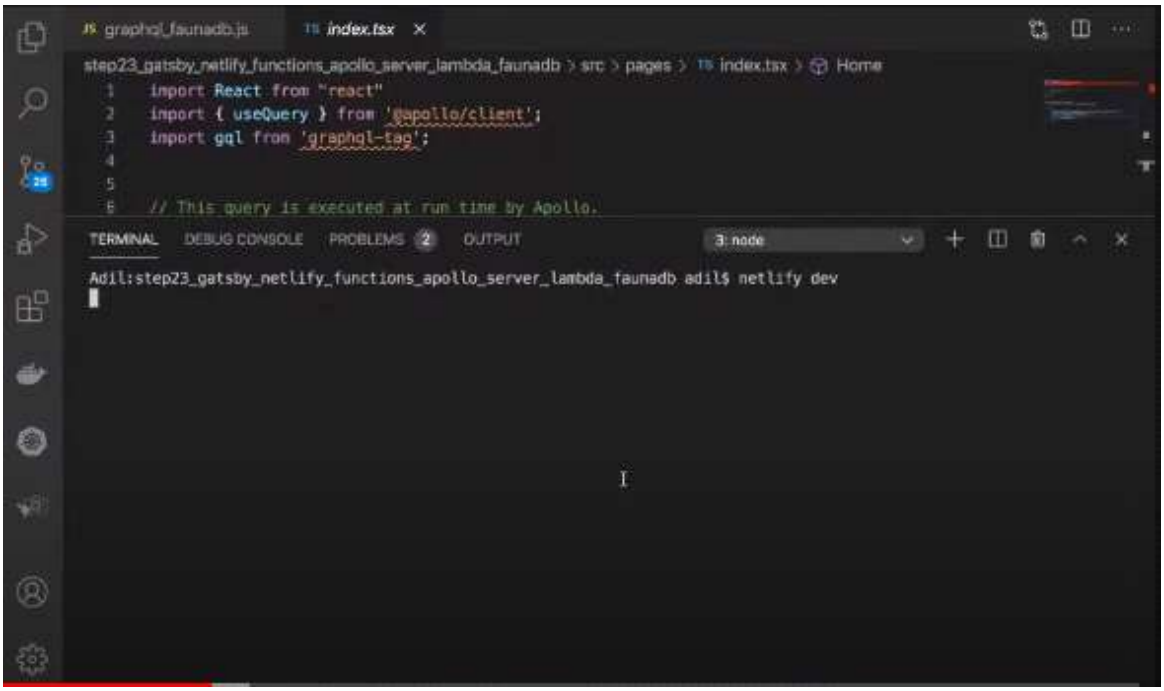
```
Const  (Appolloserver, gql ) = require ("appollo_server_lamba")

const typeDefs = gql {
message= string,
user: user
)
type user
name: string,
age: Int,

);
constant revolvers = {
Query: {
message: (parents, args, context) => {
return "Hello world zia!";
   }

}
```

3. If u want to add more users then follow the same step

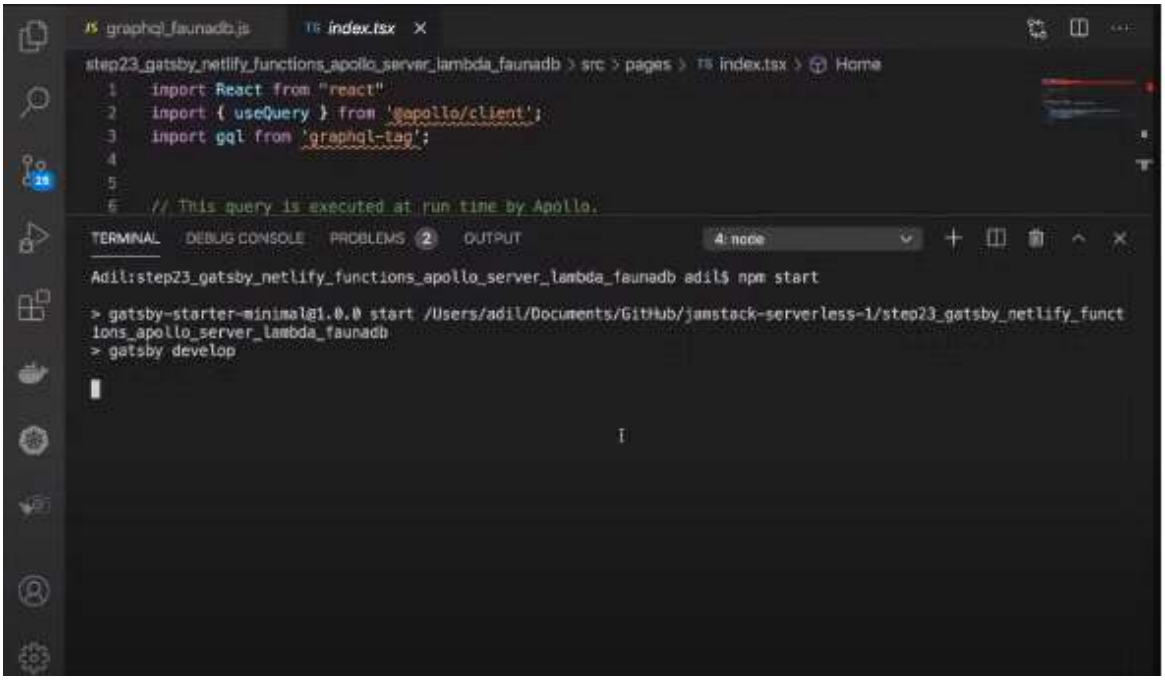
STEP 23

Gatsby Netlify Functions Apollo Server Lambda Faunadb

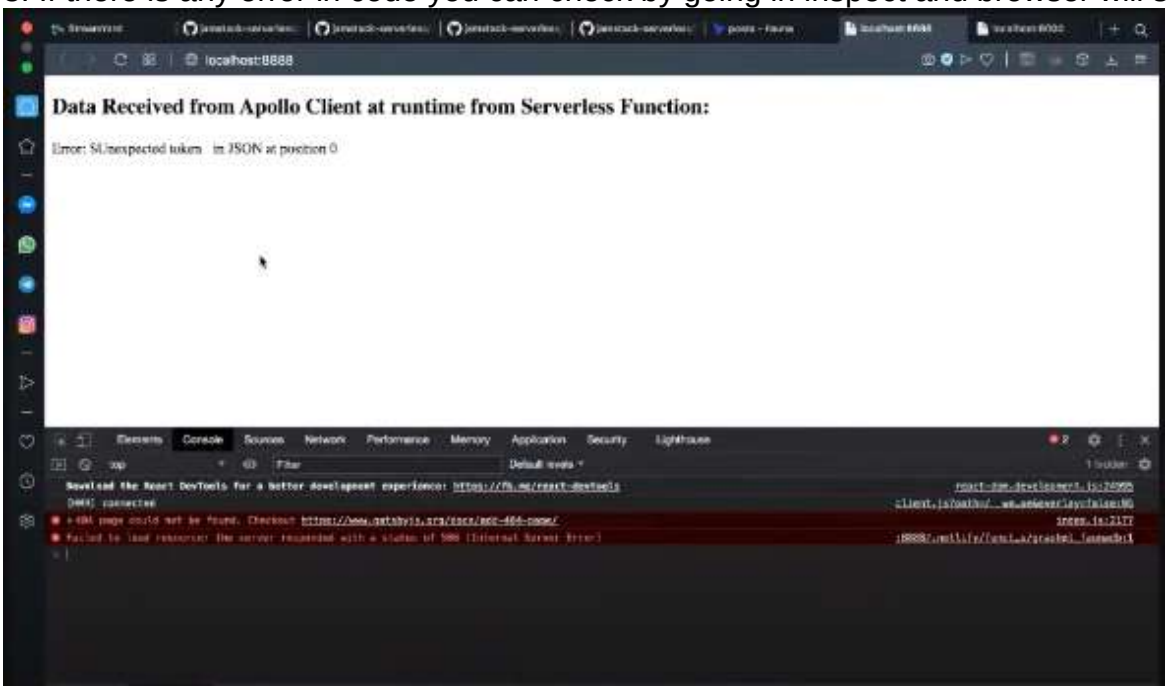
1. Go to terminal and add faunadb netlify dev.



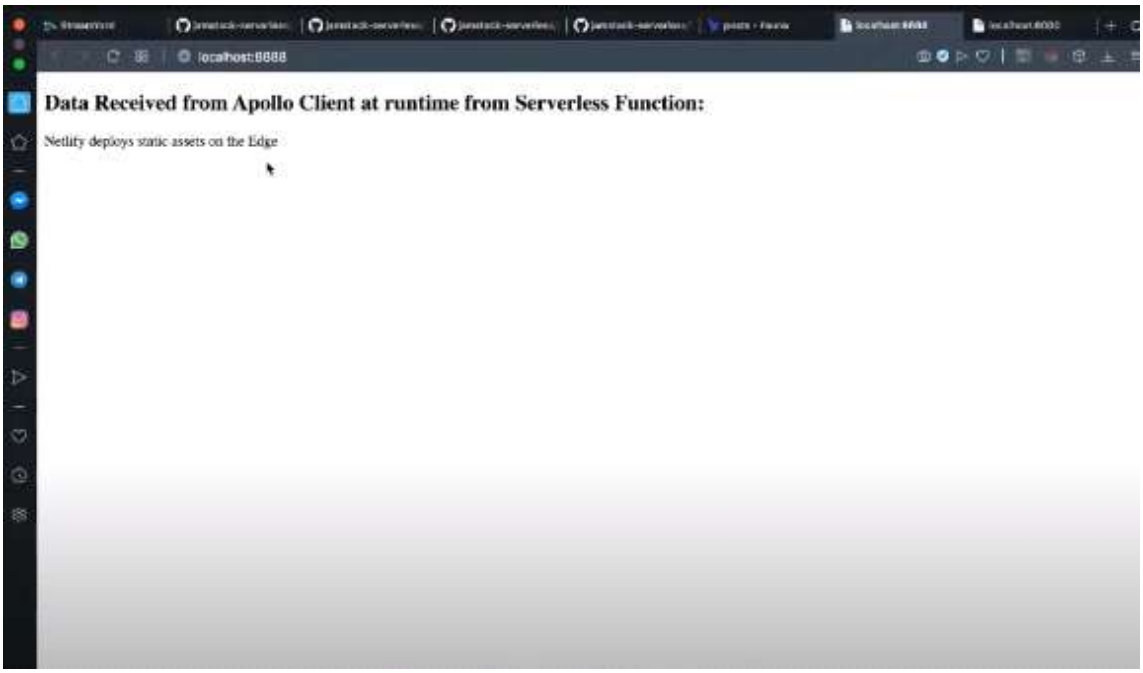
2. Go to graphQL and add npm start in the terminal



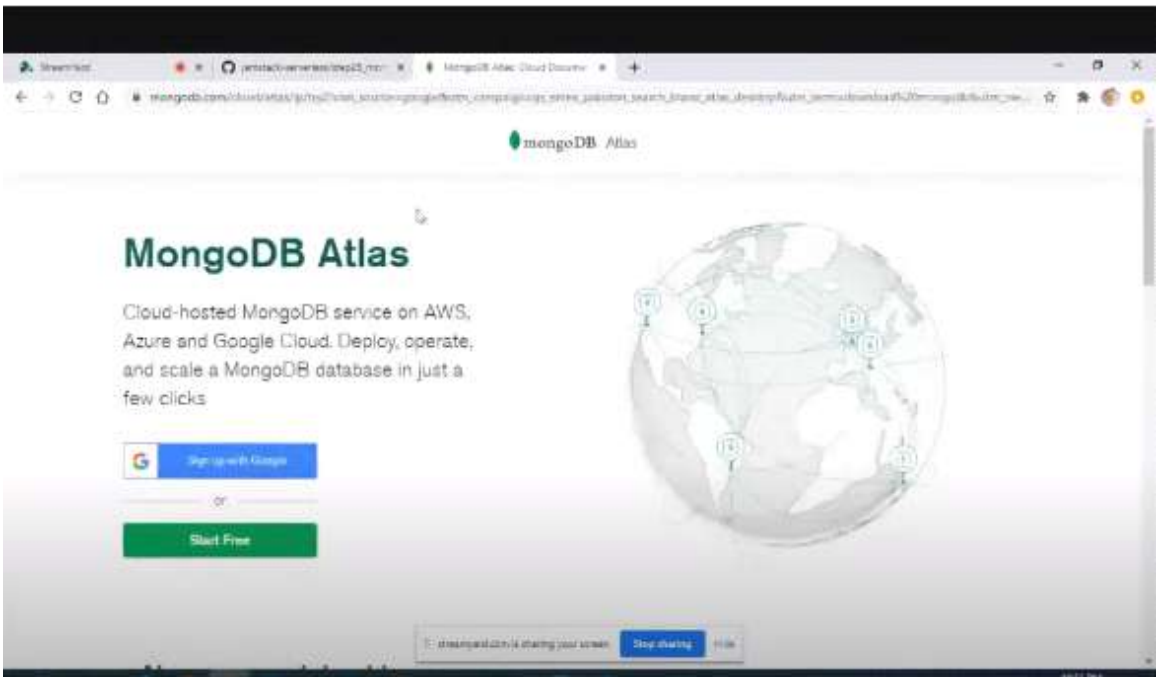
3. If there is any error in code you can check by going in inspect and browser will show the error



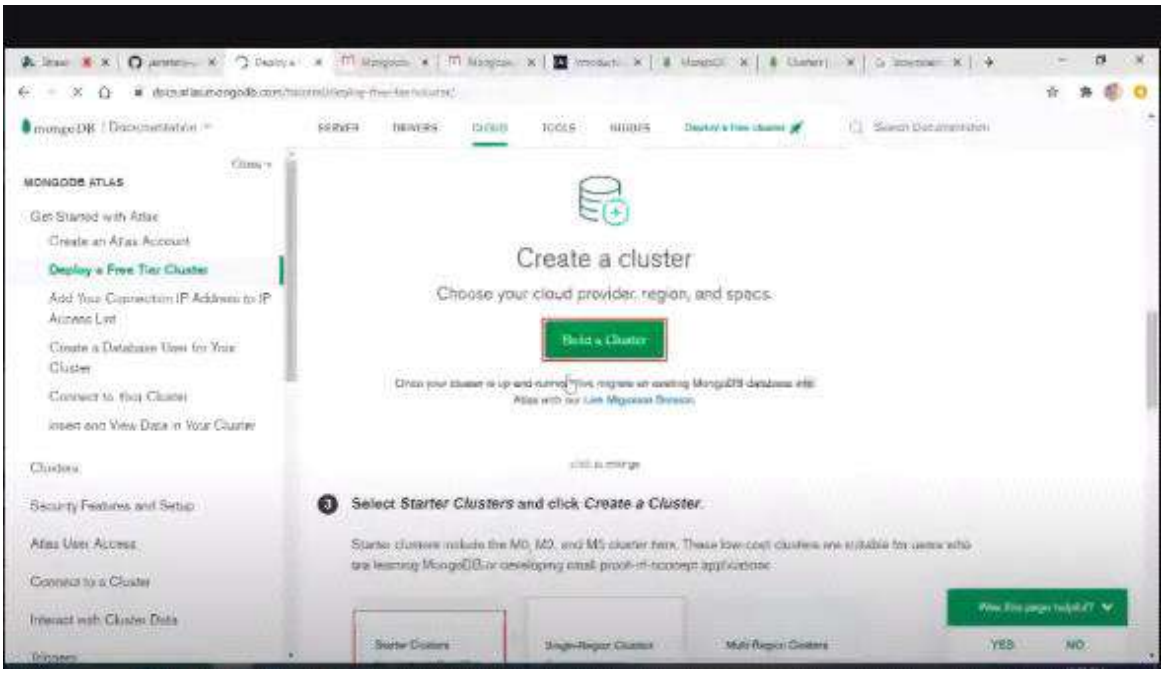
4. When it connects to the server your code will run successful and give the results as follows



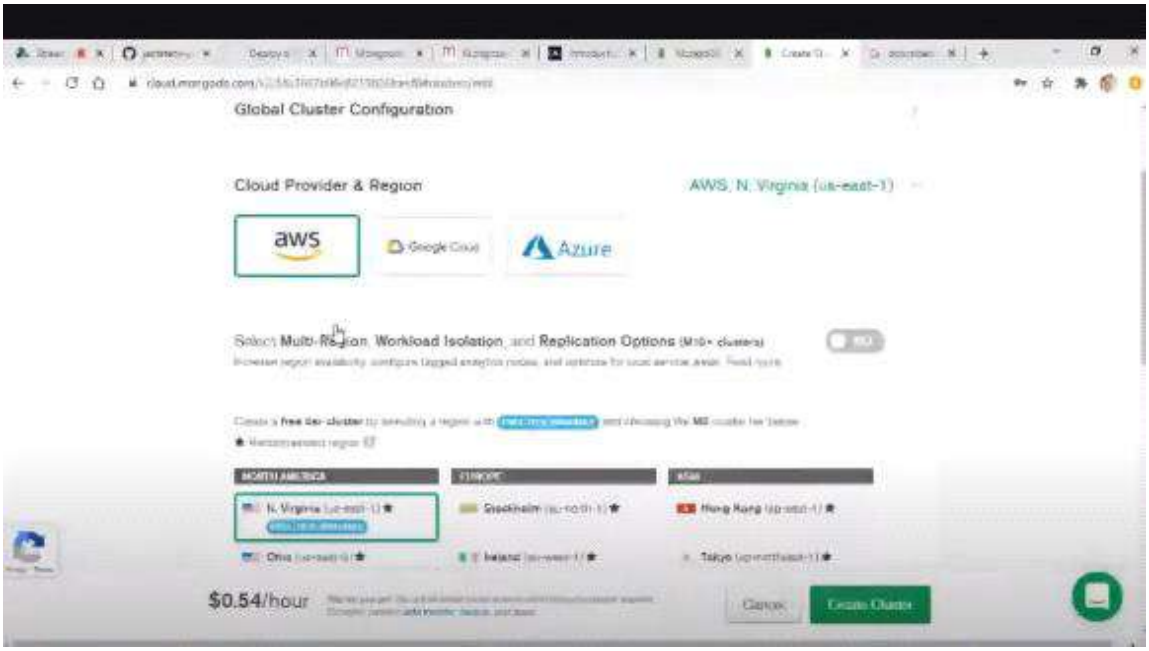
5. To install mangodb atlas database search mangodb



6. If you are new to atlas it will show you the option to create clusters otherwise get started with atlas directly



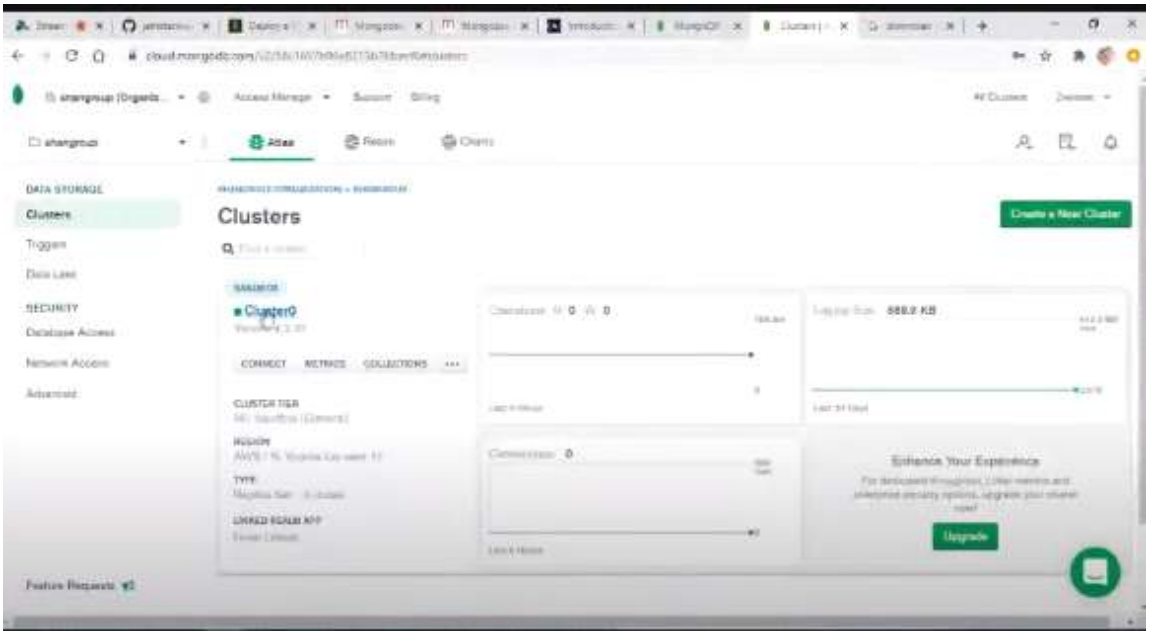
7. To create new cluster provide your server as follows



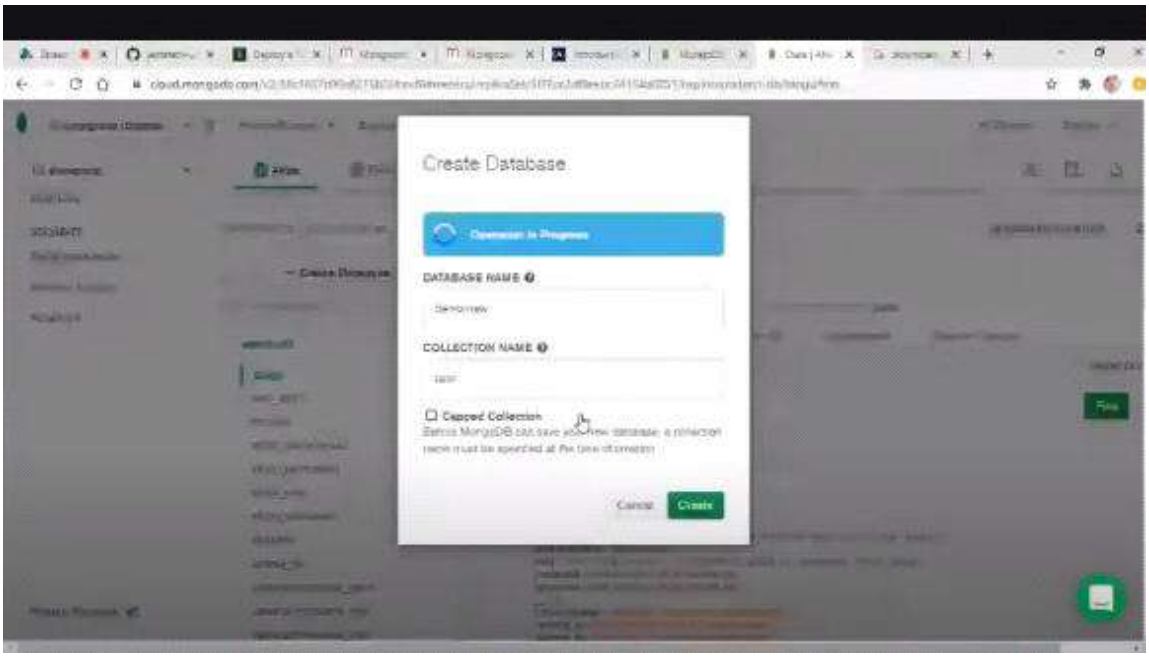
8. Select the free tier cluster and regin by default north america

9. Atlast click create cluster and if you want to give it a name you can.

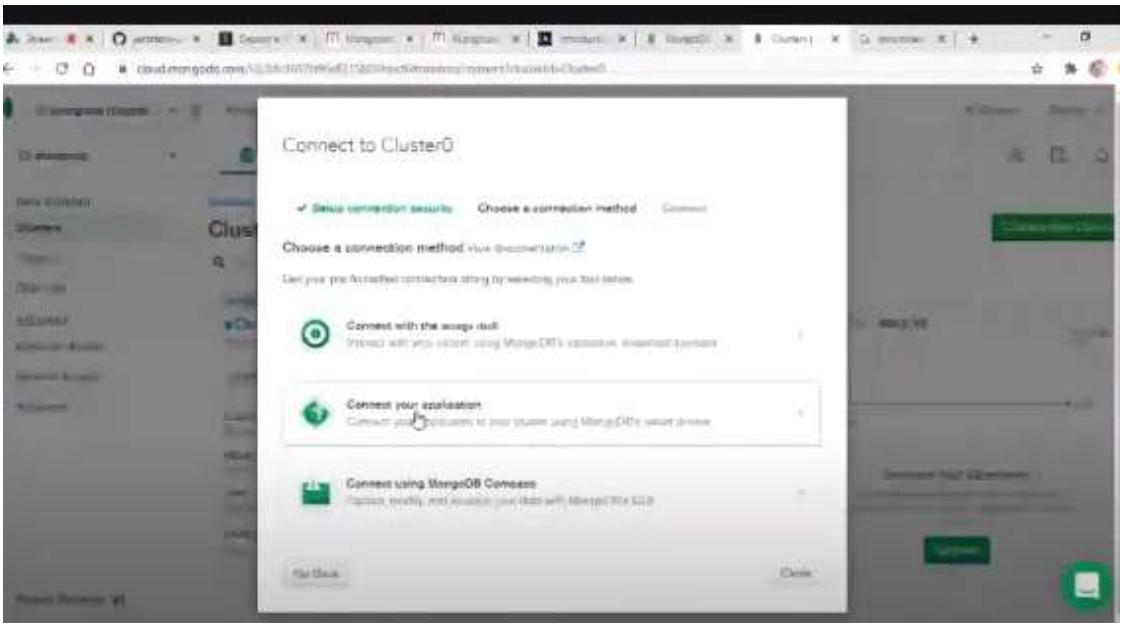
10. When your cluster is create it will show such screen



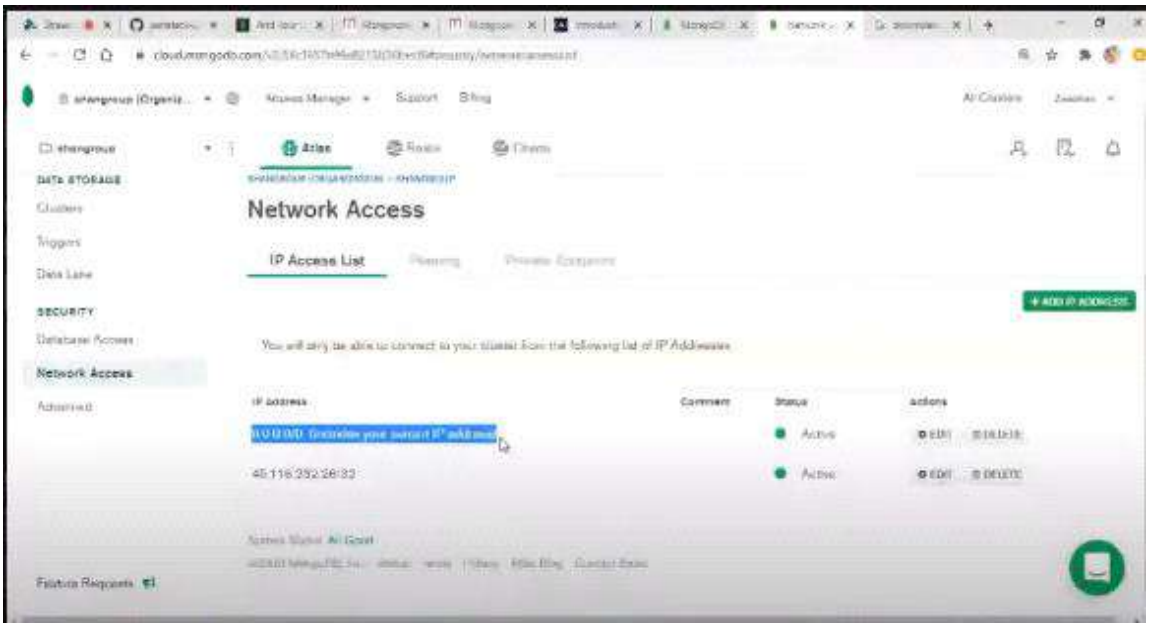
11. Create database and name it



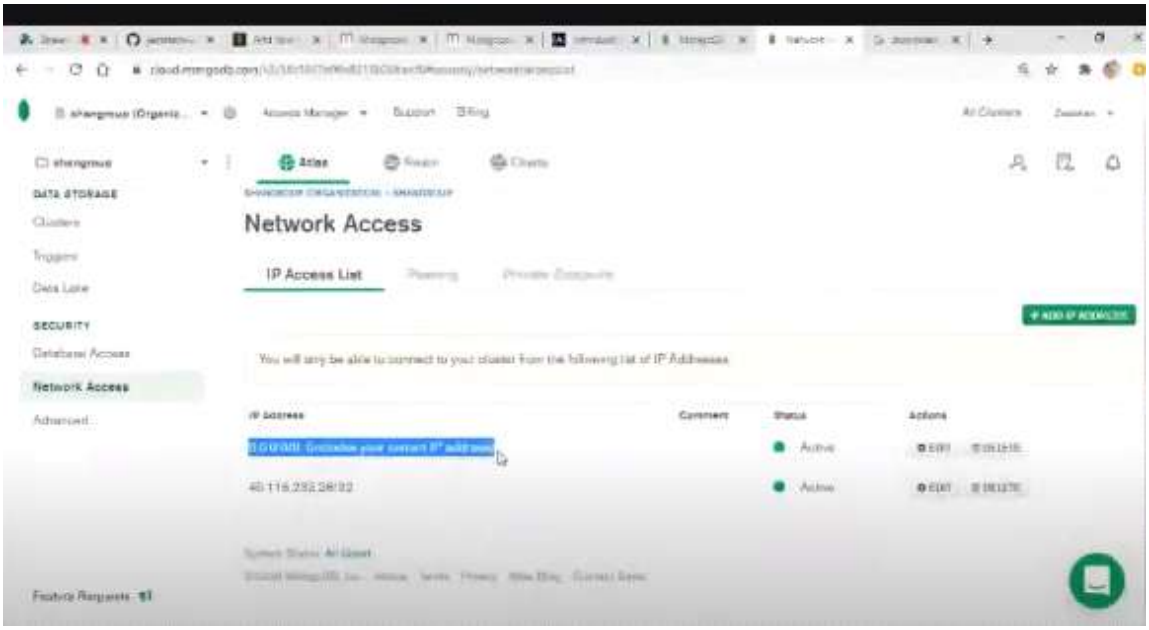
12. Click on connect option and choose where you need to connect database as we are going to connect to our code application in mangodb atlas



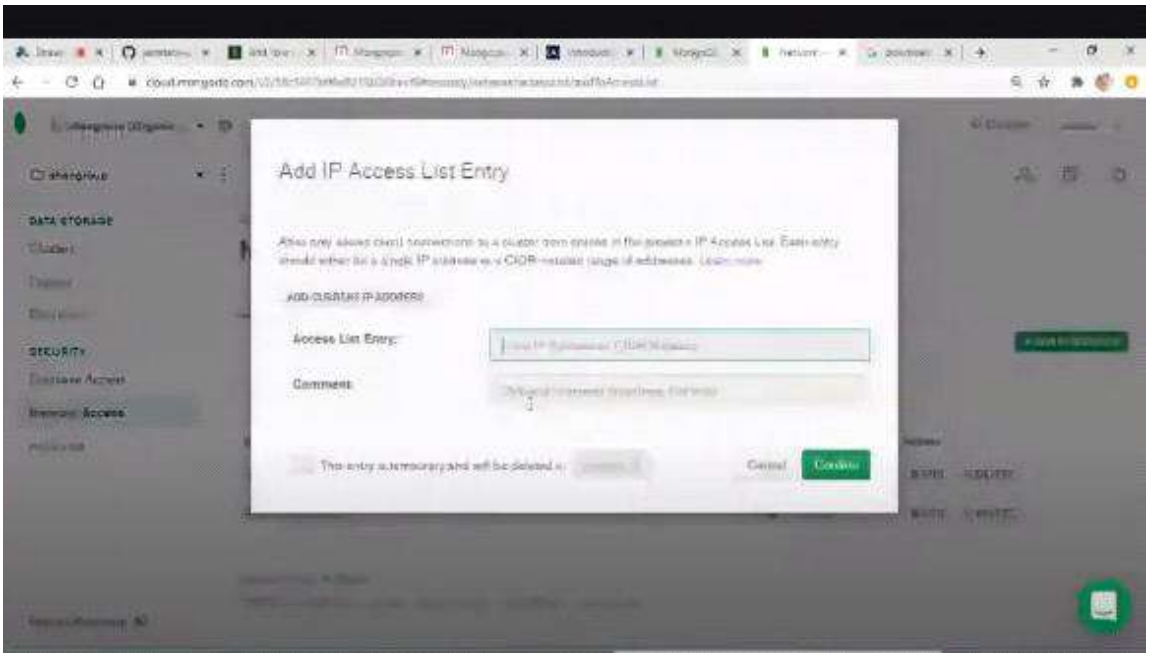
13. Once you are done with your creating clusters you have to add users
14. Whitelist your IP as shown



15. 0000 means it’s a public IP and can connect to any user



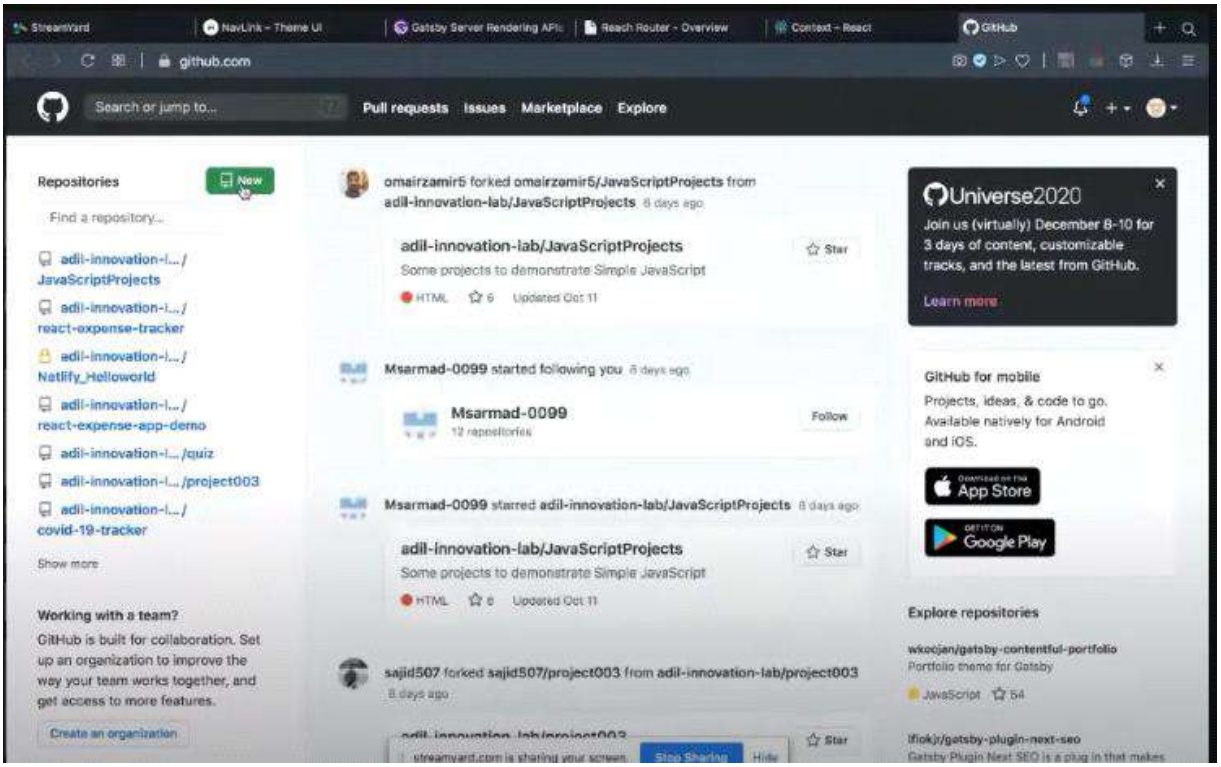
16. If u want to add particular IP you can add by click on Add An IP as shown



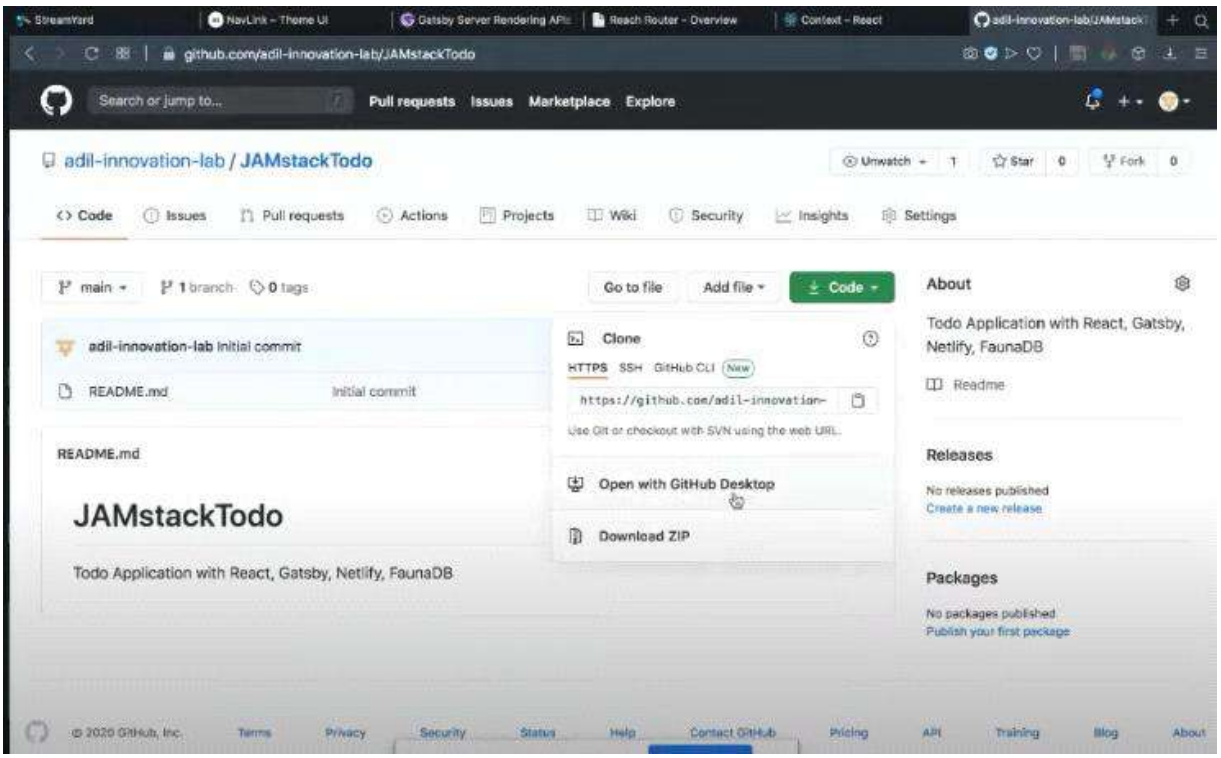
Session-39

Jamstack Todo

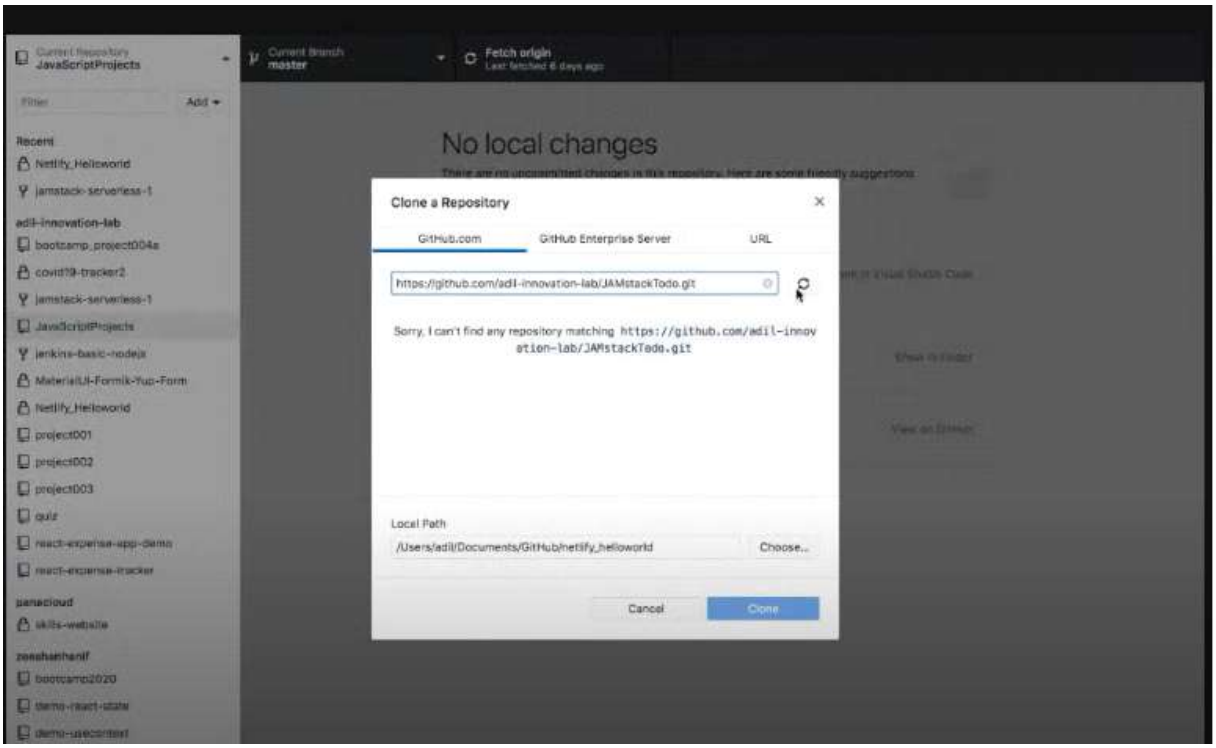
1. Go to GitHub and create a new repository



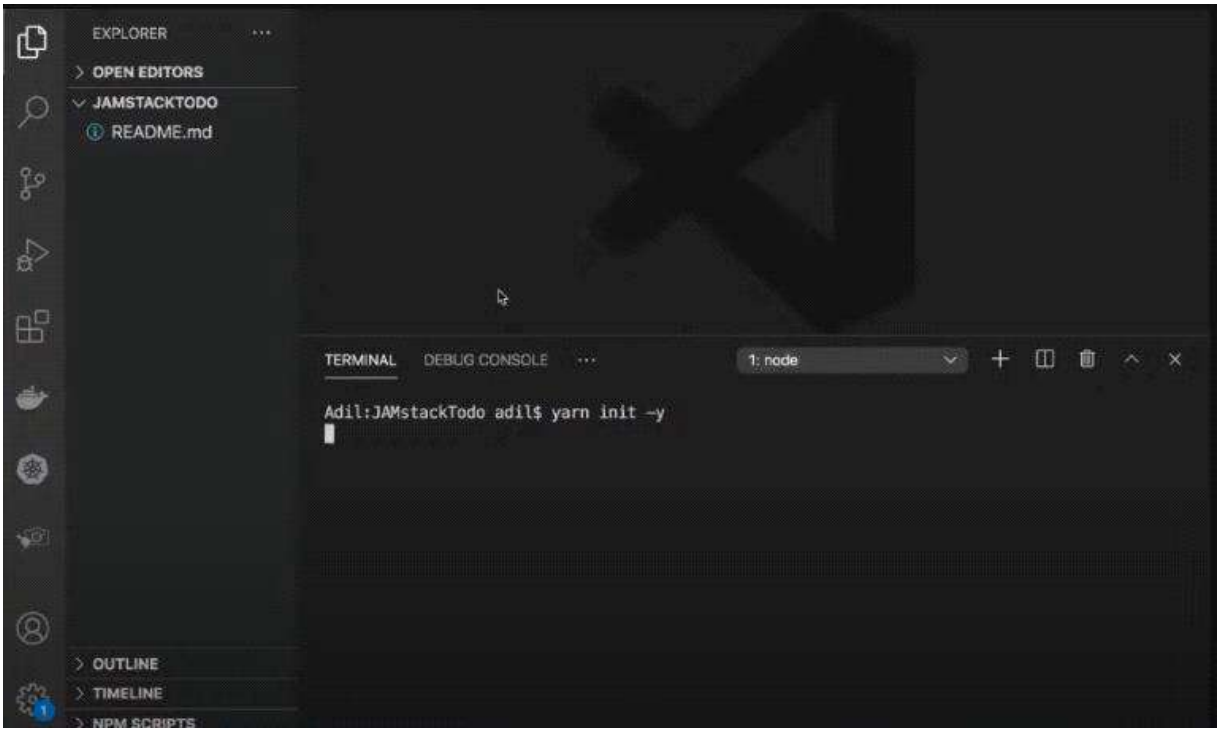
2. Copy the code from repository



3. Then go to GitHub Desktop and paste the code here



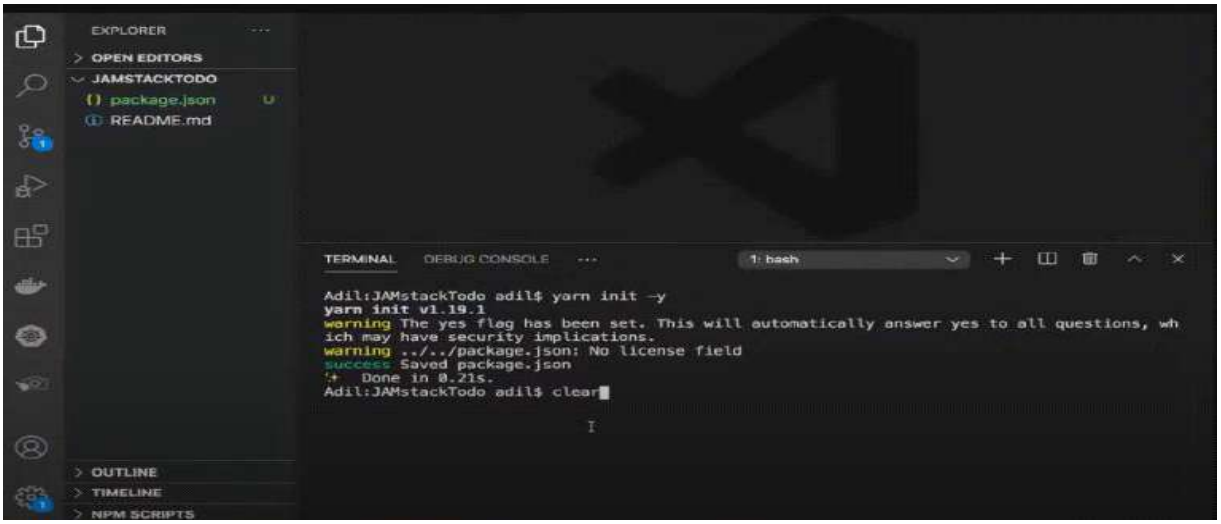
4. After the creation open the repository in coke sudio
5. Then go to integrated terminal
6. Add yarn init in terminal



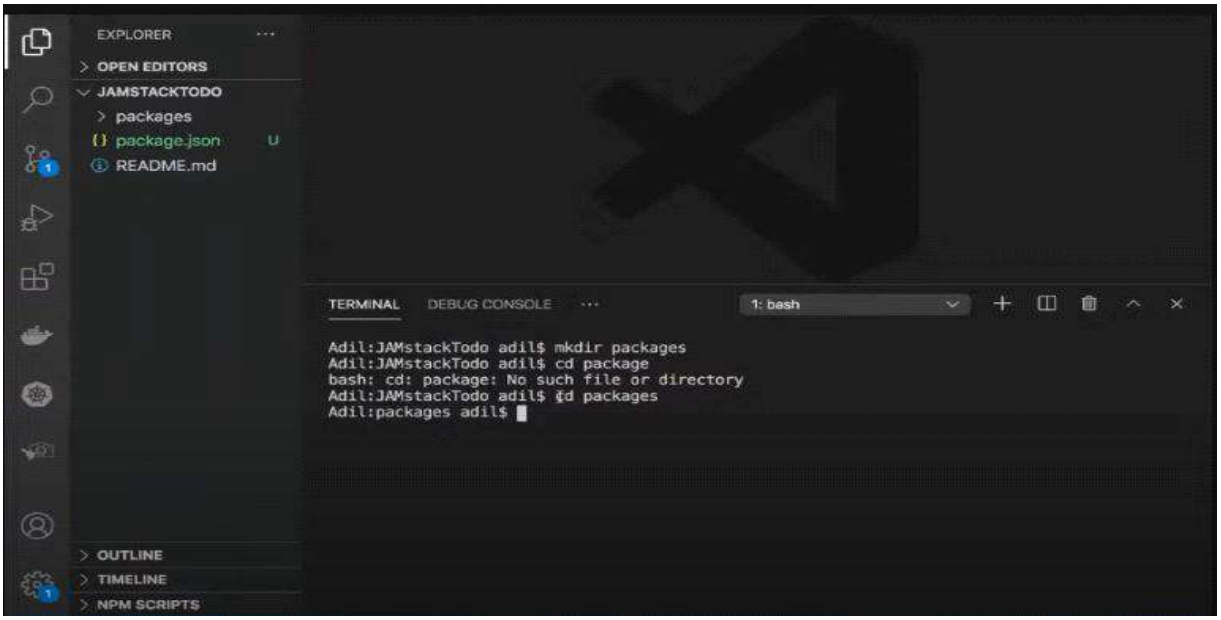
7. Go to packages.icon and make some changes in code

```
"name": "jamstacktodo"
"version": "1.0.0",
"main": "index.js",
"repository":  (respository link)
"author": (add author name)
"licsense": "MIT",
"private": true,
"workspaces": [
    "packages/*"
 ]
```

8. Add clear in terminal



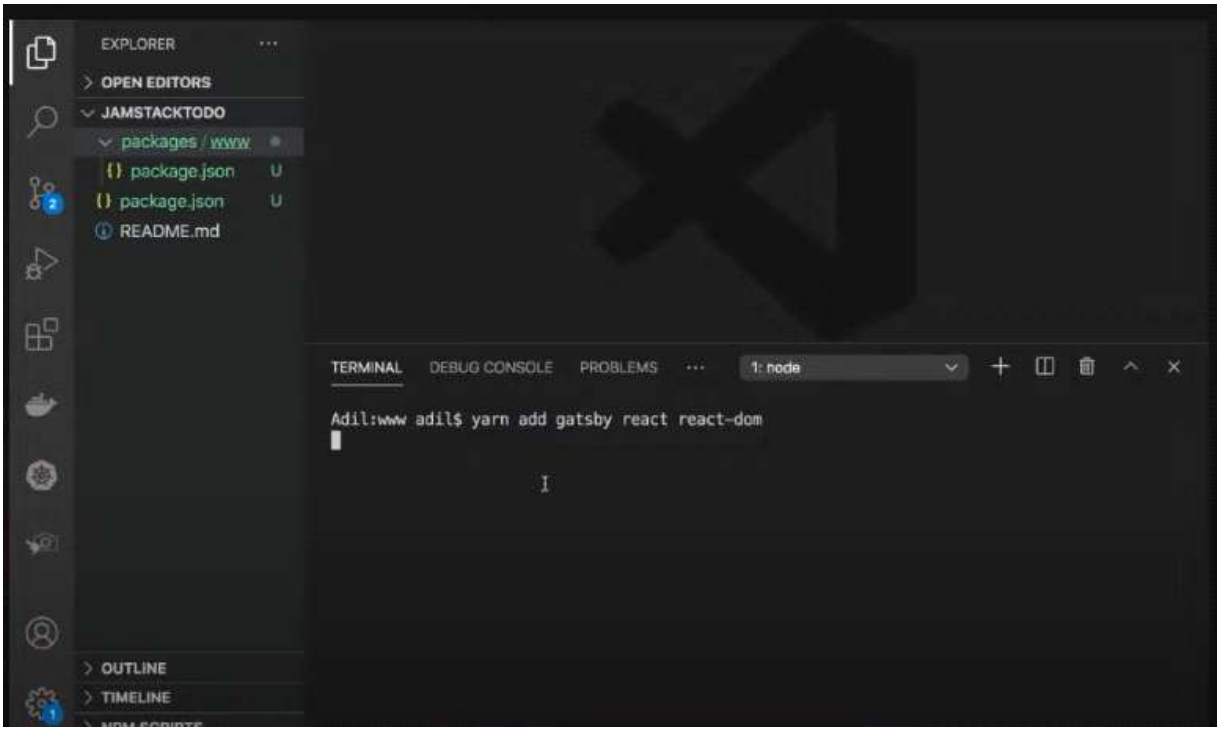
9. Then add mkdir packages and cd packages



10. Then add these codes

Admin:packages admin mkdir www
Admin:packages admin cd www
Admin:ww admin yarn init

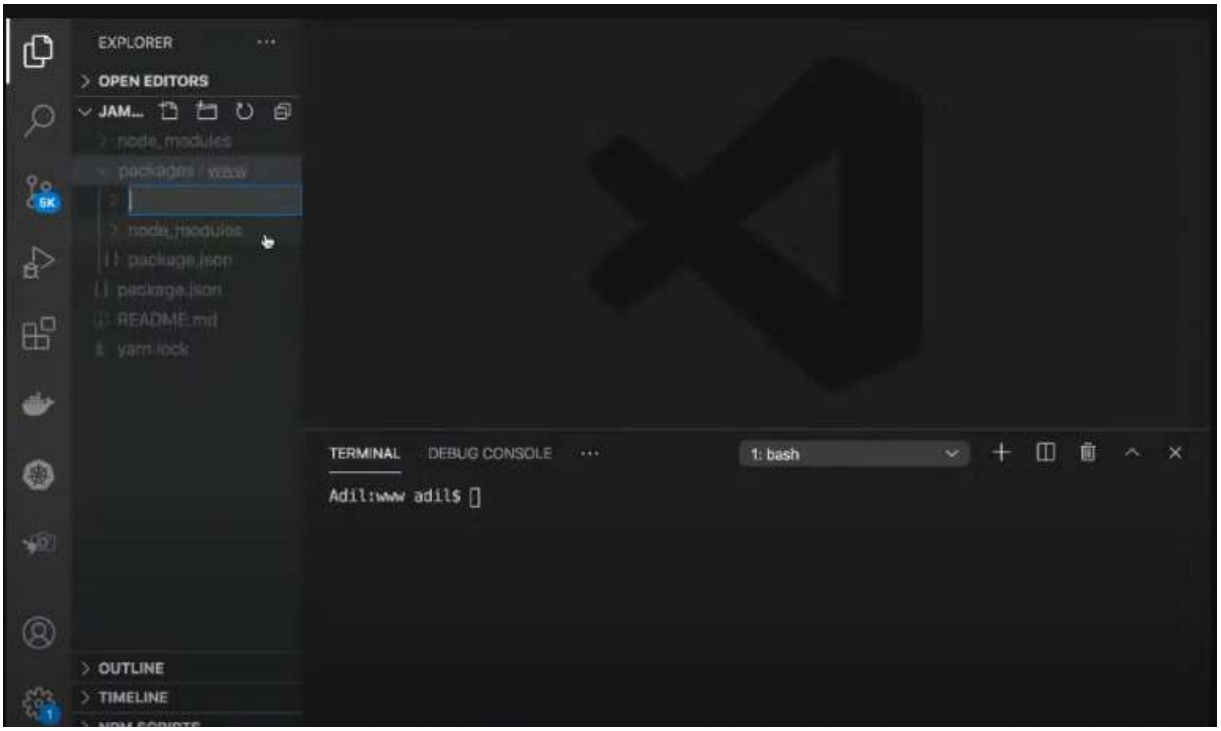
11. Then you have packages /www folder
12. Then go to terminal in packages/www folder
13. Then add yarn add gatsby react react dom



14. Then add “scripts” in the code

```
"name": "jamstacktodo"
"version": "1.0.0",
"main": "index.js",
"repository":  (respository link)
"author": (add author name)
"licsense": "MIT",
"scripts": {
   "develop": "gatsby develop",
   "build": "gatsby build",
   "clean: "gatsby clean"
"private": true,
"workspaces": [
    "packages/*"
 ]
```

16. Create a new folder in packages/www as “src”



17. Inside src add a folder name “pages”
18. Then add index.js folder in src
19. Then go to index.js and import react from react and export default props

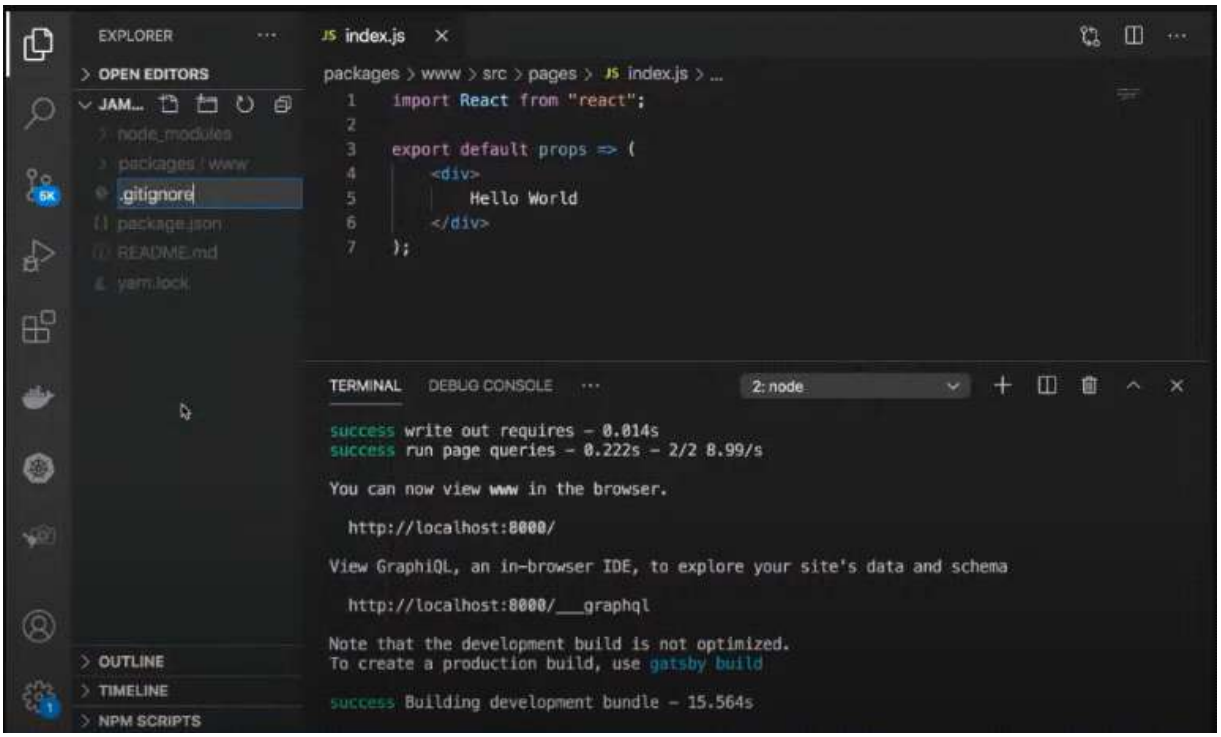
```
Import React from “react”;

Export default props => (
    <div>
        Hello World
    </div>
);
```

20. Add gatsby-config.js
21. Inside gatsby-config.js make the code

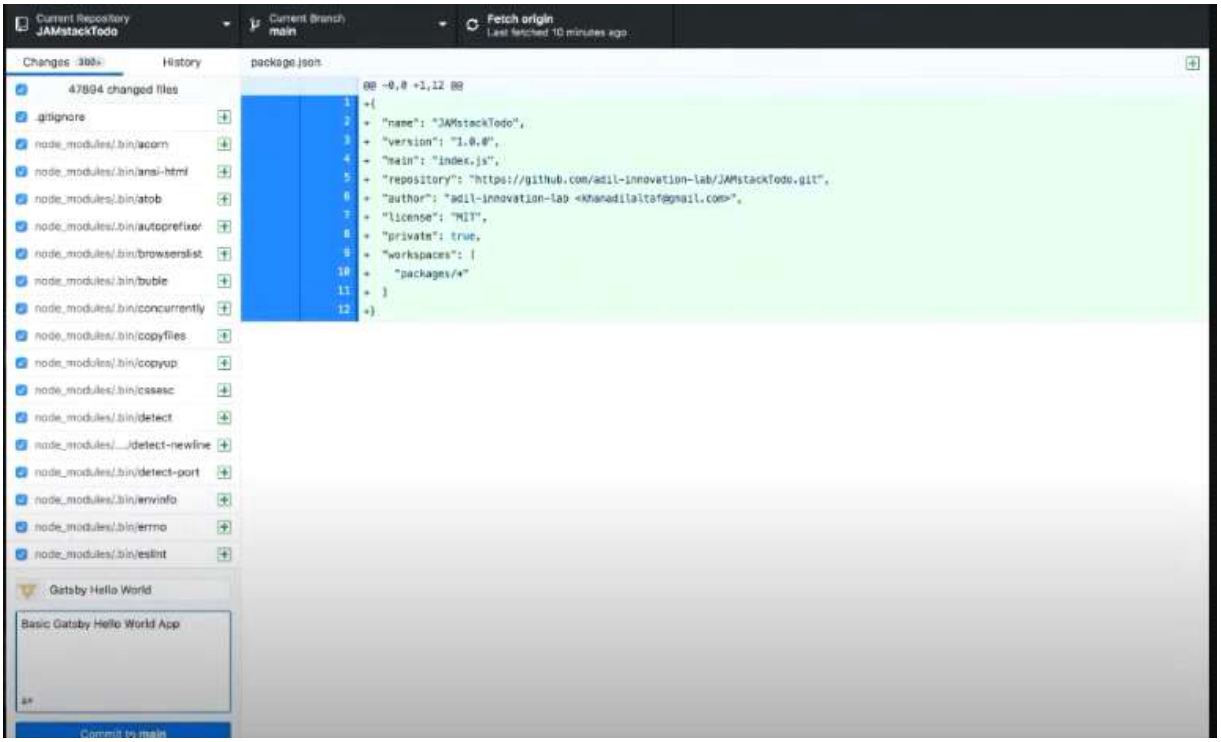
```
module.exports = {};
```

22. Then go to gatsby-config.js terminal and add yarn workspace www develop
23. Then go to "localhost8000"

24. Add new file “gitignore”

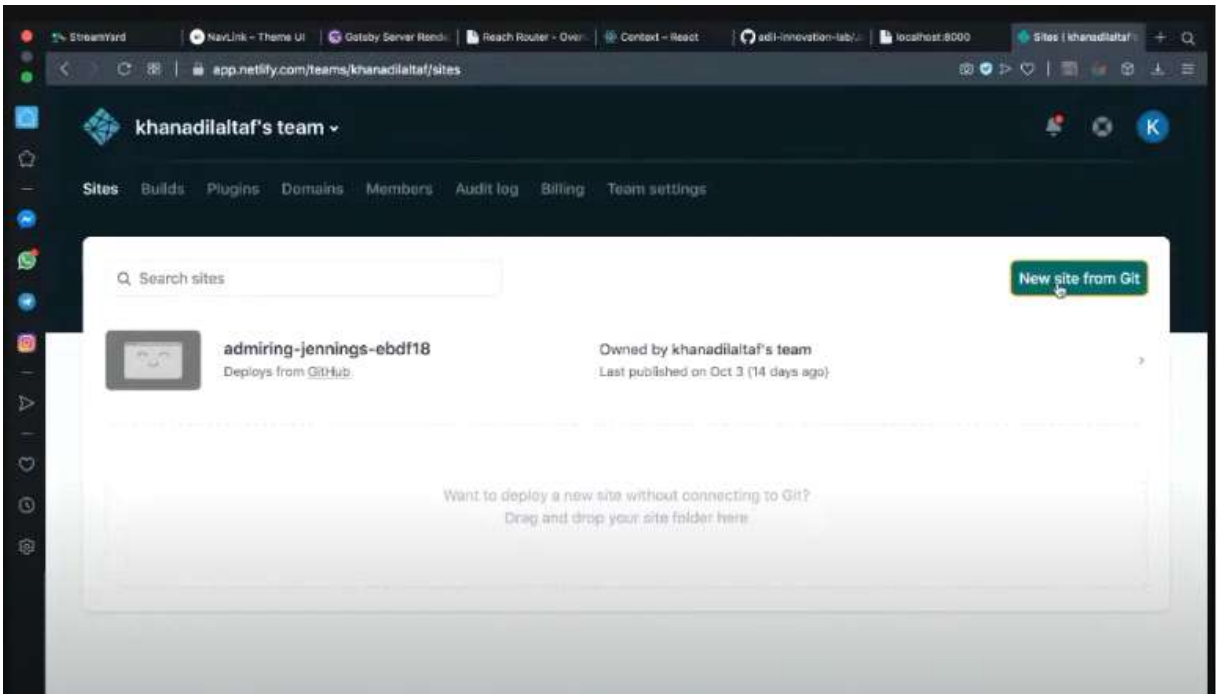


25. Go to running Gatsby site and push this

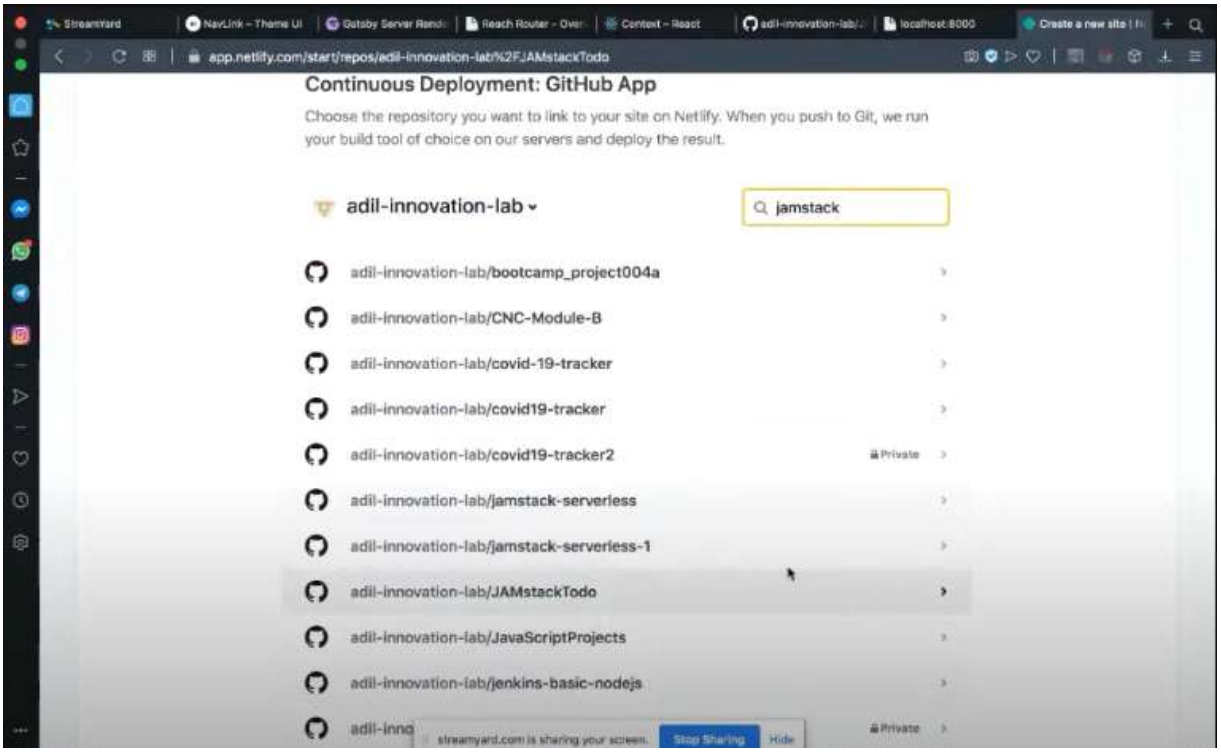


26. Setup the Gatsby site with netlify
27. Go to Netlify site
28. Go to gitignore and add node modules

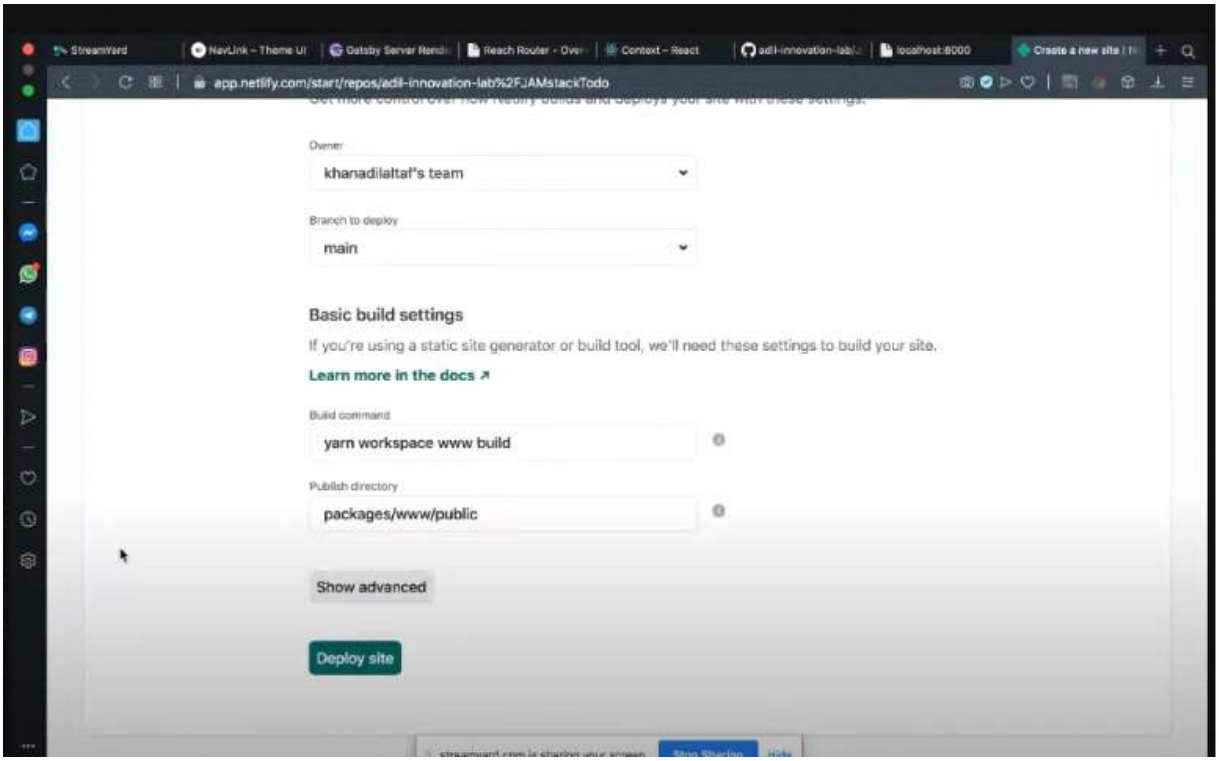
.cache
Public
Node_modules
29. Then create a new site from Git



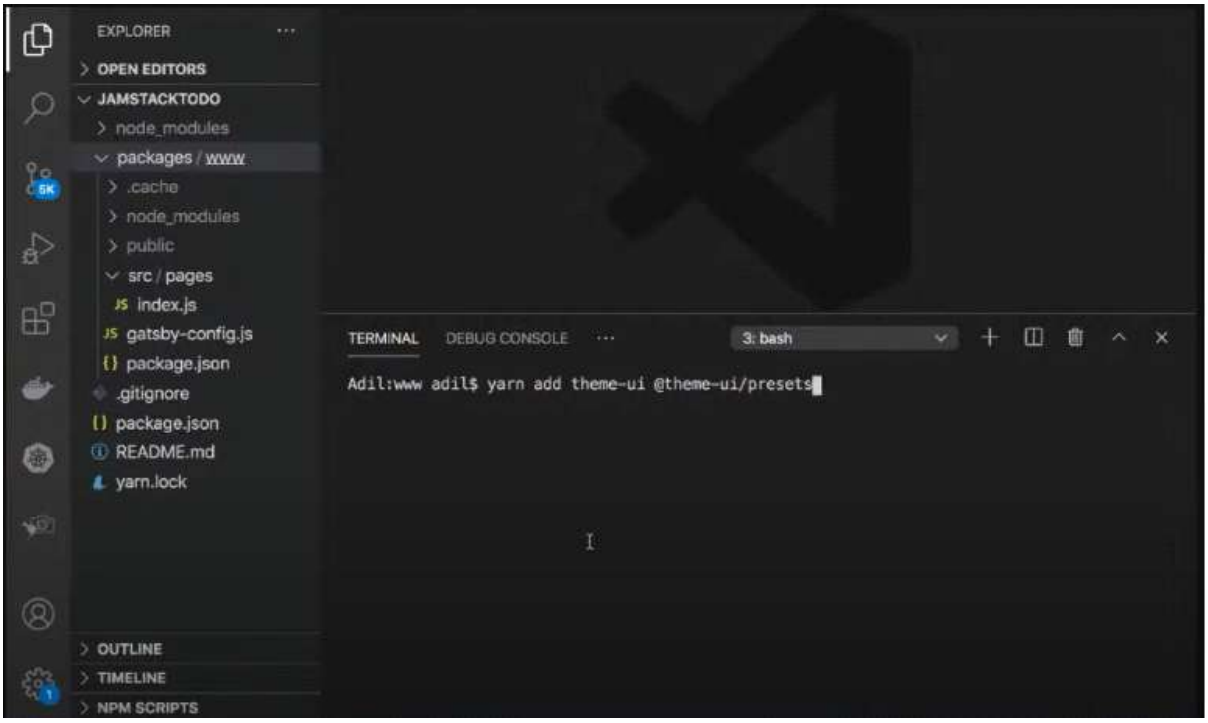
30. From GitHub
31. Then search jamstack



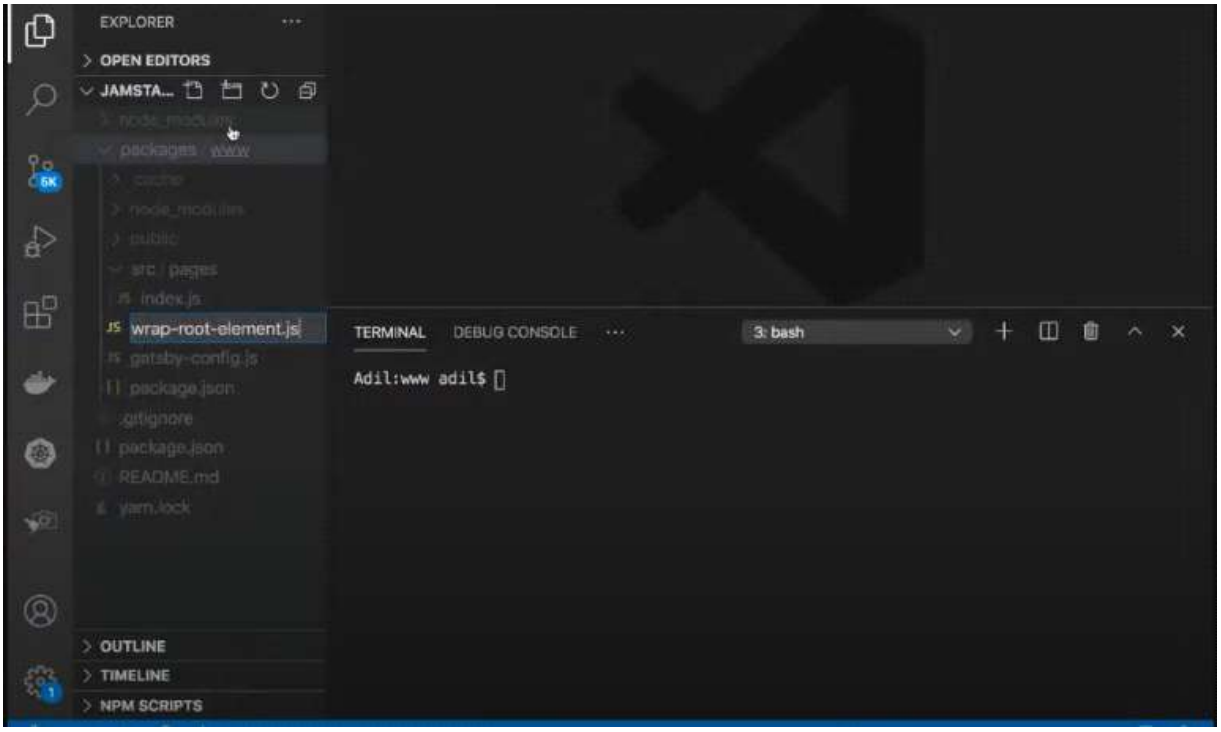
32. Then make the repository like this



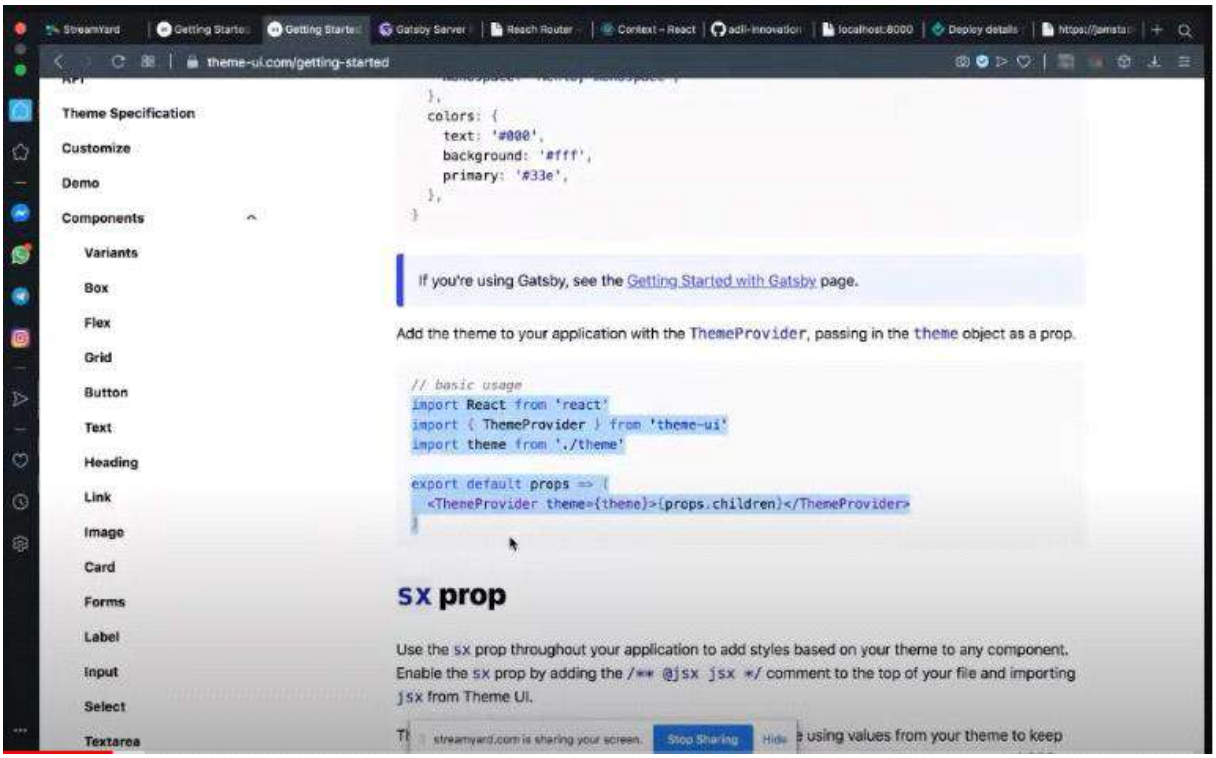
33. Deploy Site
34. Go to site setting
35. Change the site name
36. Go to packages/www terminal and add yarn theme ui @theme -ui/present



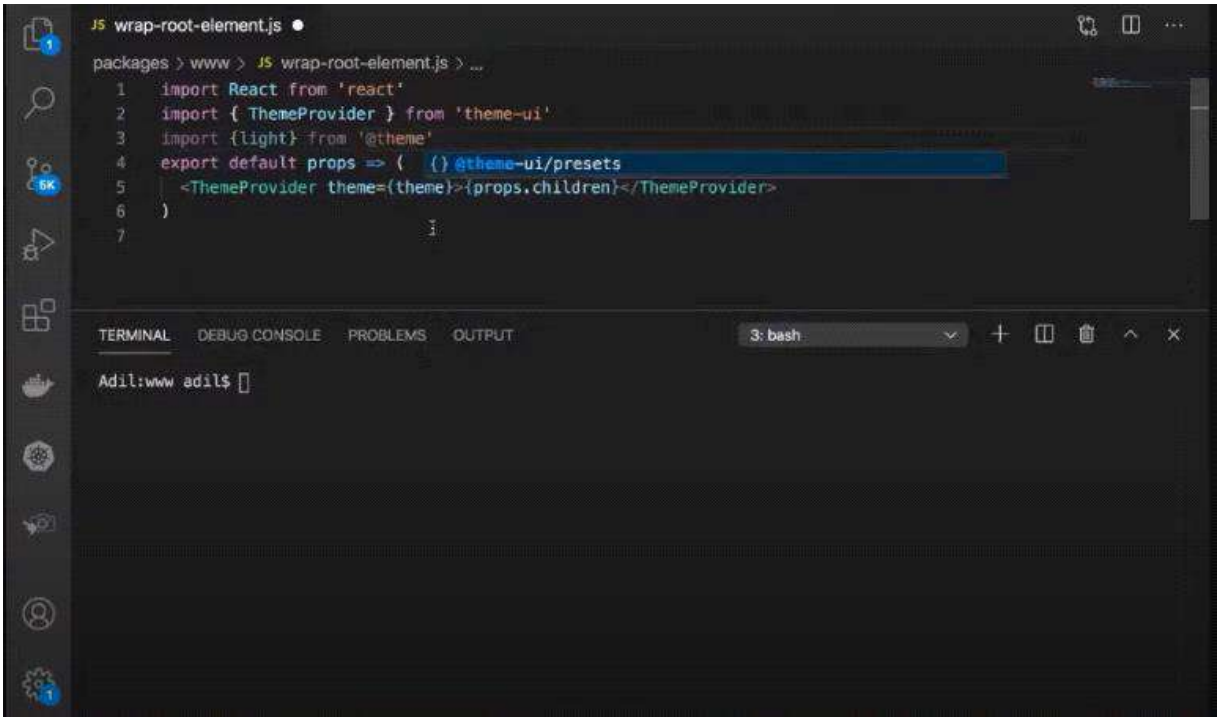
37. Go to theme ui and start it with Gatsby
38. Add new file in the project



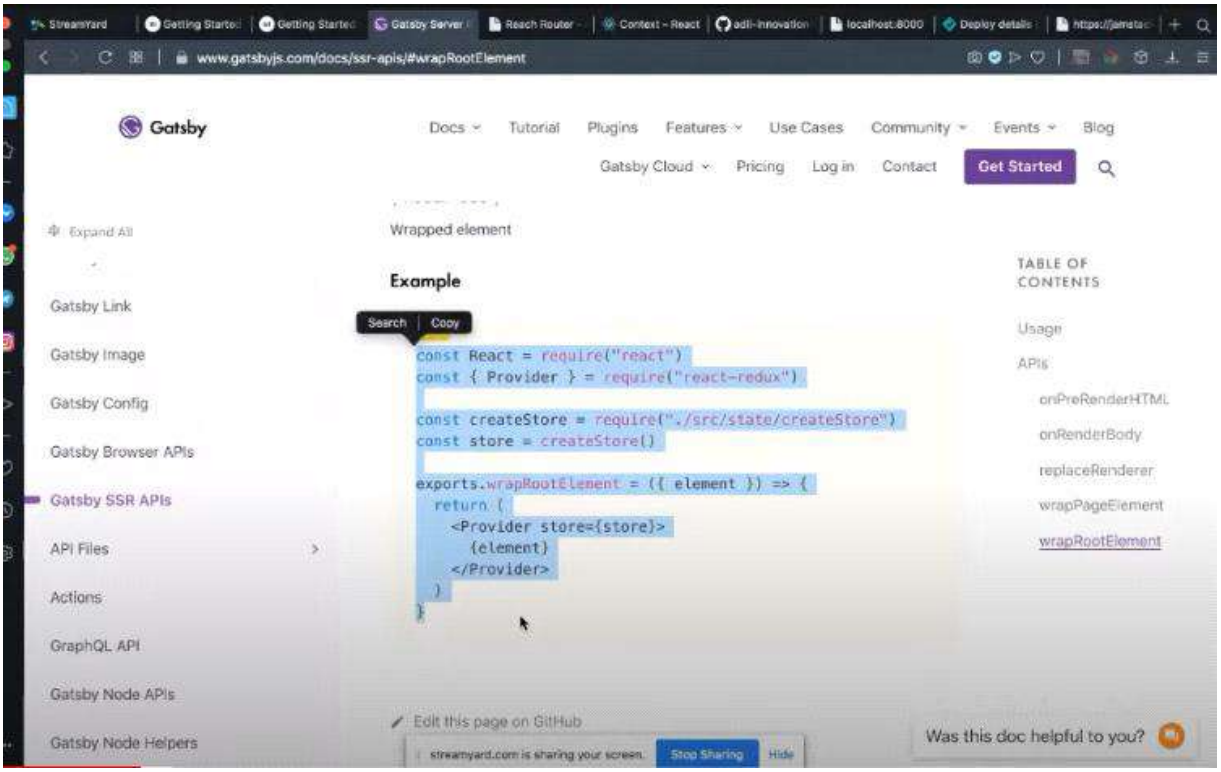
39. Inside the new file
40. Copy tthe code from theme ui



40. Paste the code in the new file
41. Update the theme according to your wish as



42. Change the props with elements
43. Create a new file “gatsby-browser.js”
44. Go to Gatsby wrapped element and copy the code from there

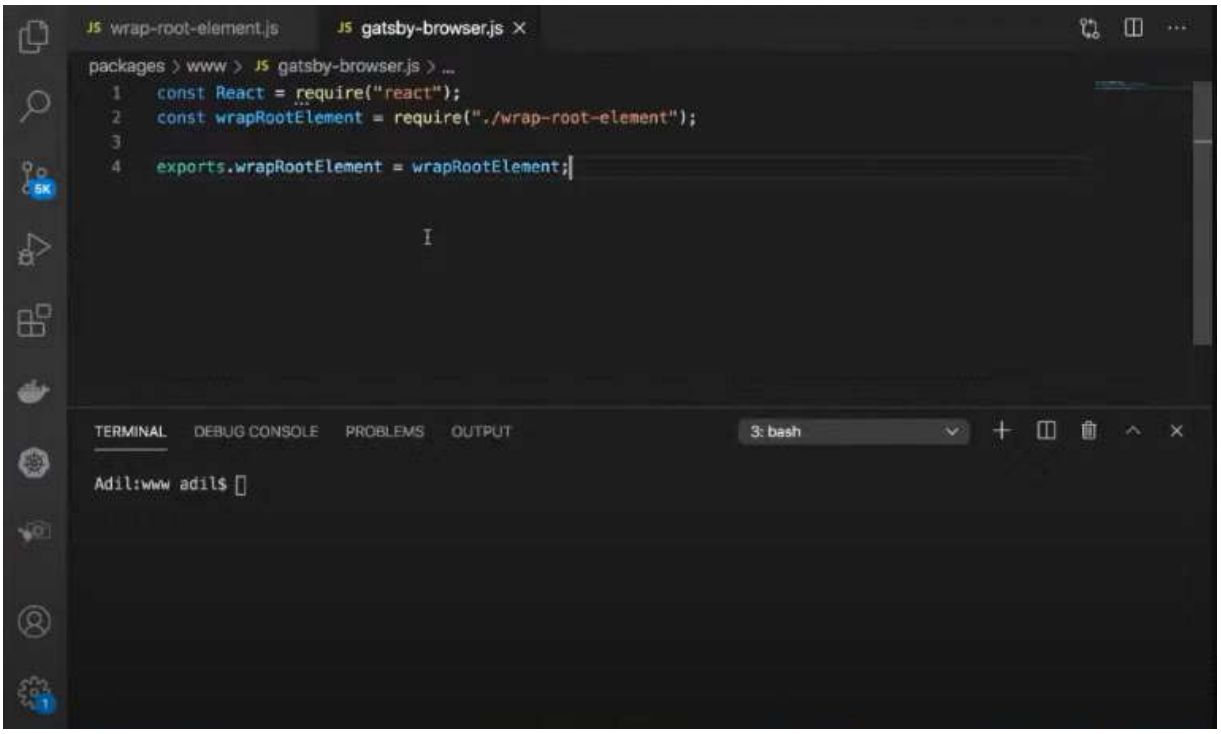


45. Paste the code in gatsby-browser.js

46. Remove the redux and store from the code

47. Add wrapRoote element

48. Make the code in this format

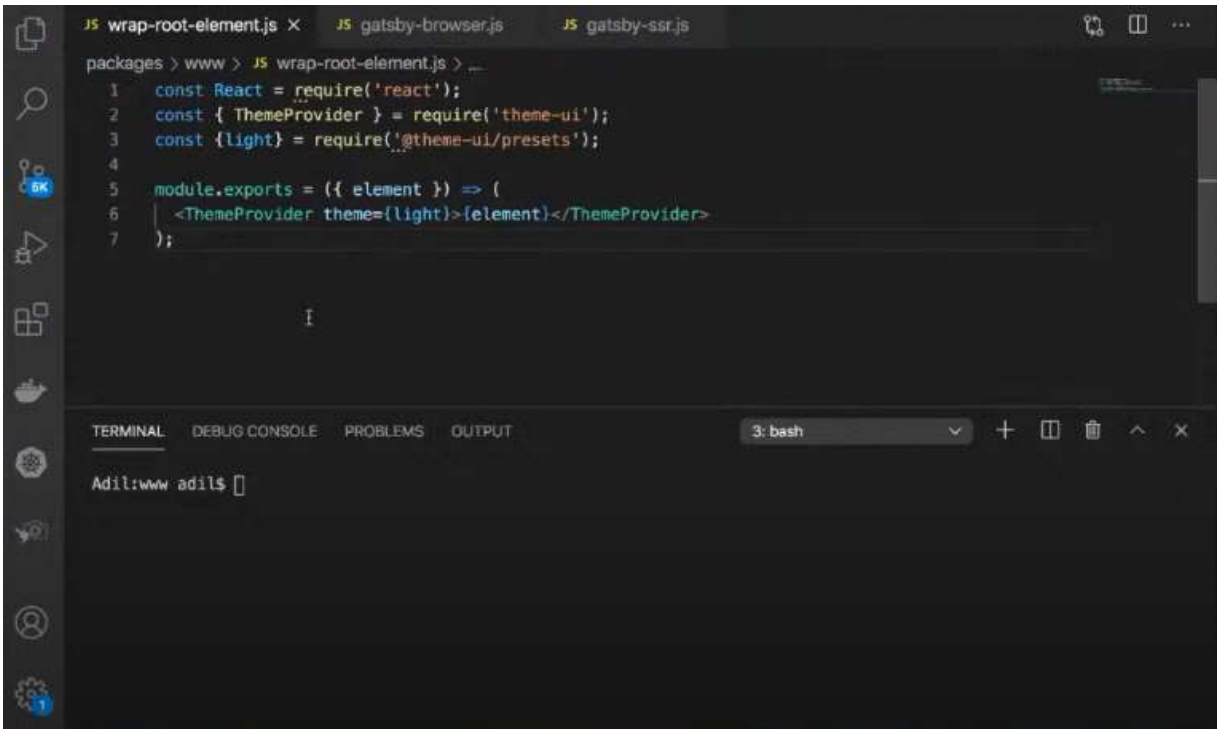


49. Create a new file “gatsby-ssr.js”

50. Copy the gatsby-browser.js code and paste it in gatsby-ssr.js

51. Then go to wrap-root elements code and replace the import with const and from to =

52. Change the export to module.export



53. Then go to terminal in wraproot element and rebuild the Gatsby

54. After rebuilding, go to localhost8000 and check your results

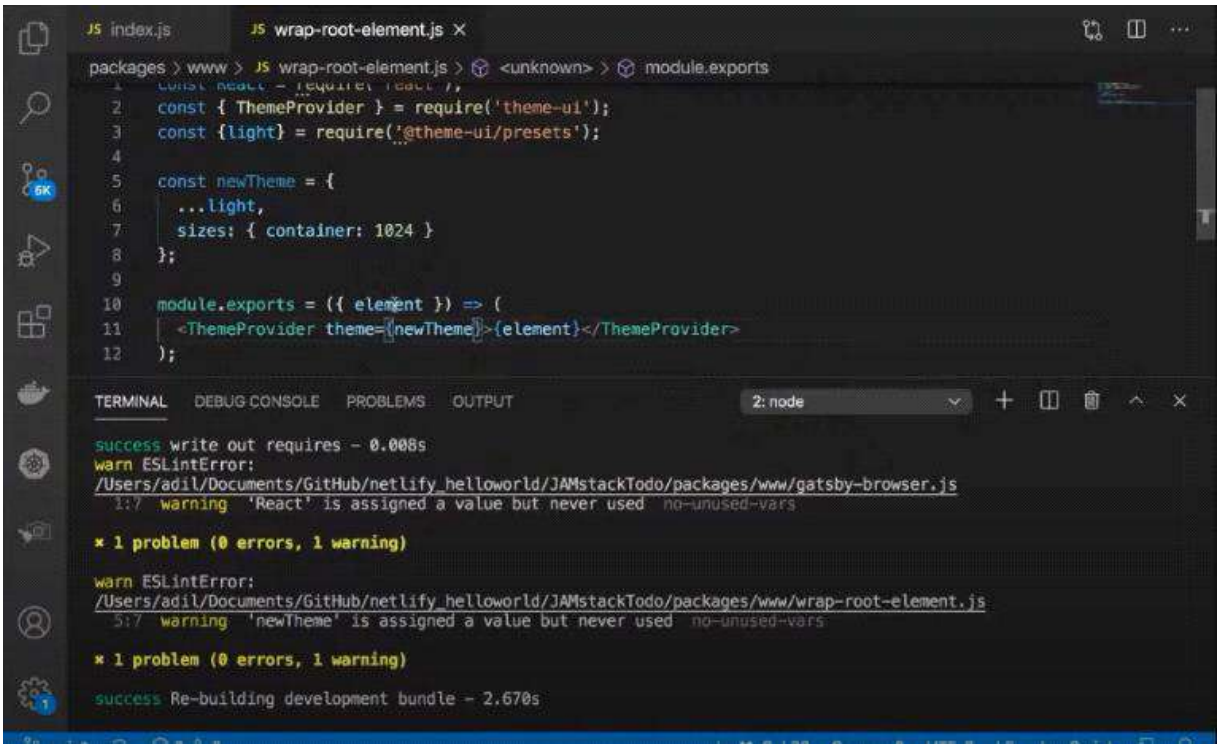
55. Go to index.js

56. Change the div with container

57. Add helloworld between the h1

```
import React from "react"
export default props => (
  <container>
      <h1>Hello World<h1>
  <container>
);
```

58. Go to wraproote element and update the code



59. Then go to “localhost8000” and check your result

60. Go to index.js

61. Update the code

62. Instead of using h1, use headings

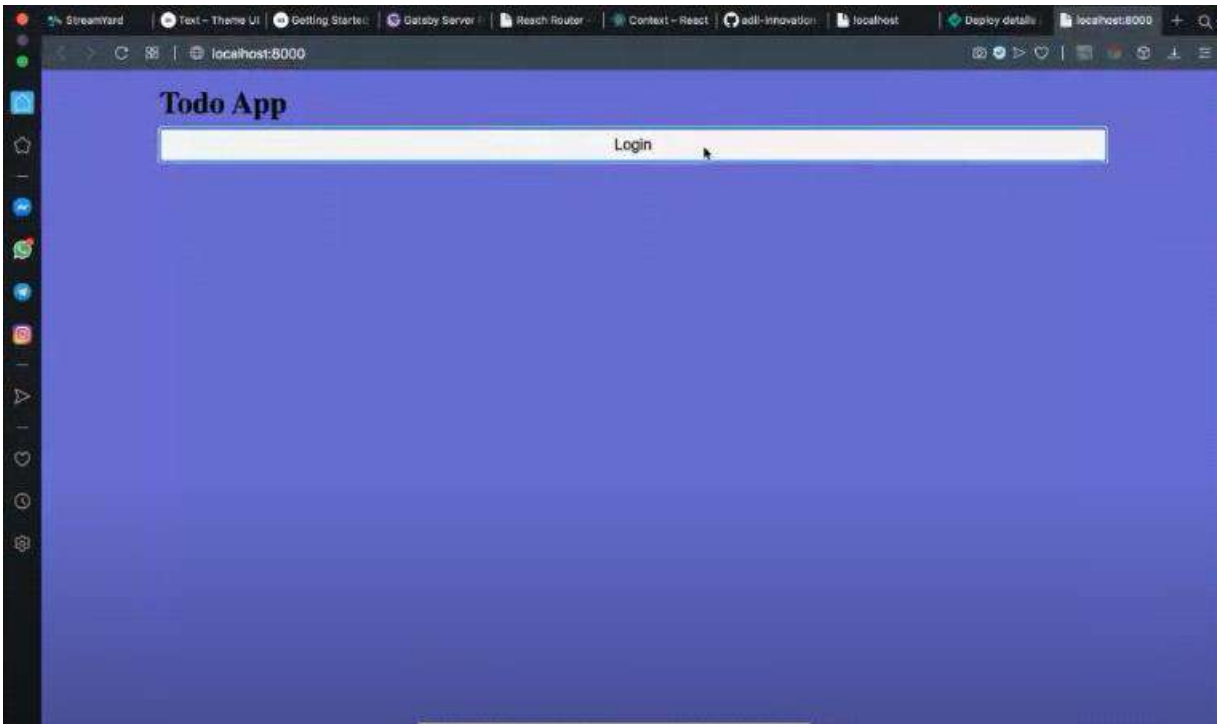
63. Add Buttons

```
import React from "react"

export default props => (

  <container>

      <heading as="h1">Todo App</Heading>

      <Button>Login</Button>

  <container>

);
```

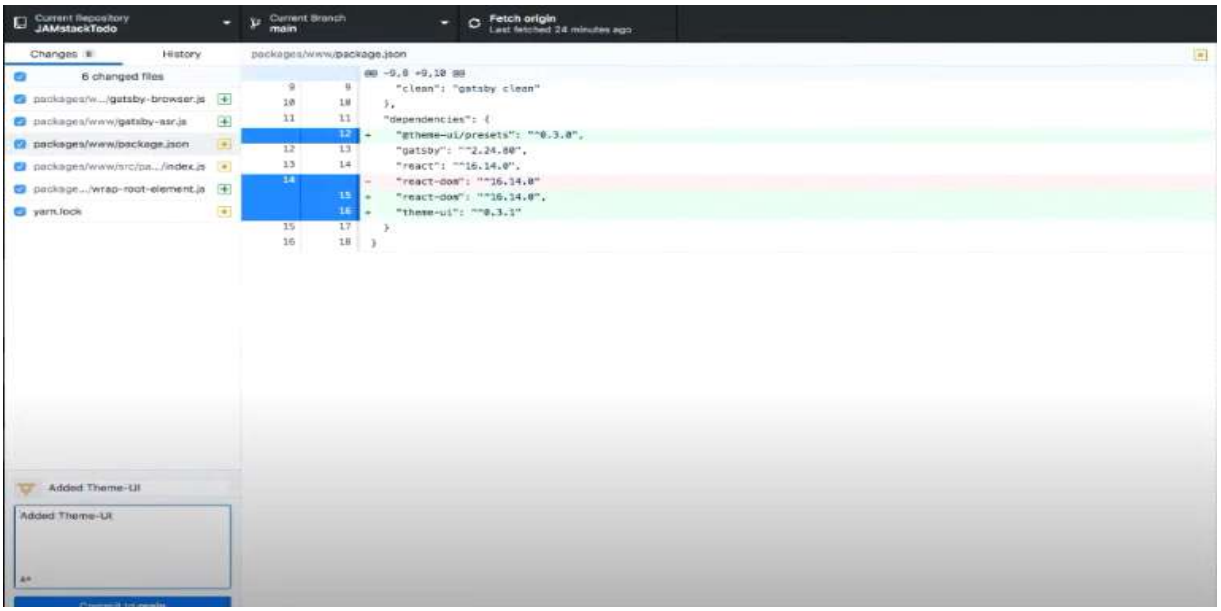
64. Add flex in the code

```
import React from "react"
export default props => (
  <container>
    <Flex sx={{ flexDirection: "column" }}>
      <heading as="h1">Todo App</Heading>
      <Button>Login</Button>
  <container>
);
```

66. The changes will be like



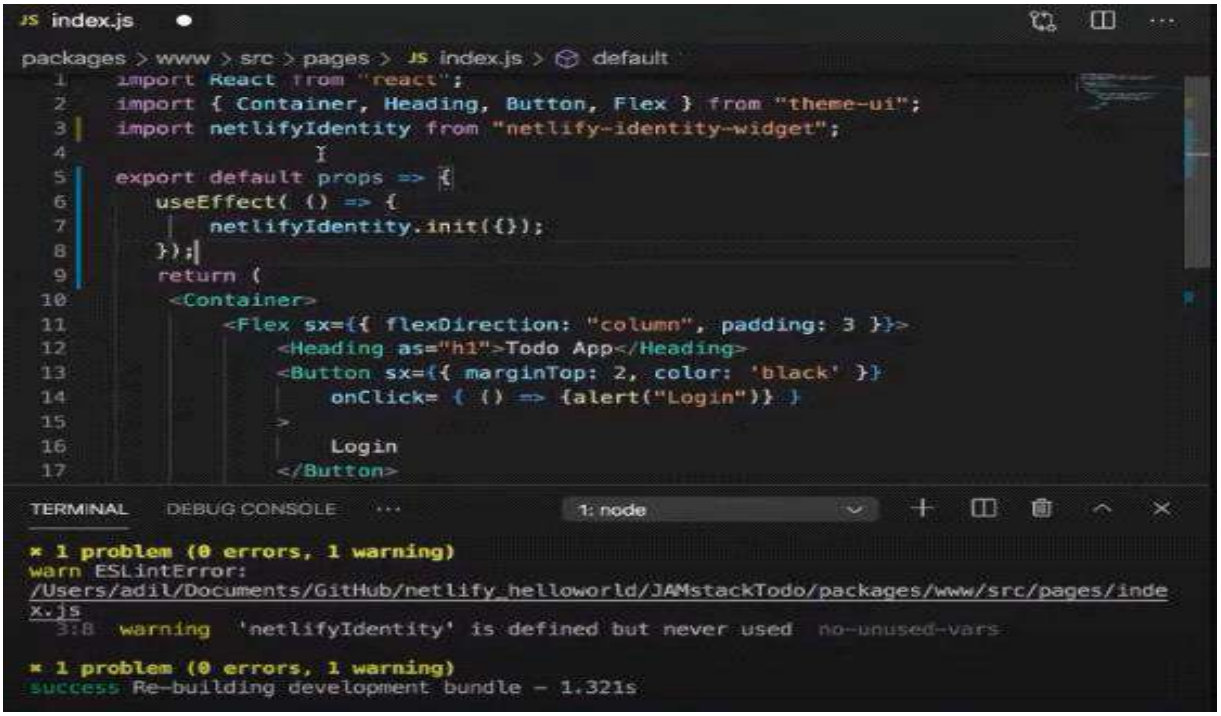
67. Add theme ui and commit to main



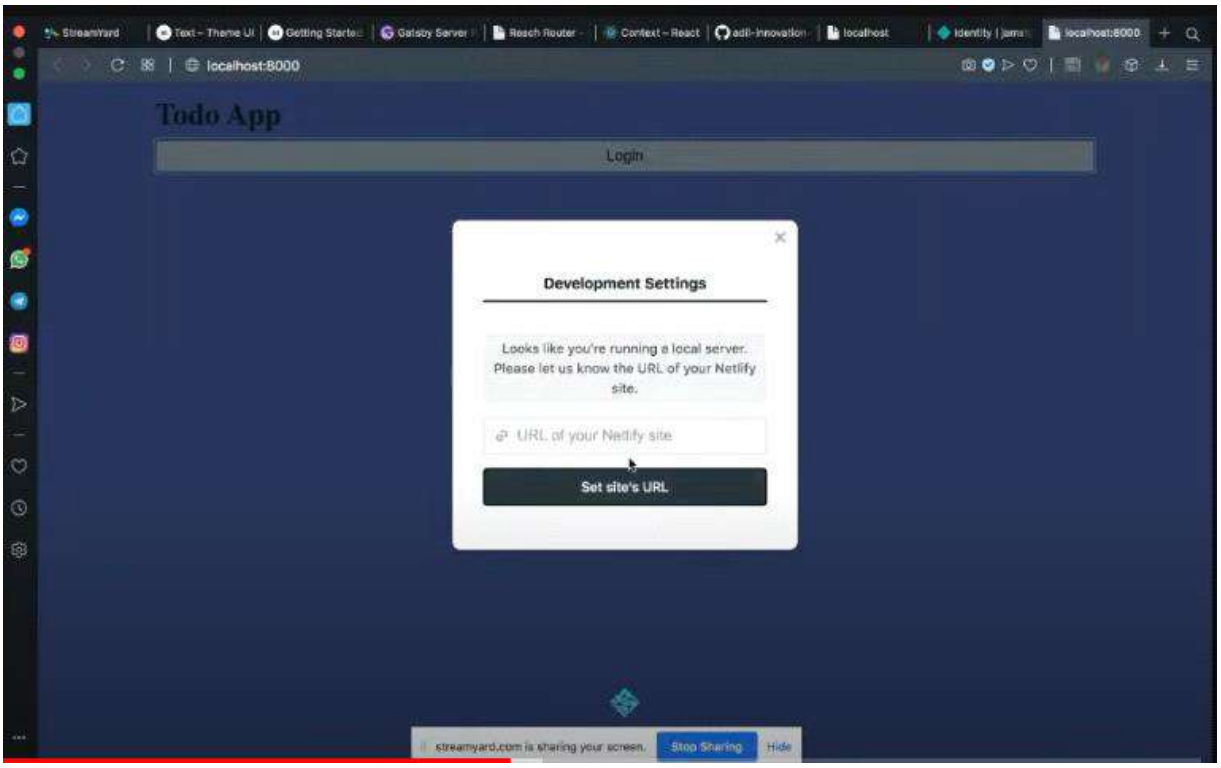
To add authentication

- Go to packages/www terminal and add “yarn add netlify-identity-widget
- Go to index.js and import the netlify identity

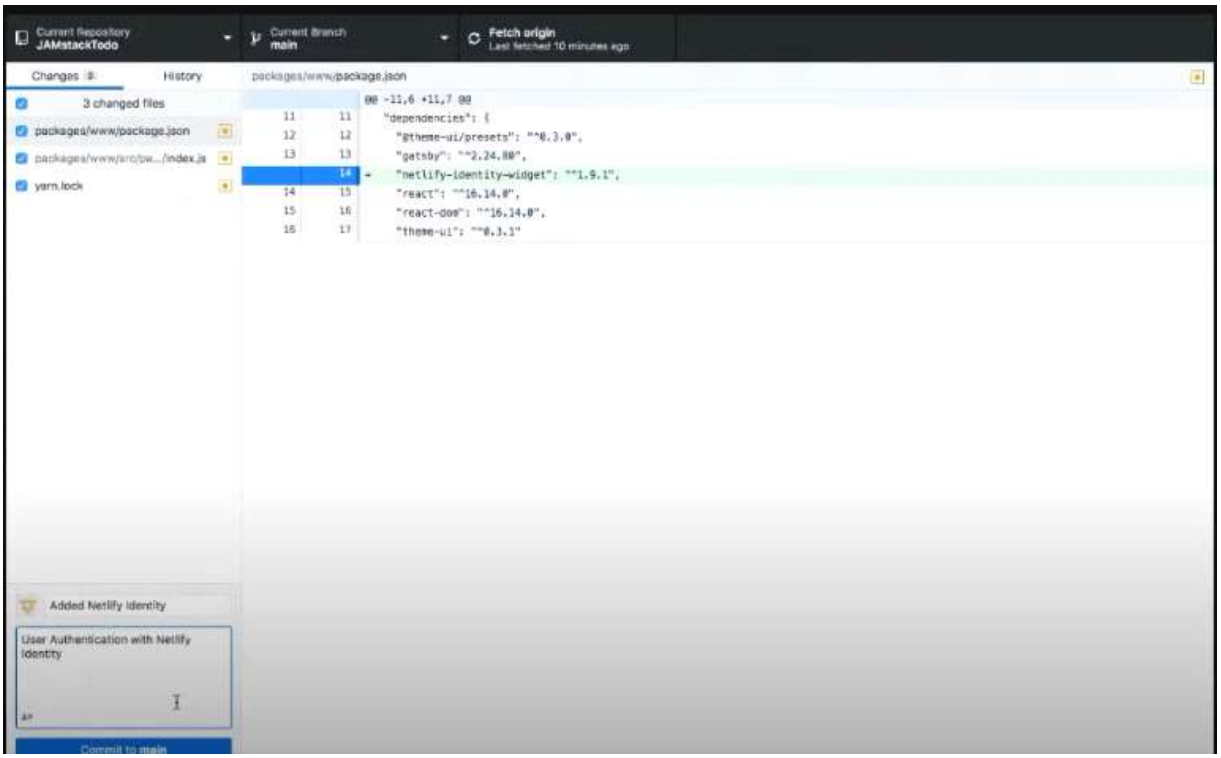
- Update the code
- Add use effect



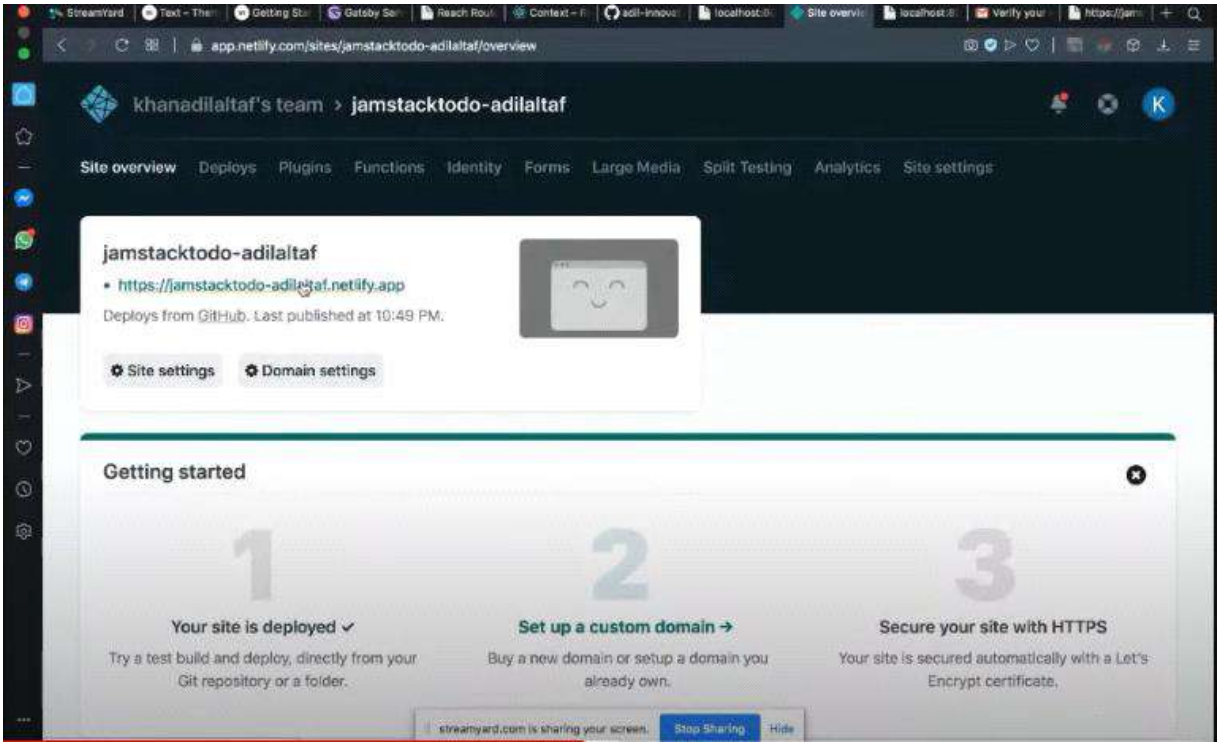
- Go to Netlify
- Go to identity tab
- Enable identity
- Add netlify-identity open in the project code
- After the development go to localhost8000
- Check your login flex
- You will have an login table



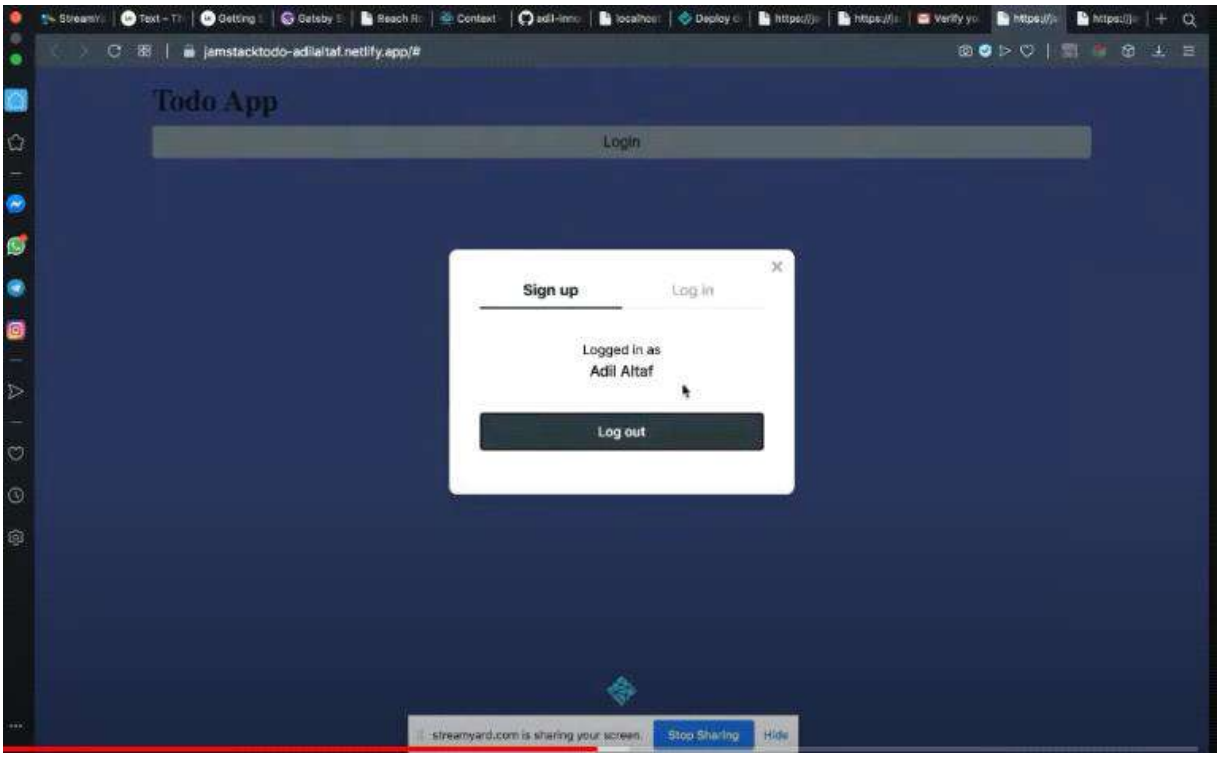
- Go to Netlify identity
- Copy the URL
- Paste the code in in your login box
- After sign up, confirm your email address through the code
- Then push origin



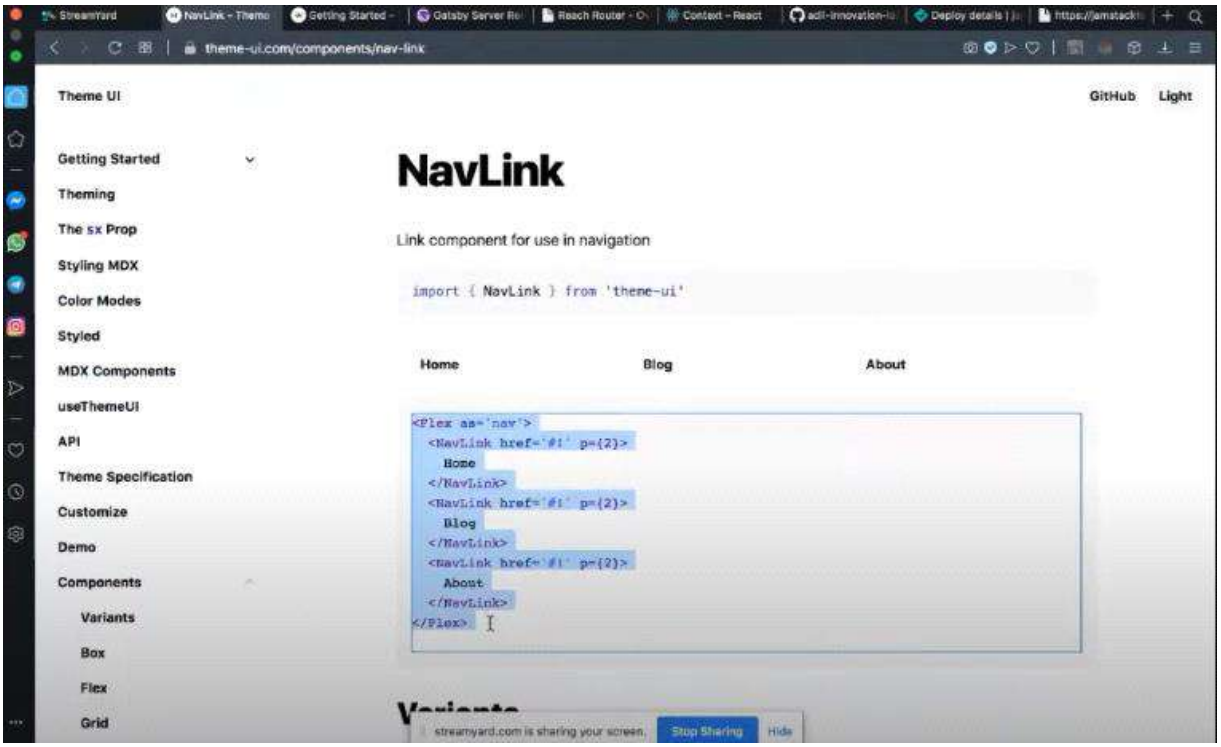
- Go to deploy log in Netlify
- After deployed, go to the sIte link and check the results



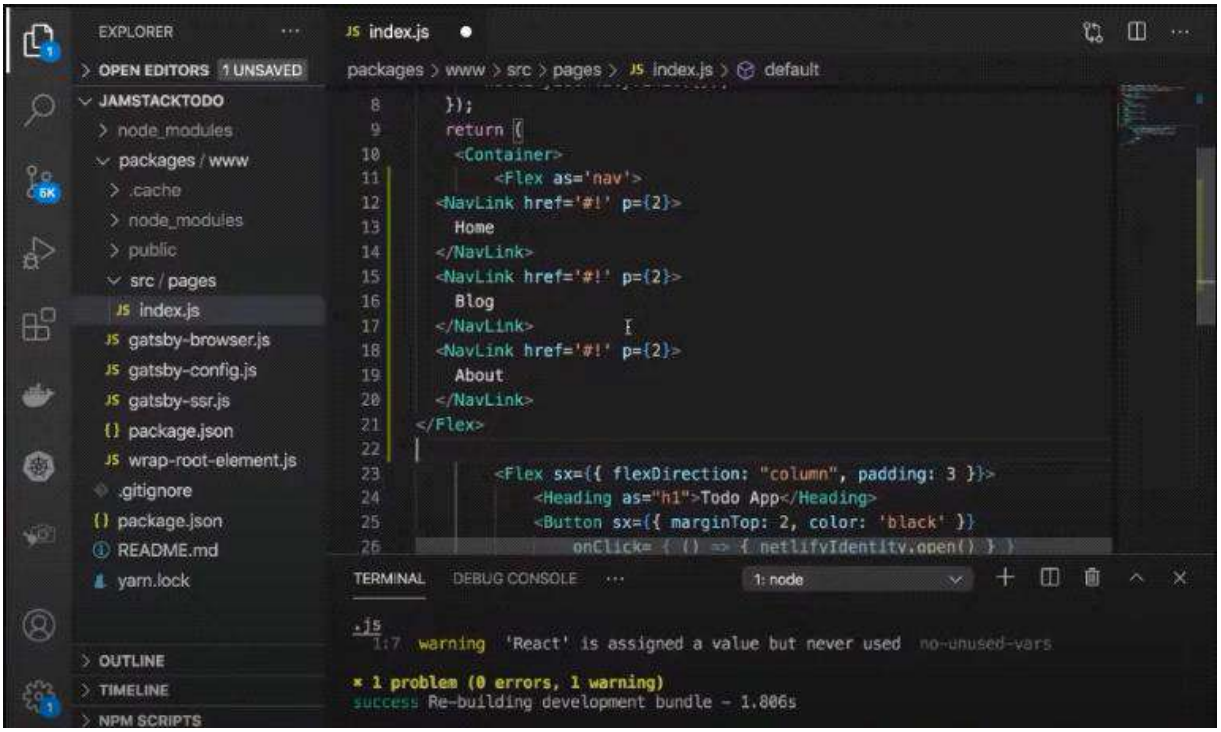
- Then it will be logged in



- Go to theme ui
- Go to Navlink
- Copy the link from the Navlink



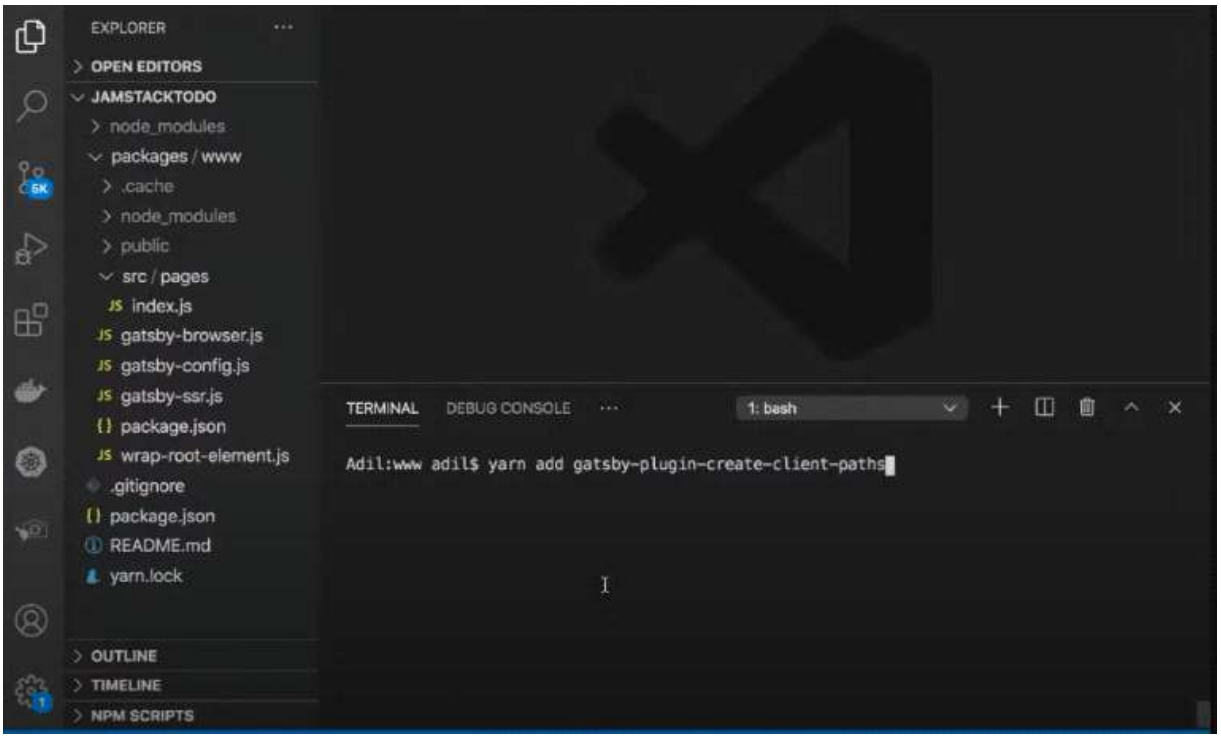
- Copy the link into the container



- Then go to "localhost8000" and check your results
- You can change the names according to your own
- Add the link

- Add the username
- Add state for user

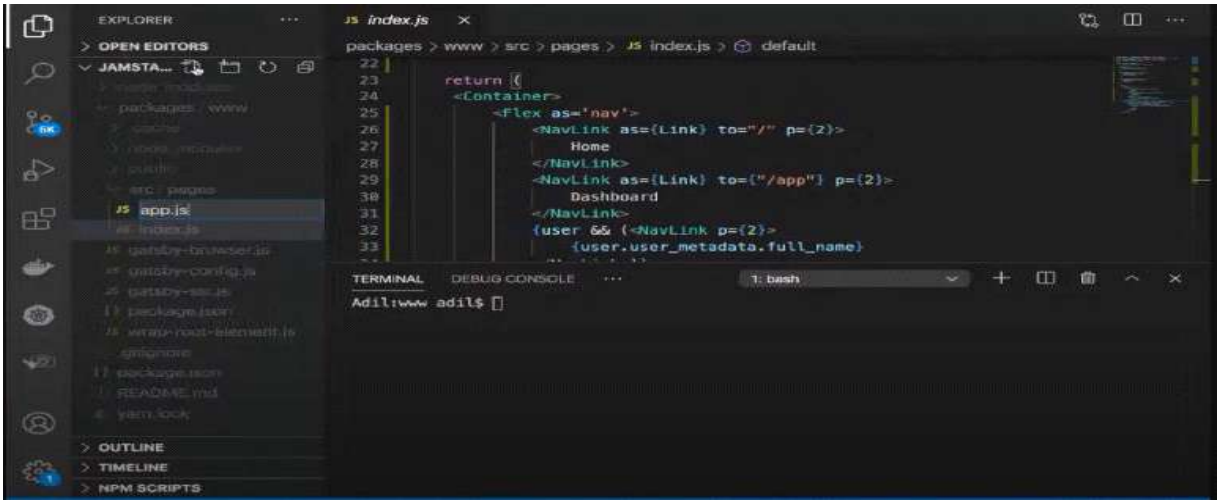
- Add Netlify identity
- Add logout user
- Then go to "localhost8000" and check your updates
- Go to packages/www terminal
- Add “yarn add gtabsy-plugin-create-client-path”



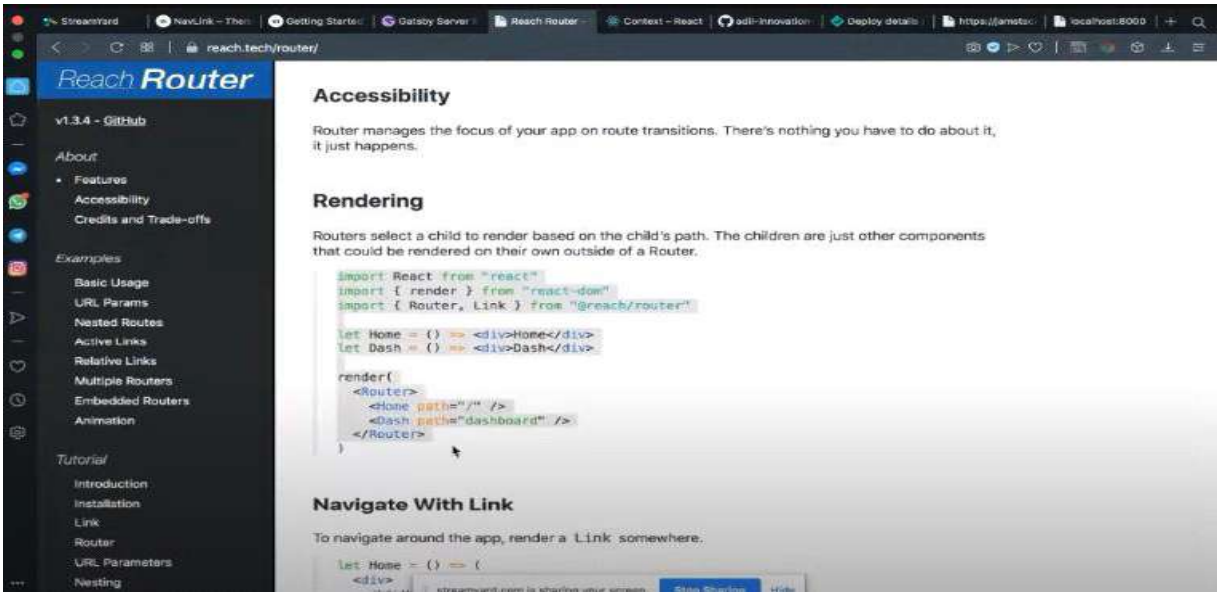
- Go to gatsby-config file
- Add plugins

```
Modeule.exports = {
    Plugins: [
        {
            Resolve: 'gatsb-plugi-create-client-paths',
            Options: { prefixes: [ /app/=  ]}
        }
    ]
};
```

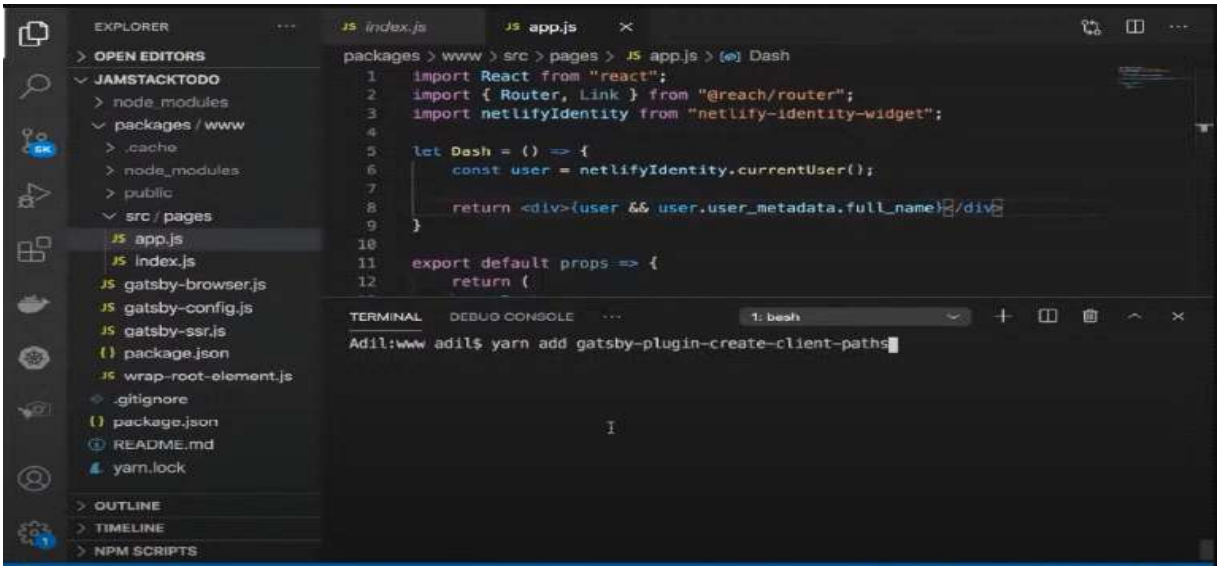
- Add a new file "app.js"



- Inside the app.js, use the router
- Go to reach router
- Use this code



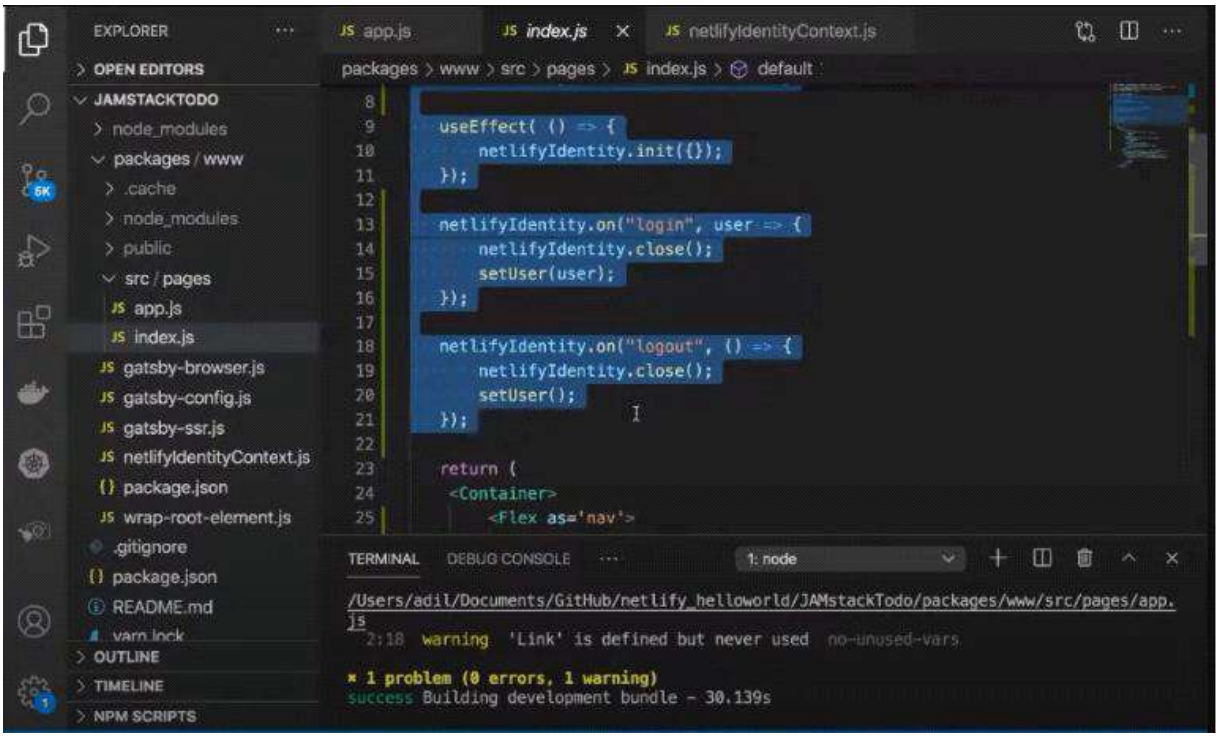
- Use this code in app.js
- Remove react dom
- Remove home from the code
- Add export default props
- Replace the path with app
- Import the Netlify identity
- Add const user



- Run the code
- Then go to dashboard in localhost8000 and check your updates
- To give access to the user all the time
- Create a new file as “netlfiy identity context”
- Inside the Netlify identity context make the code

```
Const React = require(“react”);
Const netlifyodentity = require(“netlify-identity-widget”);

Const IdentityContext = React.createContext({}):

Exports.provider = props => {
  Return {
    <IdentityContext. Provider alue=[{identity :nrtlifyIdentity}>
      {props.children}
    </identitycontext.provider>
  }
}
```

- Then go to index.js and copy this code



- Use this code in netlify identity context
- Then input the context in wraproote element

```
JS app.js    JS netlifyIdentityContext.js    JS wrap-root-element.js ×
packages > www > JS wrap-root-element.js > ...
1   const React = require('react');
2   const { ThemeProvider } = require('theme-ui');
3   const {light} = require('@theme-ui/presets');
4   const {Provider} = require("./netlifyIdentityContext");
5
6   const newTheme = {
7     ...light,
8     sizes: { container: 1024 }
9   };
10
11  module.exports = ({ element }) => (
12    <ThemeProvider theme={newTheme}>{element}</ThemeProvider>
13  );
```

```
TERMINAL   DEBUG CONSOLE   PROBLEMS   OUTPUT                    1: node
✖ 1 problem (0 errors, 1 warning)
success Building development bundle - 30.139s
success onPreExtractQueries - 0.034s
success extract queries from components - 0.177s
success write out requires - 0.018s
i Re-building development bundle
```

Session-40

Project 4

Building a Bookmarking application

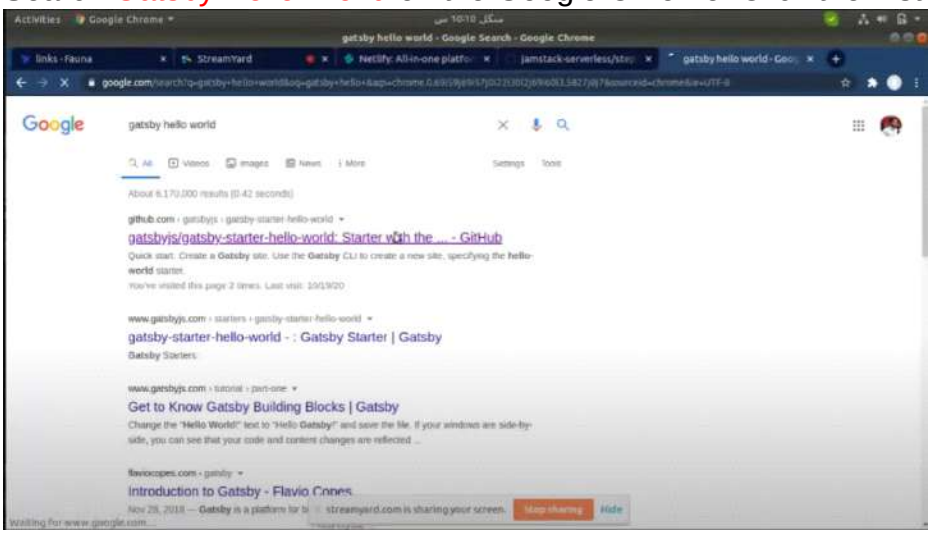
What is bookmarking application?

Bookmarking is an advanced type of Diary which helps the user to store the important things for example different links and daily routines etc. All of this information is stored in the cloud which enables the user to access this across different platforms.

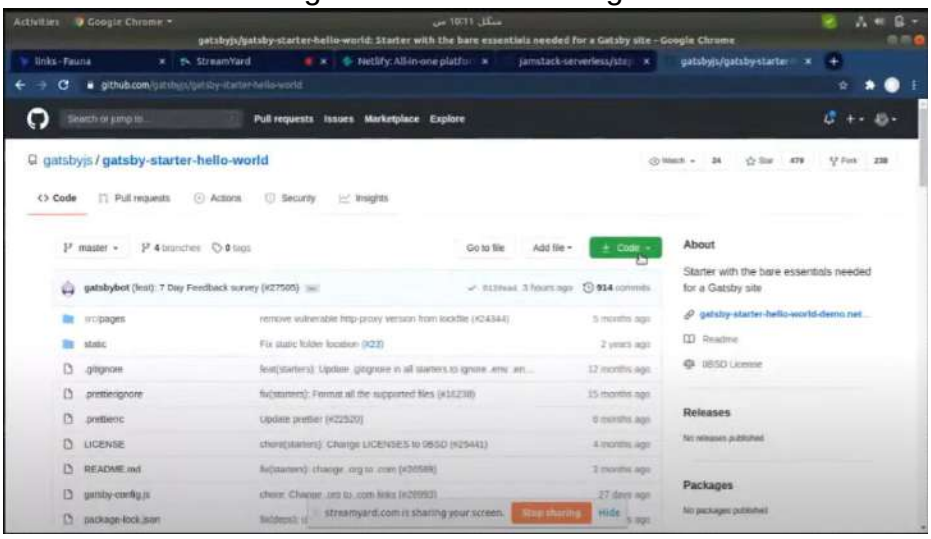
Fauna DB Graph QL and Gatsby will be used for this process.

Steps

- Search Gatsby Hello World on the Google Chrome. Click the first link as shown in the Figure.



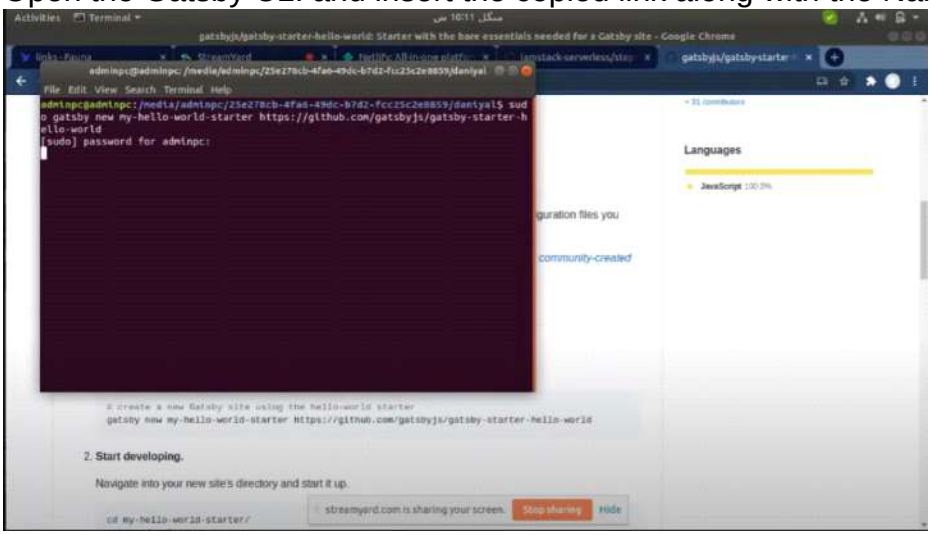
- Click Code on the Page as shown in the Figure



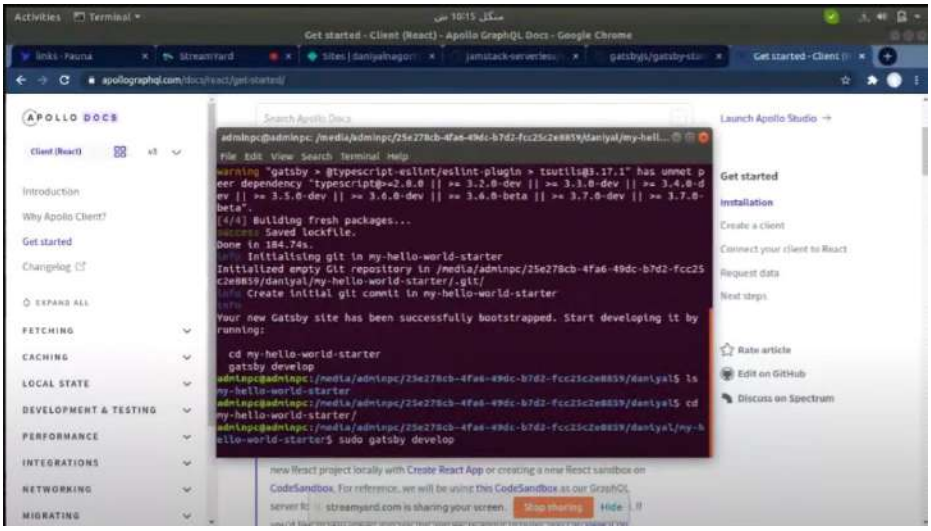
- For creation of the Boiler plate, copy the following command

gatsby new my-hello-world-starter https://github.com/gatsby-starter-hello-world

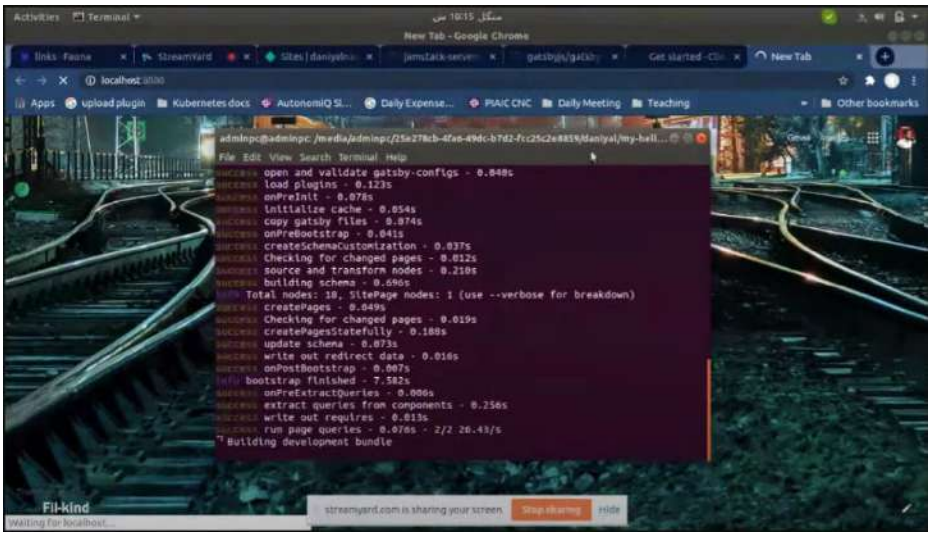
- Open the Gatsby CLI and insert the copied link along with the Name of the Project

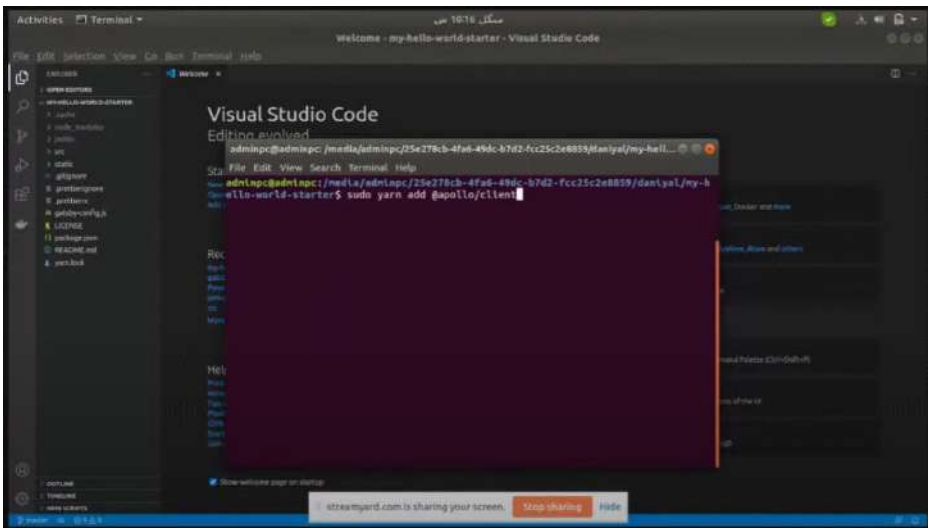


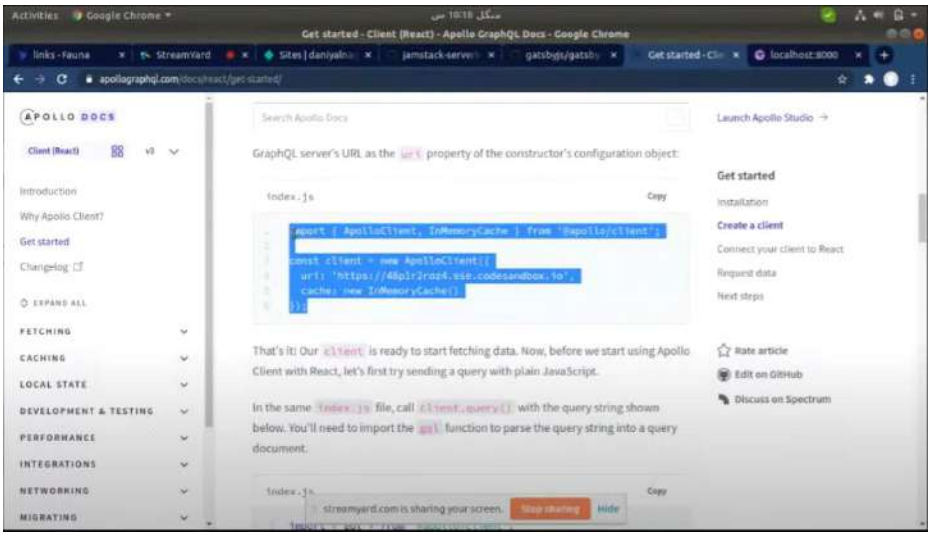
- Write the code sudo gatsby develop in the CLI.

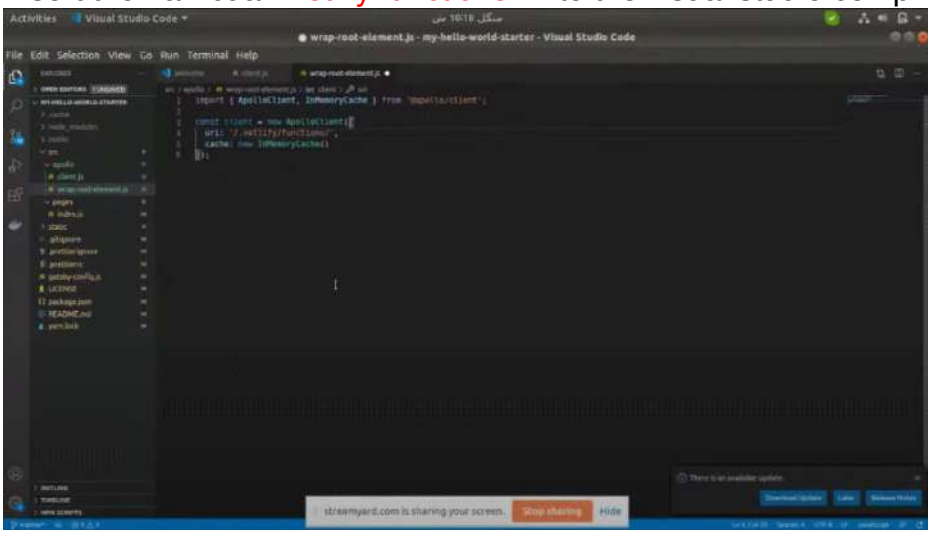


- After this program has run successfully, type local host:8000 in the browser's search as shown



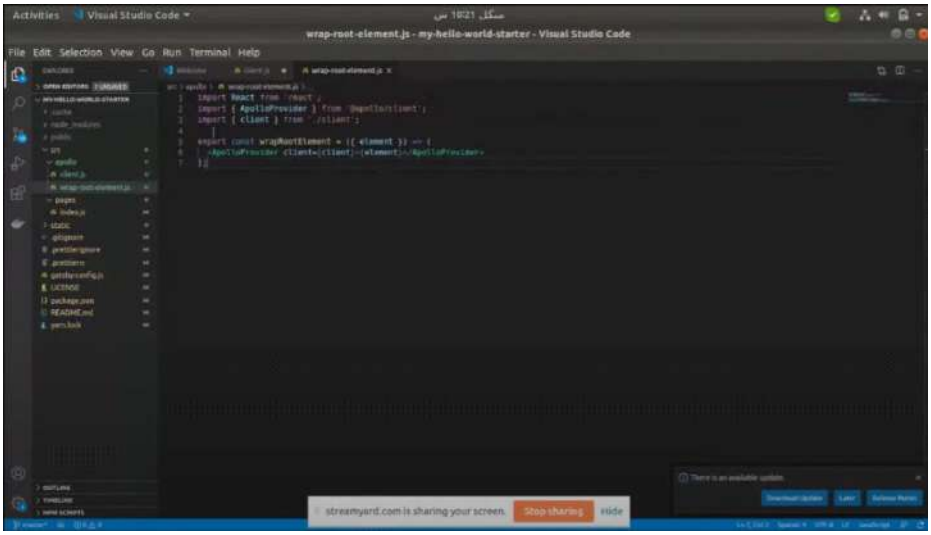
- In order to install the dependencies, Type sudo yarn add @apollo/client in the CLI.
- 

- After it copy the client.js link from the official site of apollo
- 

- Insert the Lambda /.netlify/functions/' into the visual studio compiler as shown


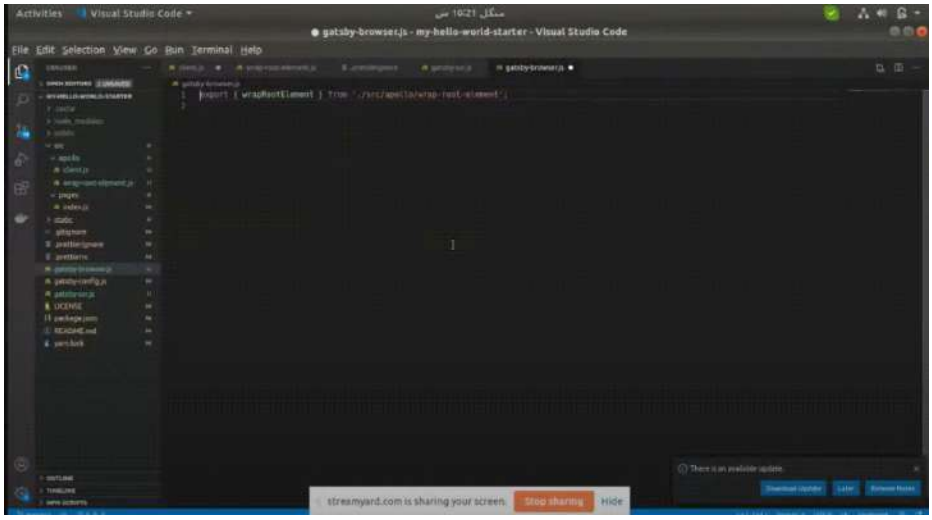
- Open the CLI and insert the command sudo netlify function:create bookmark

- Copy the apollo provider for rooting the client into the application and call the client into the main apollo provision file.

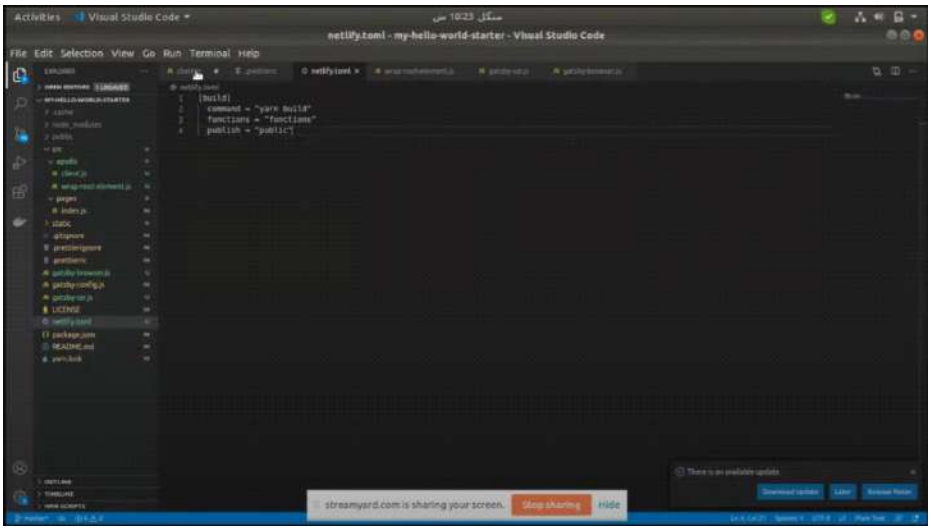


- Make a new program and insert the wrap root element in it which will complete the program.

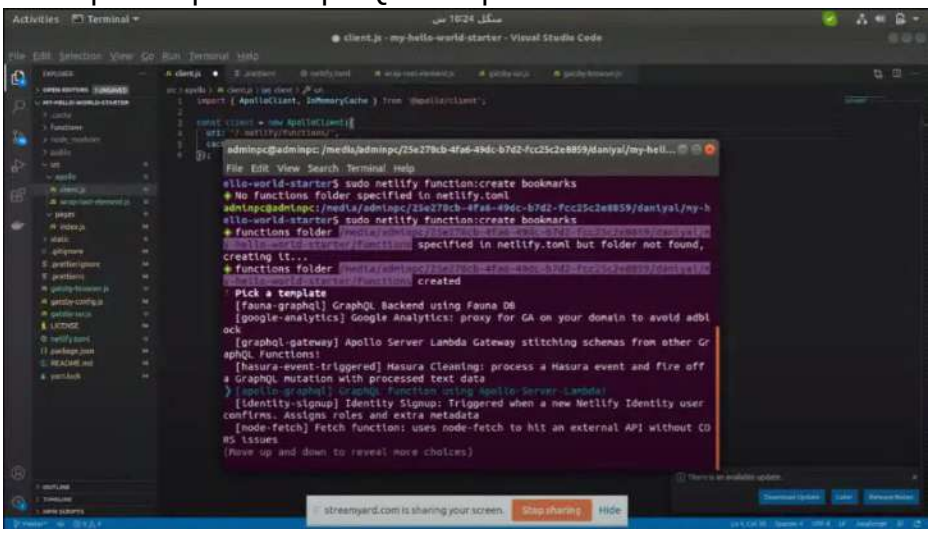
As export { wrapRootElement } from ‘./src/apollo/wrap-root element';



- Make a new file netlify.toml and paste the full [build] command as shown in the figure

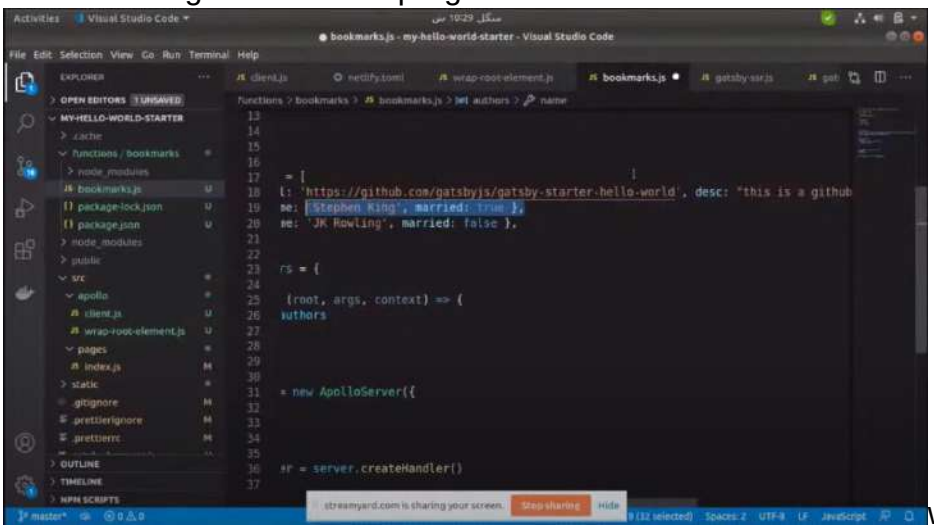
- 

- Pick up the Apollo Graph QL Template from the CLI command as reflected below



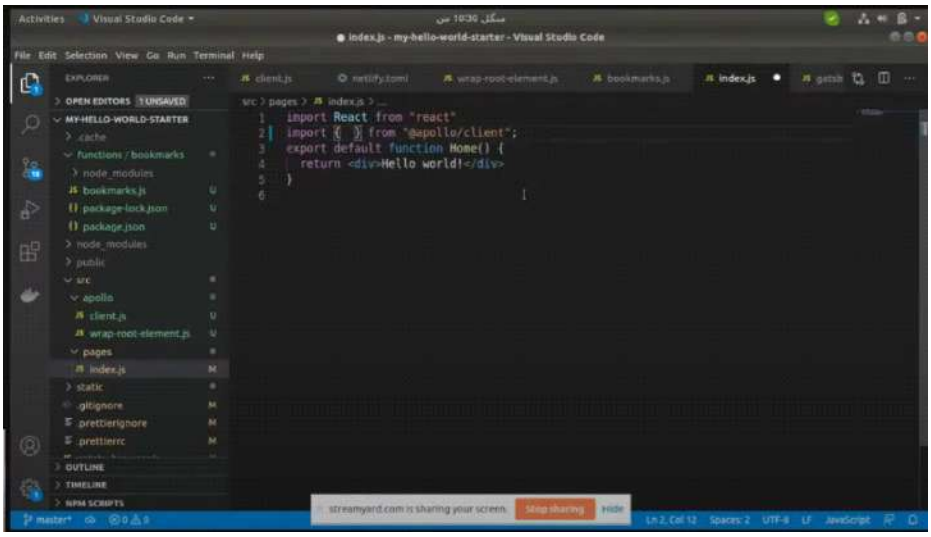
- Modify the bookmark file by adding ID access, URL and description
- Copy the URL of the target website. Say if it is Gatsby official then paste the link of the website as follows
- https://www.gatsbyjs.com/

- Paste the target URL in the program

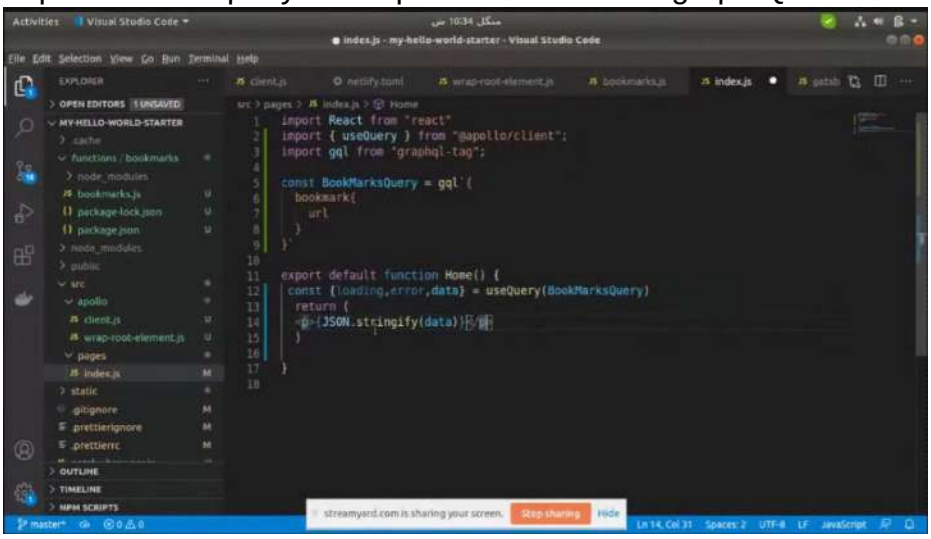


ws

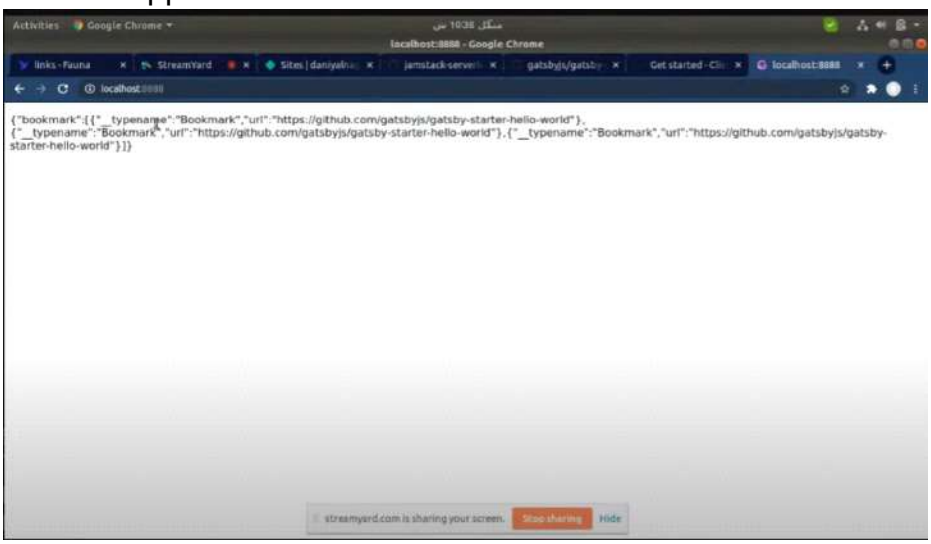
- In the index.js file, call the apollo by typing @apollo/client in the compiler along with import tag and graph QL tag



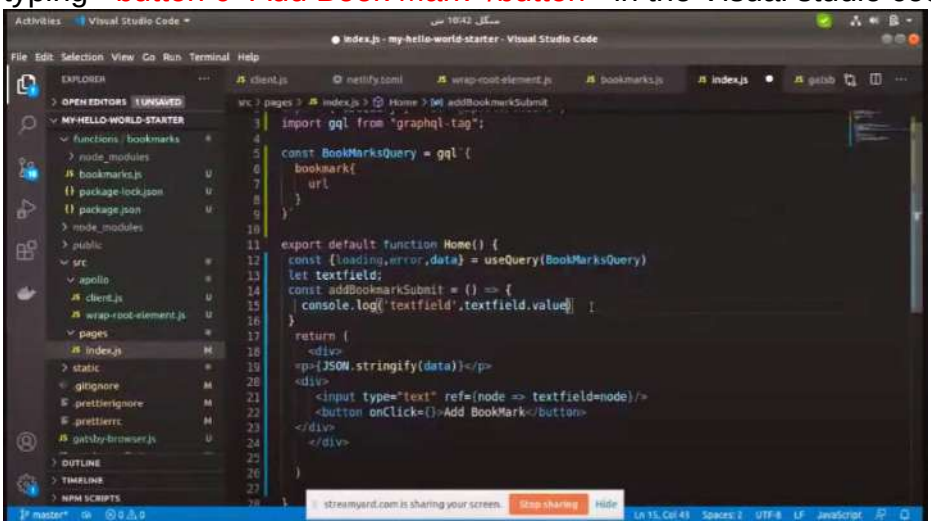
- Import the use query from apollo client and the graph QL as shown below to create a query.



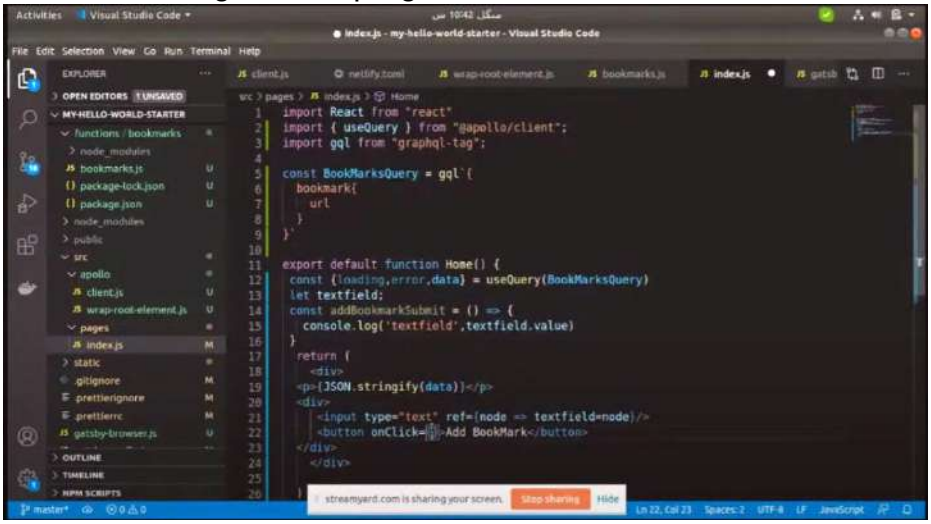
- Run this application on the local host:8888 to check if it runs correctly



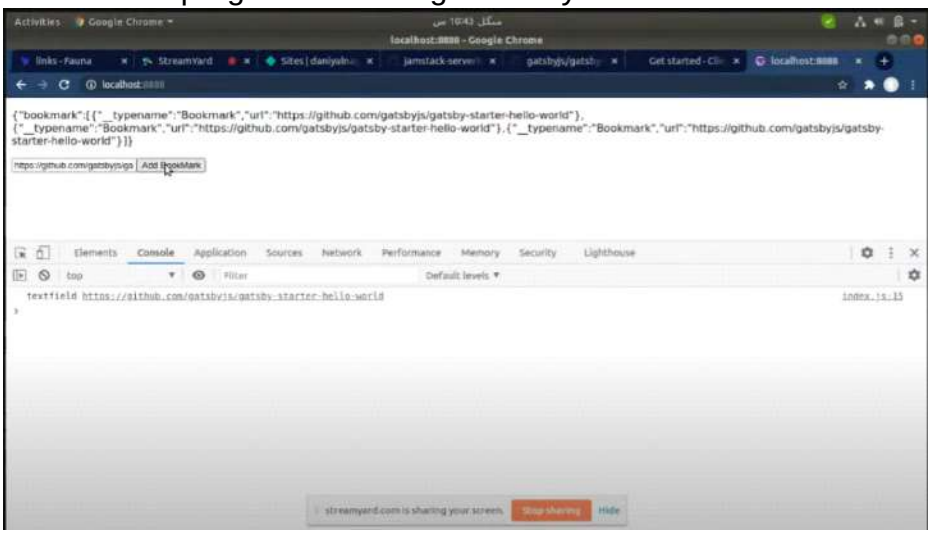
- Input type, reference and node to assign a new field to the program. Also create a button by typing <button 0>Add Book Mark</button> in the Visual studio code-index.js file



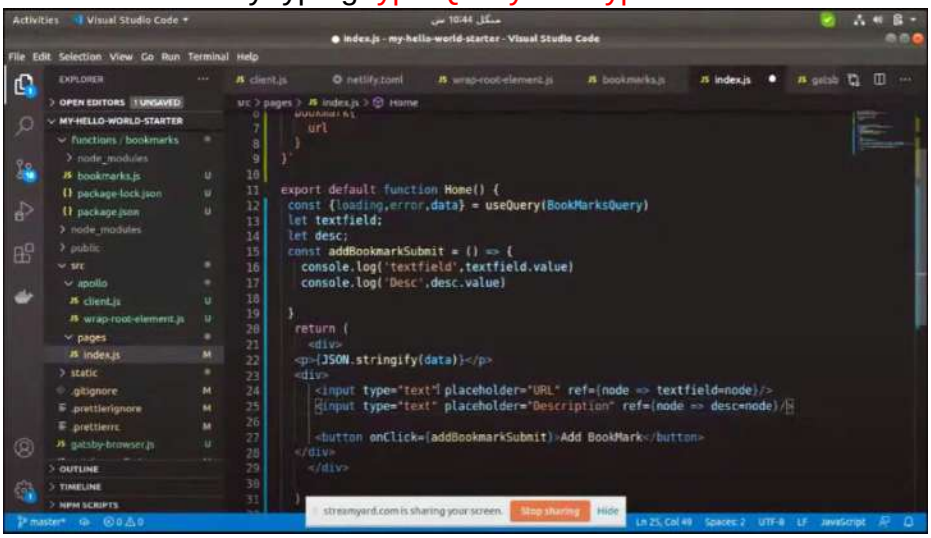
- Add console.log into the program and call the function of addBookMarkSubmit as follows



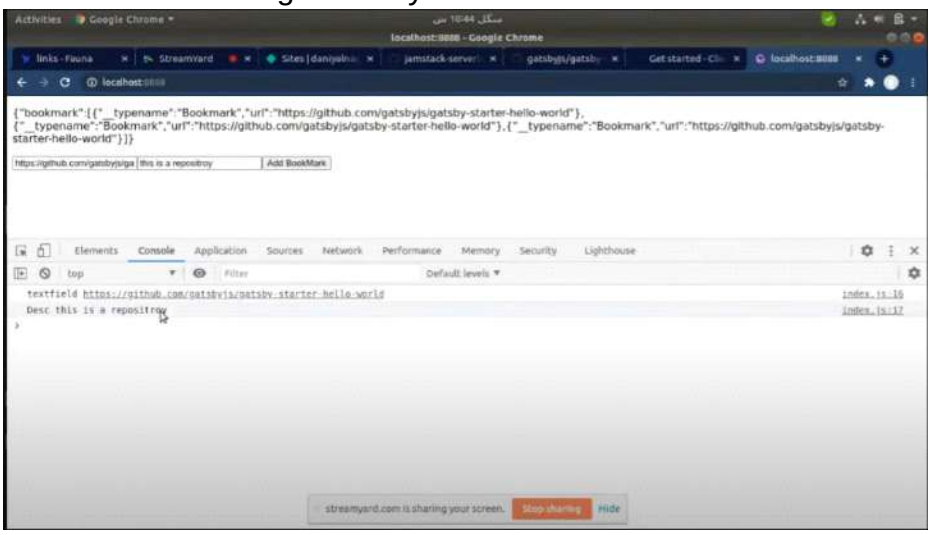
- Check if the program is running correctly.



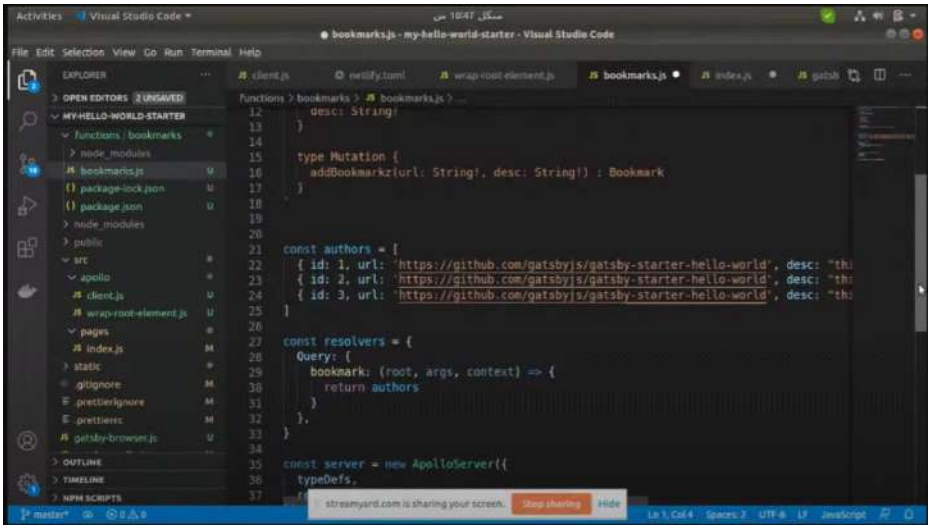
- Create another field for description by building a variable
- Build a variable by typing type:Query and type:Author as reflected below


index.js - my-hello-world-starter - Visual Studio Code
export default function Home() {
const {loading,error,data} = useQuery(BookMarksQuery)
let textfield;
let desc;
const addBookmarkSubmit = () => {
console.log('textfield',textfield.value)
console.log('Desc',desc.value)
return (
{JSON.stringify(data)}
<input type="text" placeholder="URL" ref={node => textfield=node}/>
<input type="text" placeholder="Description" ref={node => desc=node}/>
<button onClick={addBookmarkSubmit}>Add BookMark</button>

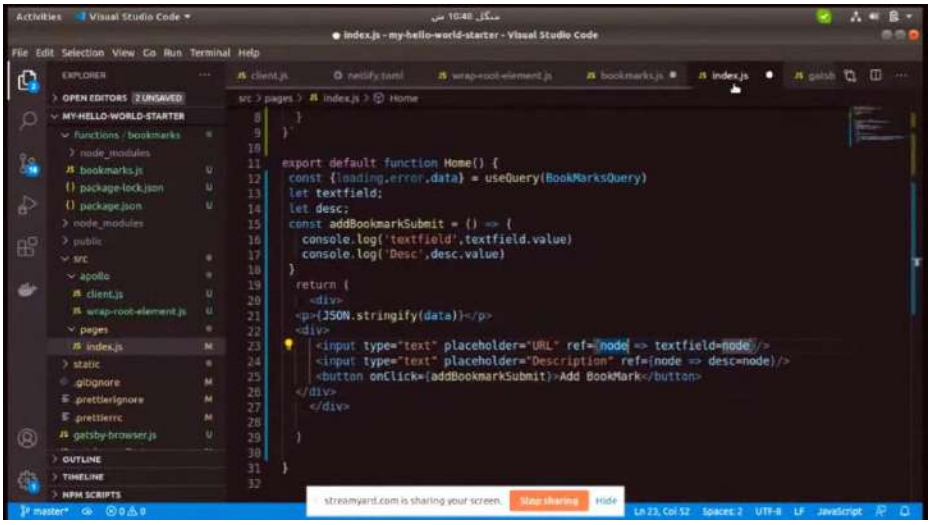
- Check if it is working correctly



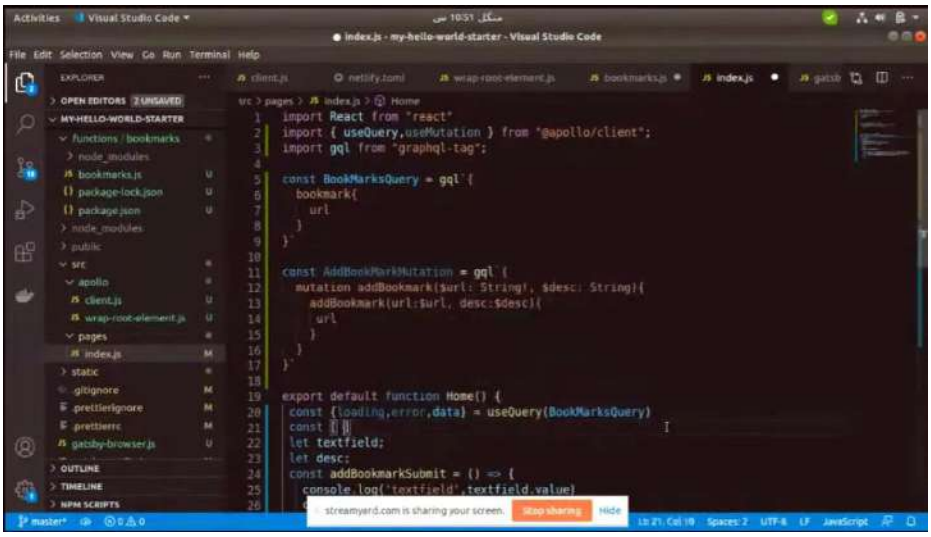
- Insert the command type mutation{add bookmark} (url: string!, desc: string!): bookmark in the program as shown



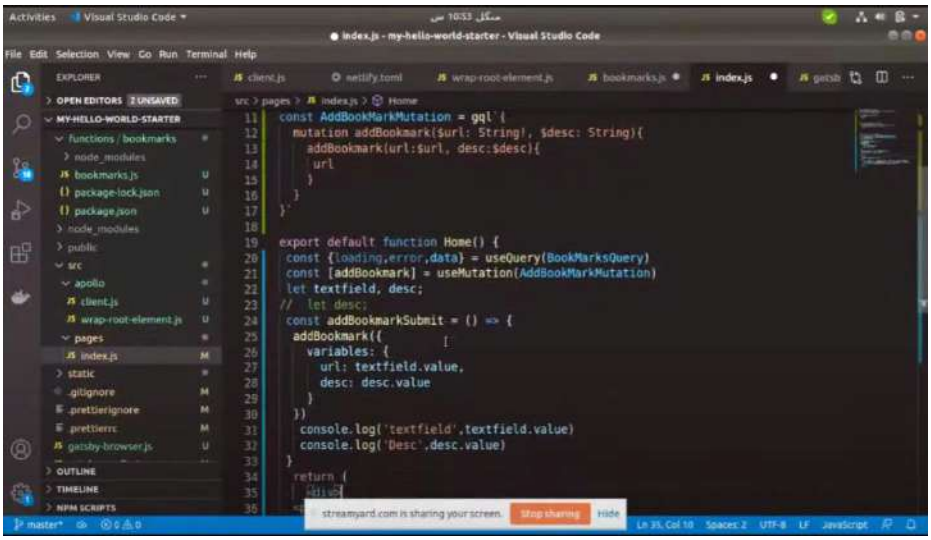
- Use conslole.log in the mutation to test the proper connection with the cloud.



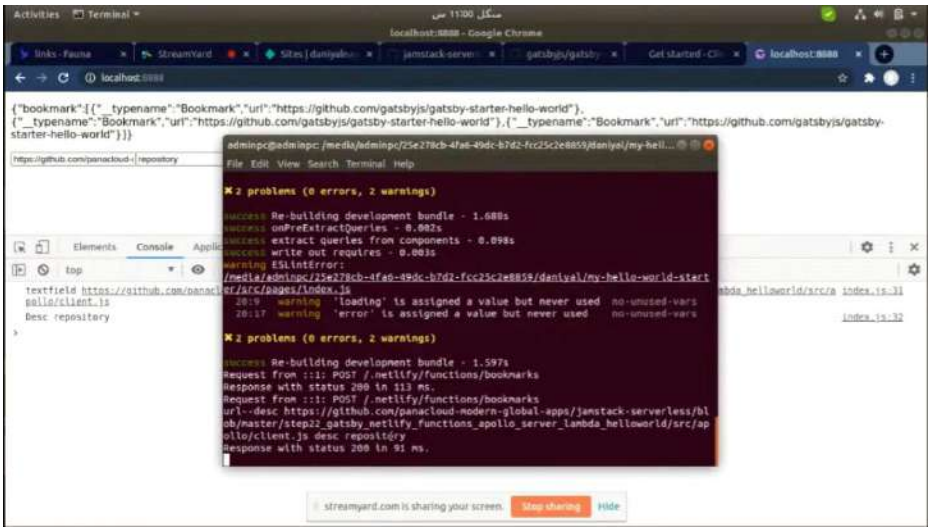
- Insert the command const AddBookarkMutation = gql'{ mutation addBookMark($url: string!, $desc: String)} into the index program



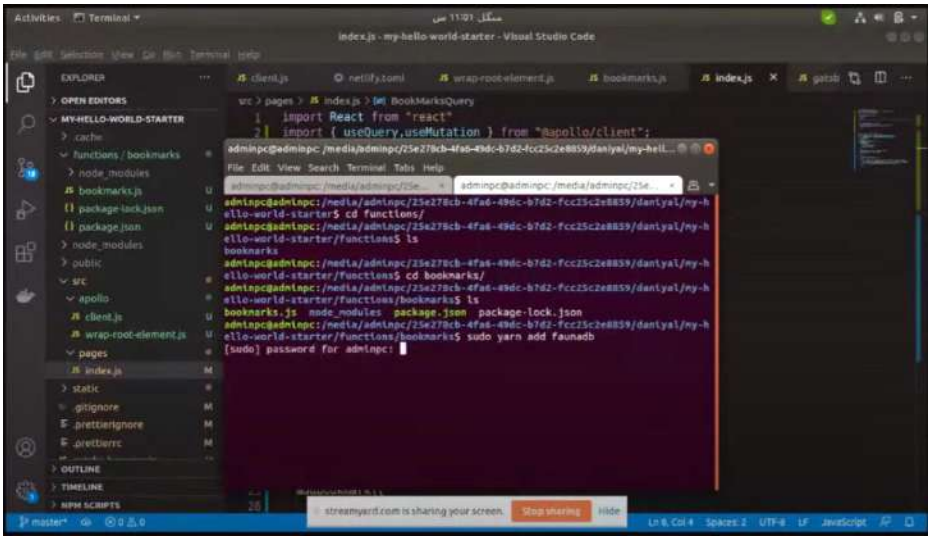
- Insert the variables as reflected below



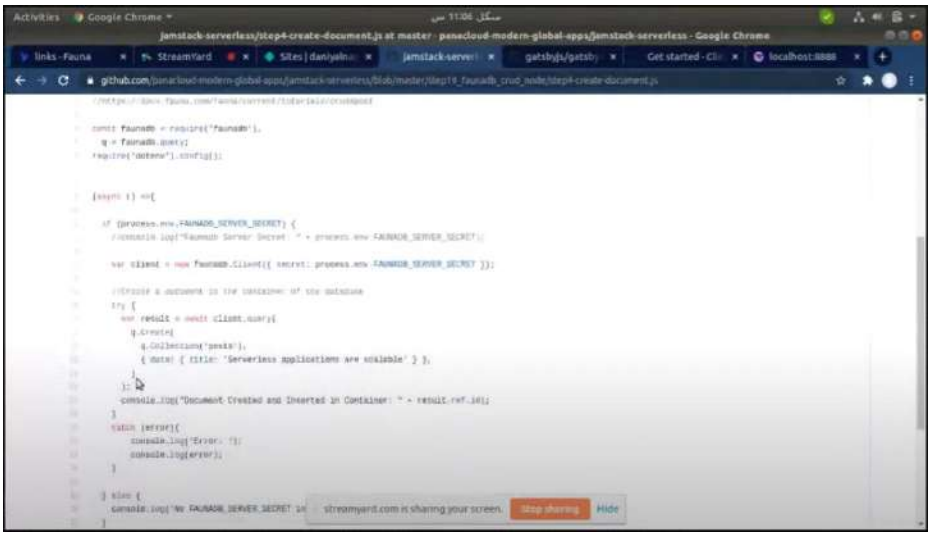
- Run the program to see if it runs perfectly.



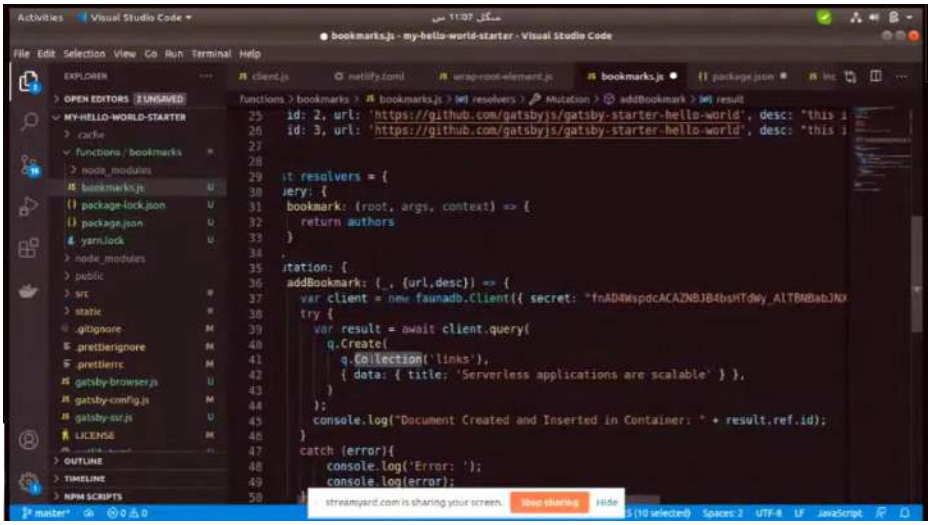
- Type sudo yarn add faunadb in the CLI to install Fauna DB into the program.



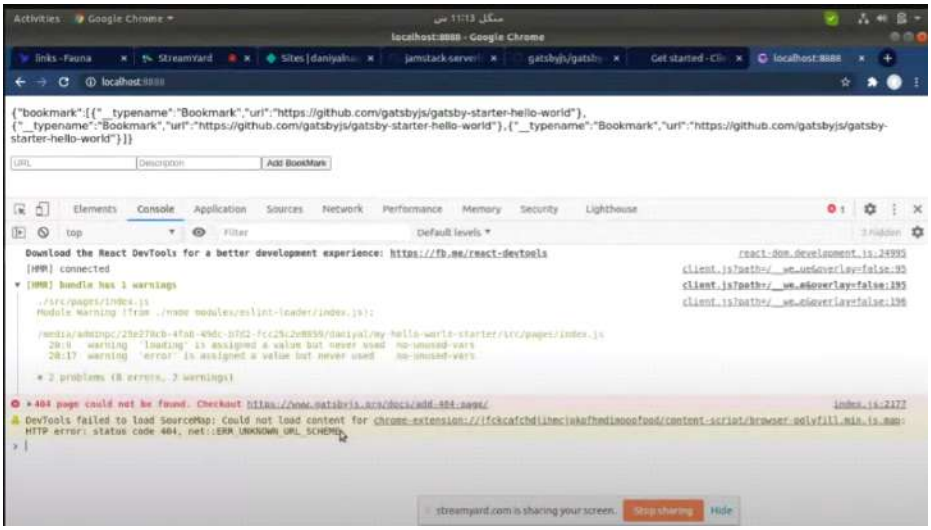
- Copy the document code from Fauna DB website and paste it in the program



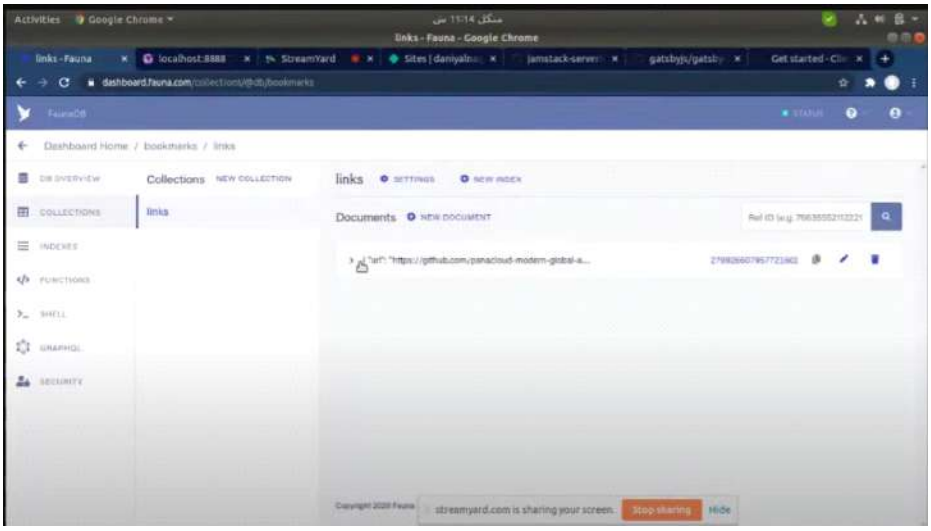
- Re assemble it in accordance with your servers and pass the tokens to the visual studio file as shown below in the figure



- Deploy JASN in the program and check if it runs correctly.



- Test if the information is present on the Fauna DB cloud as well to ensure the links are working as it is supposed to

- 

Congratulations, the bookmark application has been successfully created in the above steps.

Session-41

Project 3

Advanced Full stack serverless project development

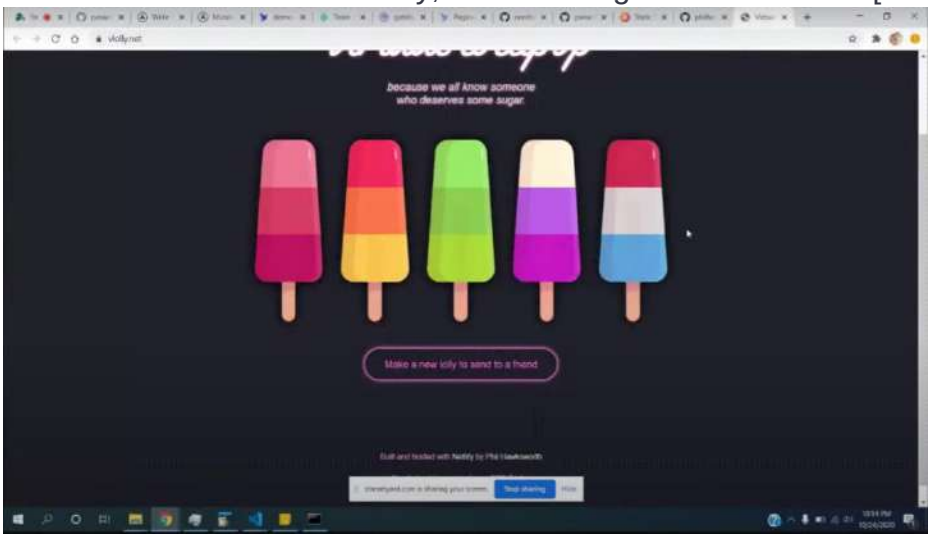
Creation Of Virtual Lolly

What Is a Virtual Lolly?

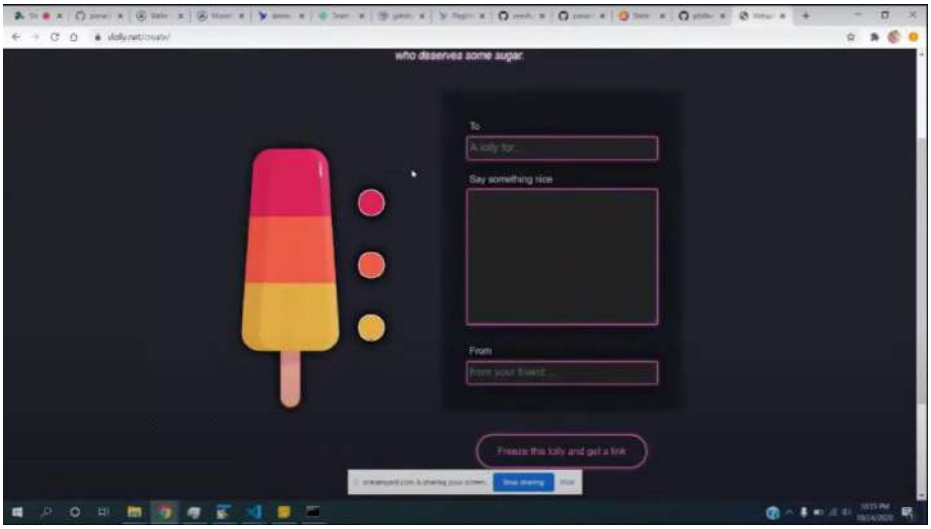
A virtual lolly is a way to express one's feelings to others online in form of lollipops. Anyone can access virtual lollies and send them to their near and dear ones. These are aided by textual and visual information which is in form of a surprise.

How to create a virtual lolly?

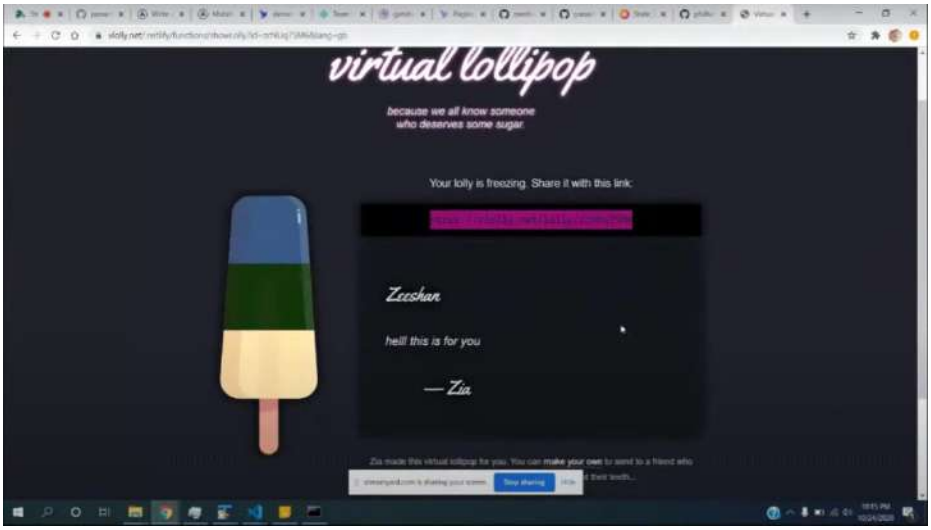
- In order to create a virtual lolly, we have to go on the website [vlolly.net].



- There we have an option for the creation of a lolly.
- Now we have the controls for the creation of a virtual lolly including, naming, coloring, tagging information and name of the sender.



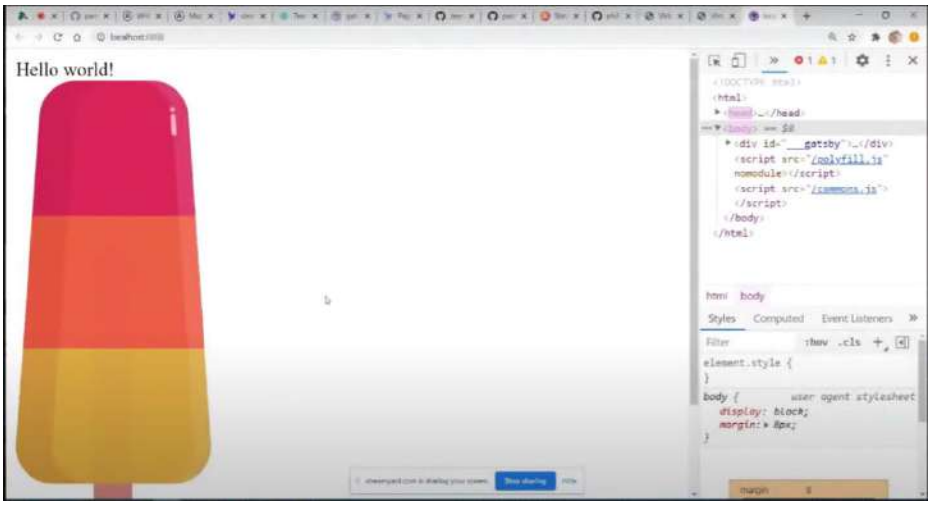
- Now we have to click, freeze the lolly, which will provide us with a link



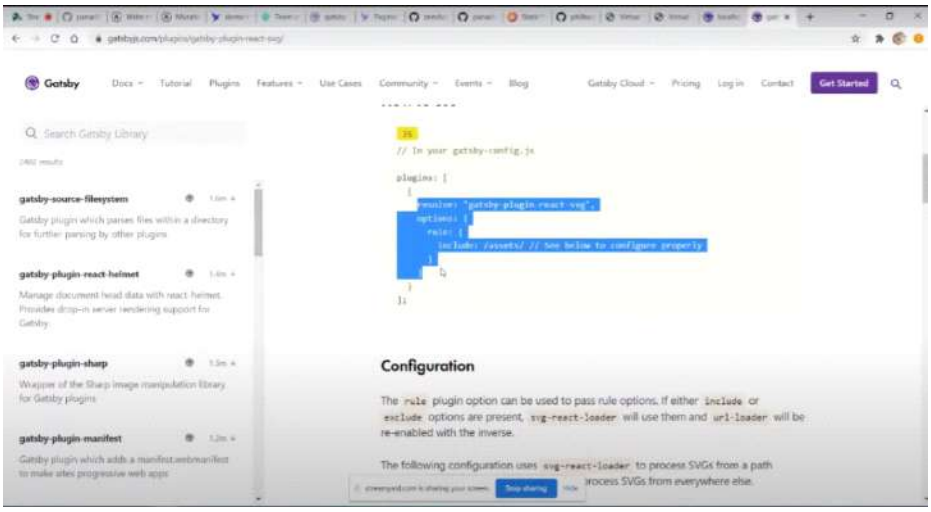
- Send this link to your and dear ones to surprise them with happiness.

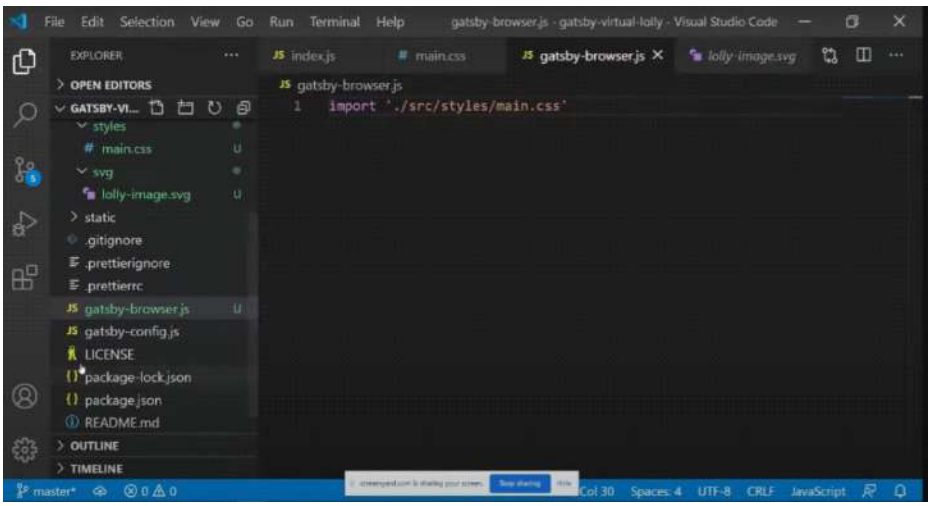
Programming a Virtual lolly

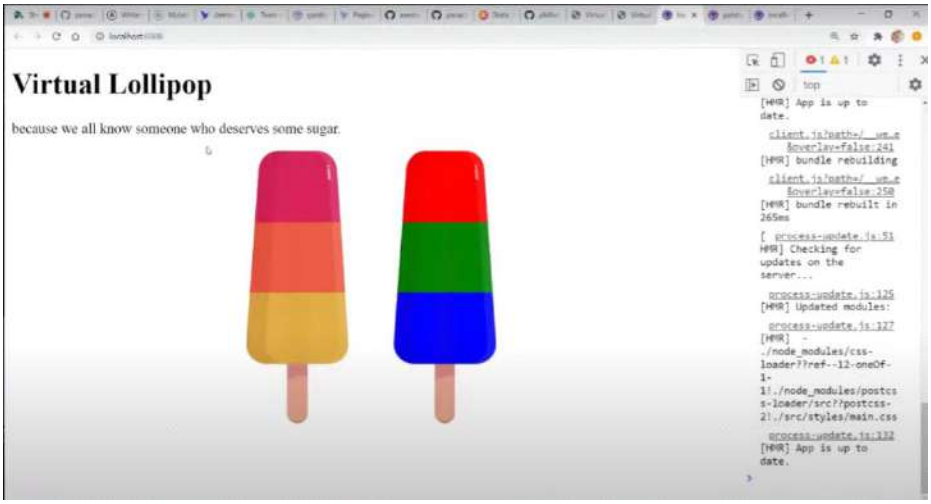
First of all, the graphical representation for the lollies must have a high resolution and have no pixilation so make a new SVG in adobe illustrator prior to the programming.

- Make a new simple hello world program and insert the SVG file into the hello world program from the directory by inserting this code Import React from “react” export default function home () {return Hello world!
- Copy the repository code from the Apollo website and paste it directly into the directory of the program.
- Run the netlify-dev program in the CMD and after wards run the program.
- 

- Install the plugin form the official website which will make the SVG a component of your program and one may face a few compilation errors.

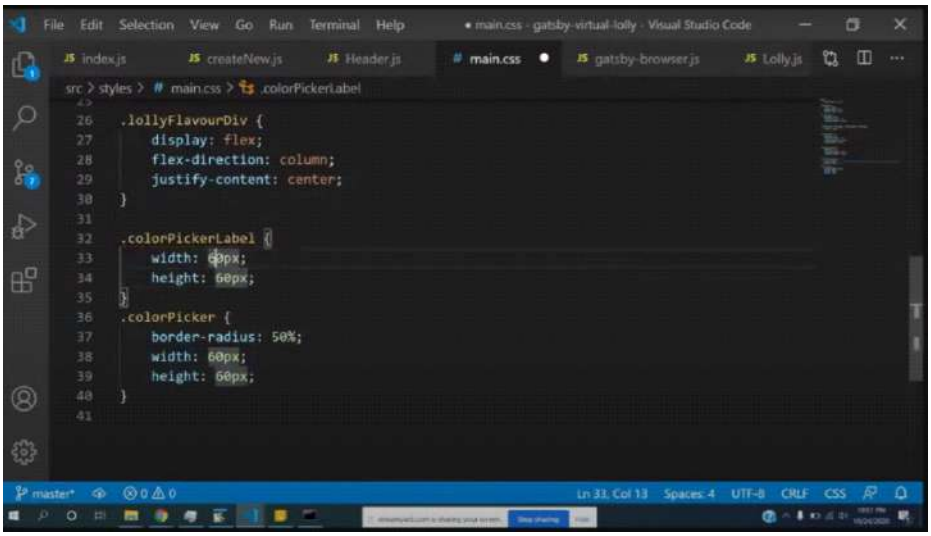
- 

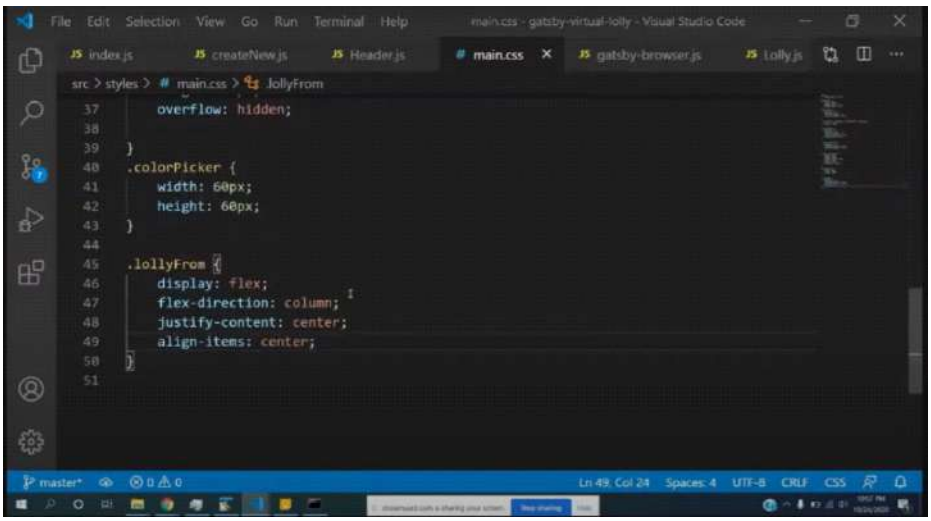
- Now import the CSS from official website into the program by using the code import ‘./src/styles/main.css’
- 

- Insert the naming command as follows <h1>virtual lollipop</h1> because we all know someone who deserves some sugar.
- 

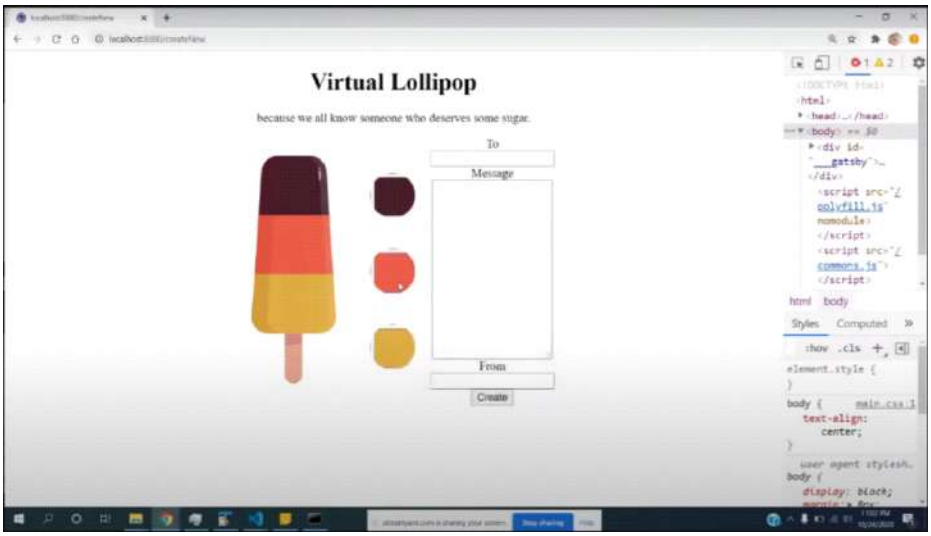
- Make a new file and insert the commands as follows

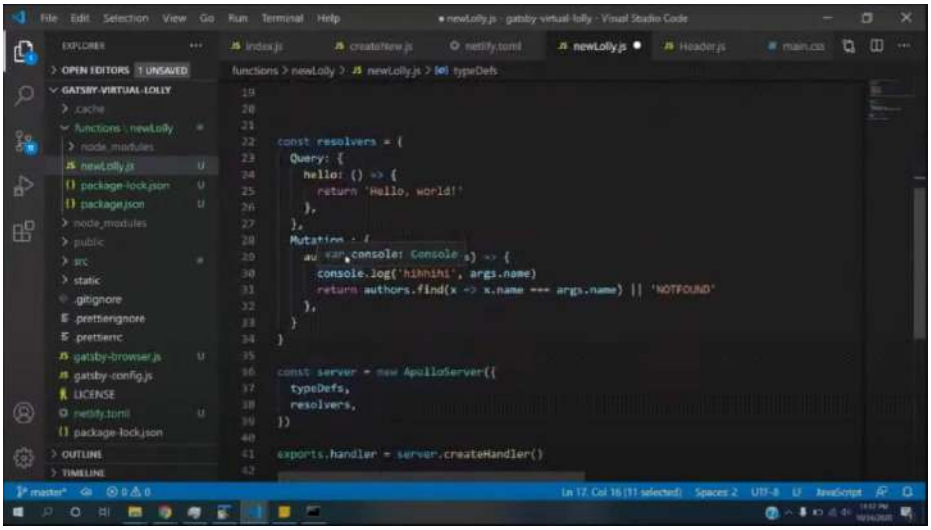
.list lollies {display: flex; flex-direrction: row; Justify-content : center; in order to skew the content perfectly.

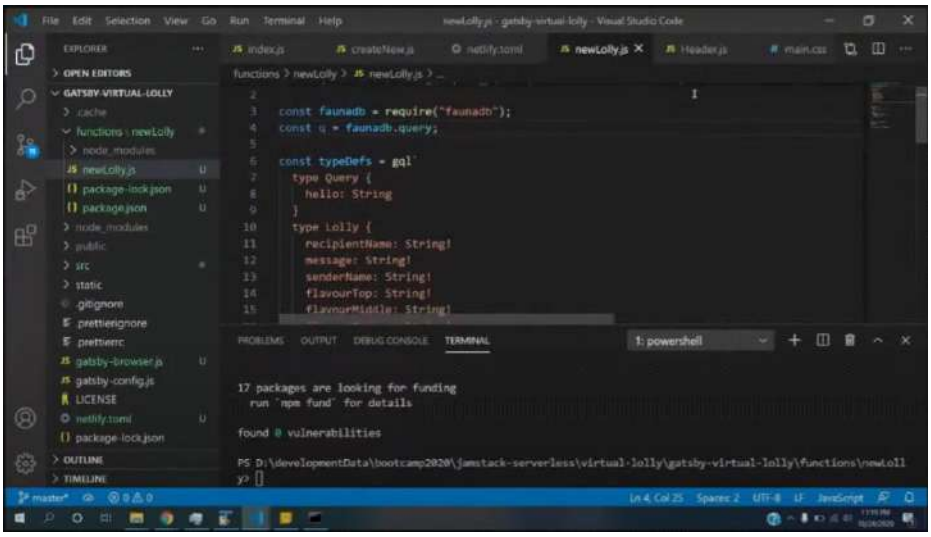
- Insert a color picker label into the program
- 

- Insert labels for the boxes and button (to be used for customization) and insert the user controls for them into the program as reflected in the image below.
- 

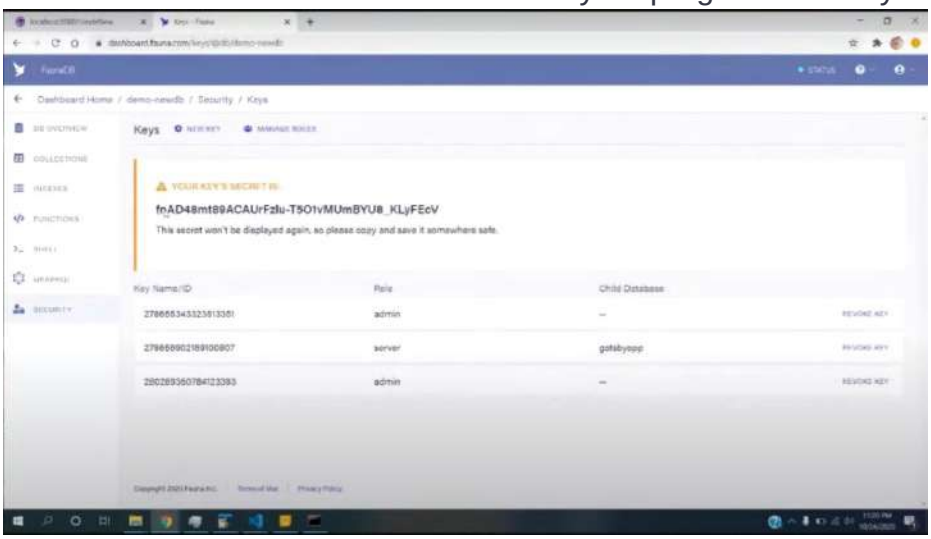
- The end result of the program if constructed properly will be as follows.

- 

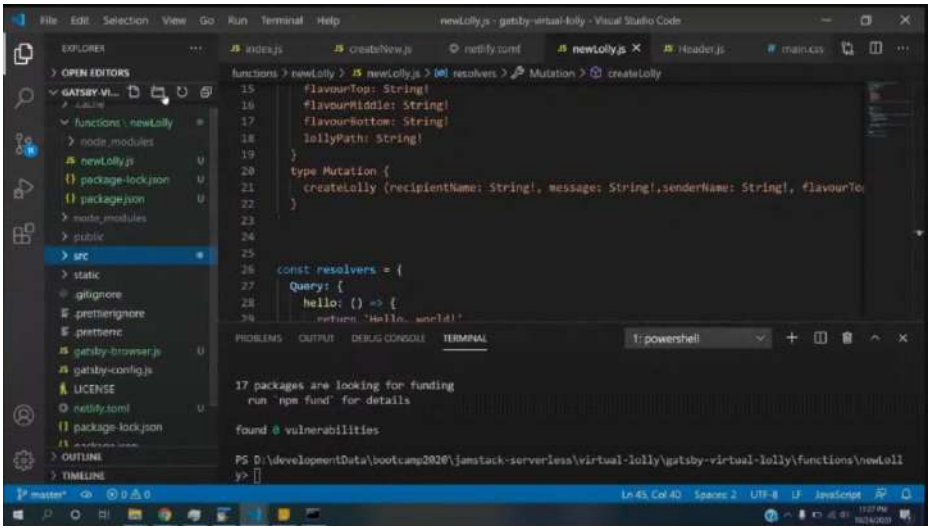
- Insert a type Mutation into the program for the generation of a successful program.
- 

- Insert const for faunaDB in order to link the lolly to the database
- 

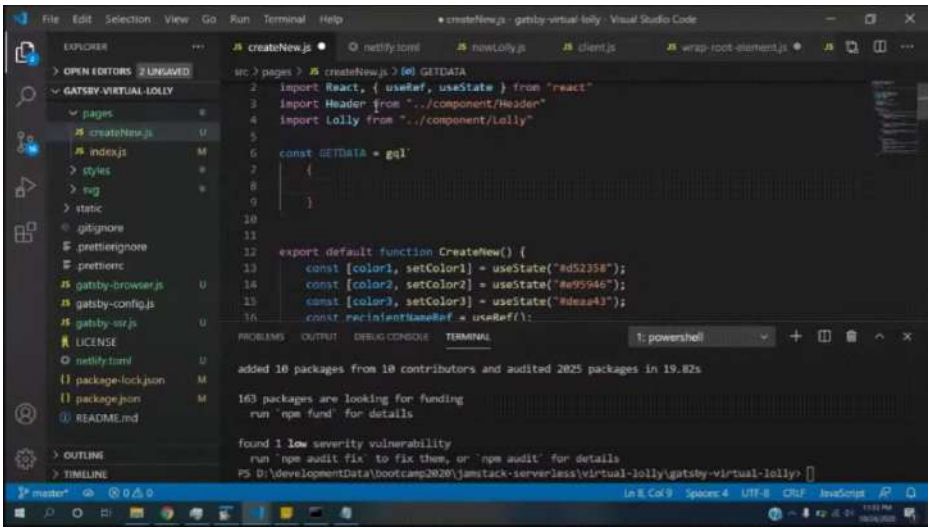
- Go to website of fauna DB and then use your programmed lolly to obtain a security key from it.



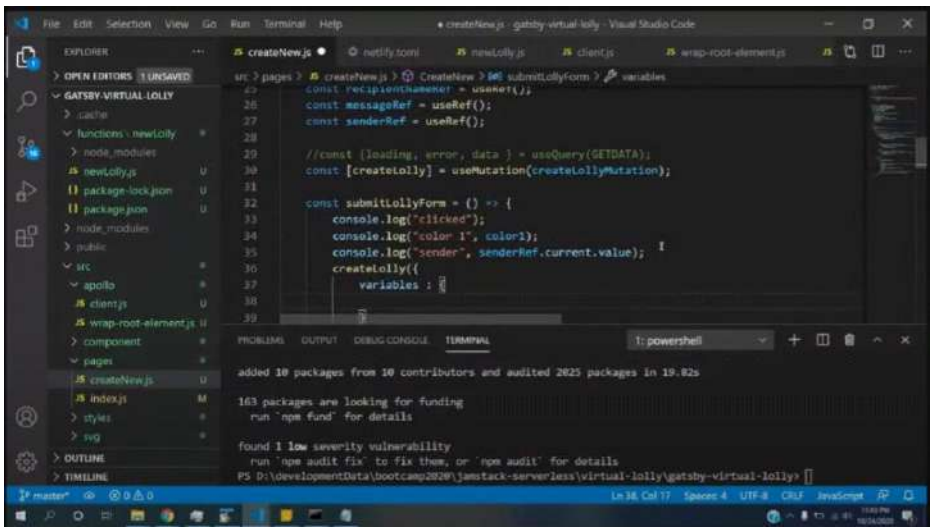
- Copy this key from FaunaDB website and paste it into the variable environment program linking it to FaunaDB Database.



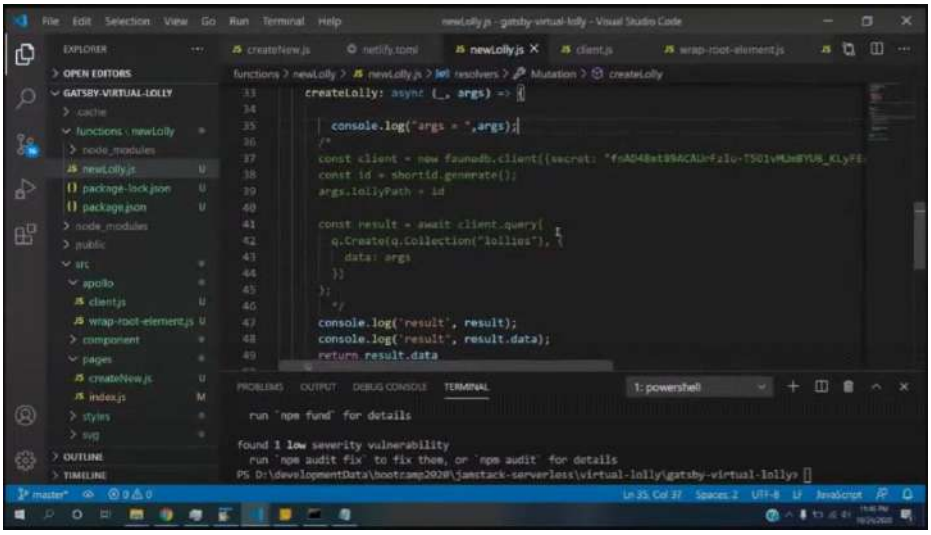
- Insert use query command from the Appolo client into the program. Use GETDATA command to fetch data from the FaunaDB website

- 

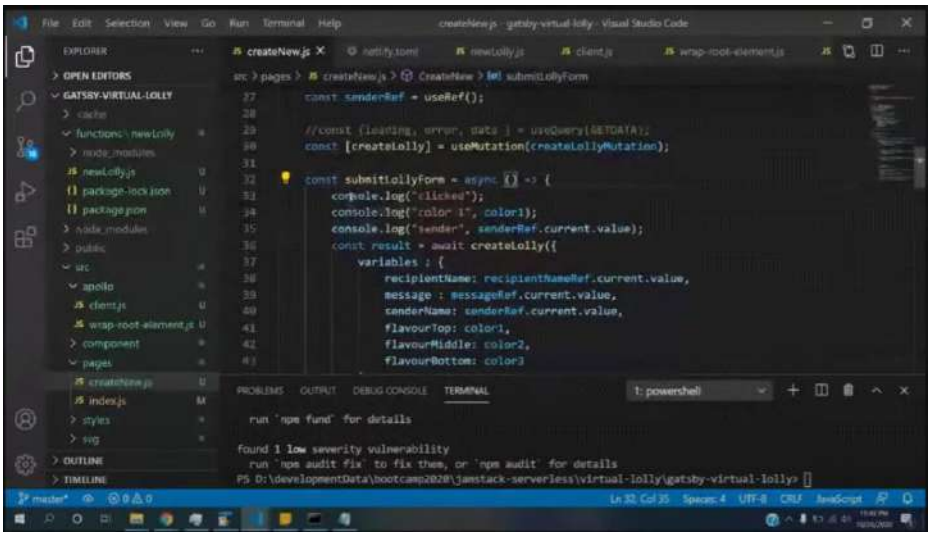
- Insert variables, constants and Mutations into the program and error check them as a control in the CMD.



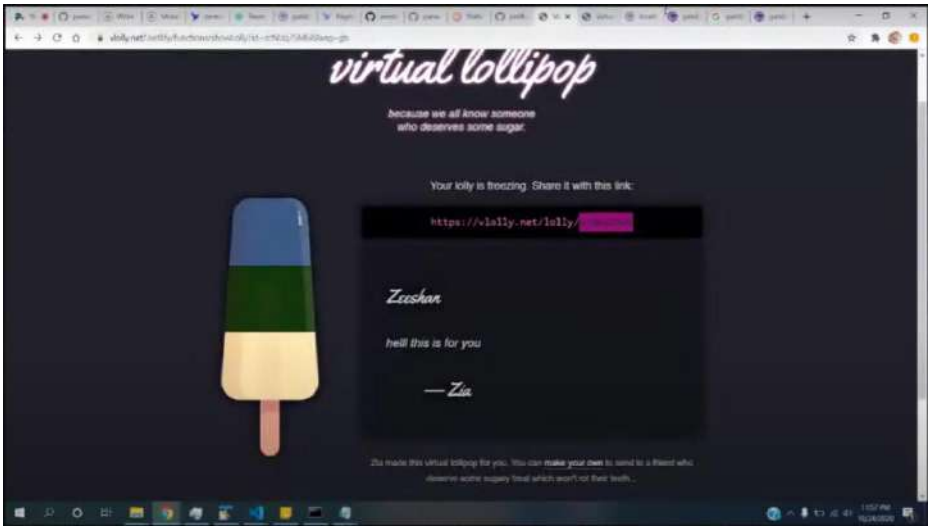
- Insert the server track and server information into the program for the purpose to run.



- Insert the async function into the program.

- 

- Adjust the preferences, themes and appearances to adjust the program accordingly



Session-42

Learn To Build An Ecommerce Website and app using JAMstack.

Learn to Build Ecommerce Websites and Apps using JAMstack

If you don't have web programming background or don't know React or Serverless we suggest you start from this app development foundation bootcamp and JAMstack Serverless Repo.

What is Ecommerce?

What is Ecommerce in 2020?

Building an eCommerce Website

A Developer's Guide to Headless E-Commerce 2020

Is JAMstack Right for Your Ecommerce Business?

9 Best eCommerce Solutions for Jamstack Websites

What do we know about Pakistan's ecommerce industry?

Is Pakistan ready to go cashless?

Digital banking grows 12pc, paper-based transactions down 10pc in FY20

No matter how obstructive the present be, ecommerce is the future

First e-commerce index shows increase in digital payments and growing demand for online shopping

What is Ecommerce?

Ecommerce, also known as electronic commerce or internet commerce, refers to the buying and selling of goods or services using the internet, and the transfer of money and data to execute these transactions. Ecommerce is often used to refer to the sale of physical products online, but it can also describe any kind of commercial transaction that is facilitated through the internet.

Ecommerce

2 minute read

What is Ecommerce?

Ecommerce, also known as electronic commerce or internet commerce, refers to the buying and selling of goods or services using the internet, and the transfer of money and data to execute these transactions. Ecommerce is often used to refer to the sale of physical products online, but it can also describe any kind of commercial transaction that is facilitated through the internet.

Whereas e-business refers to all aspects of operating an online business, ecommerce refers specifically to the transaction of goods and services.

The history of ecommerce begins with the first ever online sale: on the August 11, 1994 a man sold a CD by the band Sting to his friend through his website NetMarket, an American retail platform. This is the first example of a consumer purchasing a product from a business through the World Wide Web—or "ecommerce" as we commonly know it today.

streamyard.com is sharing your screen. Stop Sharing Hide

Whereas e-business refers to all aspects of operating an online business, ecommerce refers specifically to the transaction of goods and services.

The history of ecommerce begins with the first ever online sale: on the August 11, 1994 a man sold a CD by the band Sting to his friend through his website NetMarket, an American retail platform. This is the first example of a consumer purchasing a product from a business through the World Wide Web—or "ecommerce" as we commonly know it today.

Since then, ecommerce has evolved to make products easier to discover and purchase through online retailers and marketplaces. Independent freelancers, small businesses, and large corporations have all benefited from ecommerce, which enables them to sell their goods and services at a scale that was not possible with traditional offline retail.

Global retail ecommerce sales are projected to reach $27 trillion by 2020.

- Types of an Ecommerce model.

Types of Ecommerce Models

There are four main types of ecommerce models that can describe almost every transaction that takes place between consumers and businesses.

1. Business to Consumer (B2C):
When a business sells a good or service to an individual consumer (e.g. You buy a pair of shoes from an online retailer).

2. Business to Business (B2B):
When a business sells a good or service to another business (e.g. A business sells software-as-a-service for other businesses to use)

3. Consumer to Consumer (C2C):
When a consumer sells a good or service to another consumer (e.g. You sell your old furniture on eBay to another consumer).

- Examples of Ecommerce.

Examples of Ecommerce

Ecommerce can take on a variety of forms involving different transactional relationships between businesses and consumers, as well as different objects being exchanged as part of these transactions.

1. Retail:

The sale of a product by a business directly to a customer without any intermediary.

2. Wholesale:

The sale of products in bulk, often to a retailer that then sells them directly to consumers.

3. Dropshipping:

The sale of a product, which is manufactured and shipped to the consumer by a third party.

4. Crowdfunding:

The collection of money from consumers in advance of a product being available in order to raise the startup capital necessary to bring it to market.

5. Subscription:

The automatic recurring purchase of a product or service on a regular basis until the subscriber chooses to cancel.

6. Physical products:

Any tangible good that requires inventory to be replenished and orders to be physically shipped to customers as sales are made.

7. Digital products:

Downloadable digital goods, templates, and courses, or media that must be purchased for consumption or licensed for use.

8. Services:

A skill or set of skills provided in exchange for compensation. The service provider's time can be purchased for a fee.

The two men were actually everybody is them every category is the main one but mostly people when they think about e commerce think about b2c which is basically business for example if there is a shoe making company they make their website and sell products on it but in the recent months and years business to business has grown a lot and in Pakistan and underdeveloped countries business to business e commerce is being funded in a huge way where basically we will discuss it when we go to the local section.

The most important Ecommerce statistics in 2020.

The most important ecommerce statistics for 2020

If you're interested in doing business online, it's important to stay up to date with the latest ecommerce stats, as we all know that the best way to understand any commercial sector is through hard facts and data.

We'll start with 99Firms' Ecommerce Statistics for 2020, which prove that ecommerce is growing at a steady rate all over the globe. What's more, experts predict that retail ecommerce sales will reach $4.13 trillion in 2020.

- It is expected that by 2040, 95% of all purchases will be via ecommerce.
- The world's fastest-growing ecommerce market is China with an estimated ecommerce value of $672 billion in 2017.
- The US has the highest ecommerce penetration rates, with around 80% of all internet users making at least one purchase.
- The top reason why people make online purchases is that they can shop whenever they want, 24/7.

- The top reason why people make online purchases is that they can shop whenever they want, 24/7.
- Around 43% of ecommerce traffic comes from Google search (organic).
- Slow-loading websites see an abandonment of 75%.

Finance Online's "Data and Share Market Analysis for 2020" shows that ecommerce is not only thriving in the B2C sector, but sales are also scaling in the B2B sector and can even outgrow B2C profits by the end of 2020. Here are a few more intriguing stats from this report:

- It is estimated that around 35% of Google product searches are converted into purchases within 5 days.

Finance Online's "Data and Share Market Analysis for 2020" shows that ecommerce is not only thriving in the B2C sector, but sales are also scaling in the B2B sector and can even outgrow B2C profits by the end of 2020. Here are a few more intriguing stats from this report:

- It is estimated that around 35% of Google product searches are converted into purchases within 5 days.
- Around 51% of digital buyers conduct purchases via their smartphones.
- Digital buyers are more likely to spend more if they are provided with free shipping.
- Around 93% of online shoppers declared that the visual appearance of an online store plays a key factor in their purchasing decisions.
- It is estimated that around 80% of online shoppers don't make purchases from ecommerce sites that have problematic return policies.
- It is estimated that 85% of all products purchased via social media platforms come from Facebook.

According to Statista, e-retail sales accounted for 14% of all retail sales around the globe and these figures are expected to keep growing and reach 22% by 2023.

- It is expected that mobile ecommerce retail sales will reach $3.5 trillion by 2021.
- In 2017, around 42% of online shoppers stated that they prefer to pay with a credit card.
- Online stores that have an active presence on social media platforms have 32% more sales.
- Generation X composes 34% of the online shopping population, followed by Boomers, who account for 31% of the online shopping population. Around 30% of digital buyers are Millennials.
- Around 55% of all online shoppers said that online reviews have an impact on their buying decisions.

- Building an ecommerce website

Lots of people dream of entrepreneurship, but before the internet, actually starting a business was a difficult prospect. Now, building an eCommerce website and selling products online is within reach for anyone with the time and gumption.

Building a business of any sort is work. Coming up with your product idea, and sourcing or creating your products is a big early step you'll have to tackle. But the work of website building itself can potentially be done within a day or two.

Even if you've never built a website before, the tools now available ensure anyone can create an online store with ease. By the end of this post you'll know how to find a good product and niche for your store, how to build out your site, and the approaches you can take to generate sales.

Before we jump into the *how*, let's take a look at the *why*.

What You'll Learn... [hide]

- Why You Should Sell Products On a Website
  - 1. People spend money online.
  - 2. An eCommerce website costs less than a storefront.
  - 3. Online shopping provides convenience.
  - 4. An online presence gets your products found in search.
  - 5. The growth in mobile means people can buy from anywhere.
- Before You Build Out Your eCommerce Store
- What You Need to Build an eCommerce Website
- Building an eCommerce Website the Easy Way
  - 1. Market Research and Finding Your Niche

- Why you should sell products online.

Why You Should Sell Products On a Website

Before anyone can purchase a product from you, they need a way to discover your brand and the products you're selling. For brands that only sell products through a storefront, that limits the number of people that can learn about you to begin with, much less come to your store to make a purchase.

That was already true before the Coronavirus pandemic started, but is much more the case now that customers sheltering in place have shifted almost all their shopping to online. If you don't start an eCommerce website, there's a huge market you miss out on.

If you've been hesitant to sell products online, here are a few good reasons setting up an online store is a good idea.

- People spend money online.

- An ecommerce website costs less than a storefront.

2. An eCommerce website costs less than a storefront.

Running a storefront requires paying:

- Rent for the space
- The paychecks of staff that man the store during business hours
- Proper licensing and permits
- Utility bills
- Furniture and decorations
- Building maintenance
- Cleaning supplies
- Selling supplies like a point-of-sale (POS) system and cash registers

All of that adds up to a lot of expenses – and most of them are ongoing.

Selling products on a website, on the other hand, cuts out most of those costs. You will still need to pay for things like web hosting and marketing. And you may still need to hire some staff, depending on the size of the eCommerce store you run, but the costs are overall more manageable and easy to plan for.

- Online shopping provides convenience.

3. Online shopping provides convenience.

When you're tired, busy, sick, or just feeling a little lazy – you probably don't want to spend time driving somewhere and browsing a store for the items you want to buy. People have a lot going on and sometimes there just isn't room in their lives for a trip to the store.

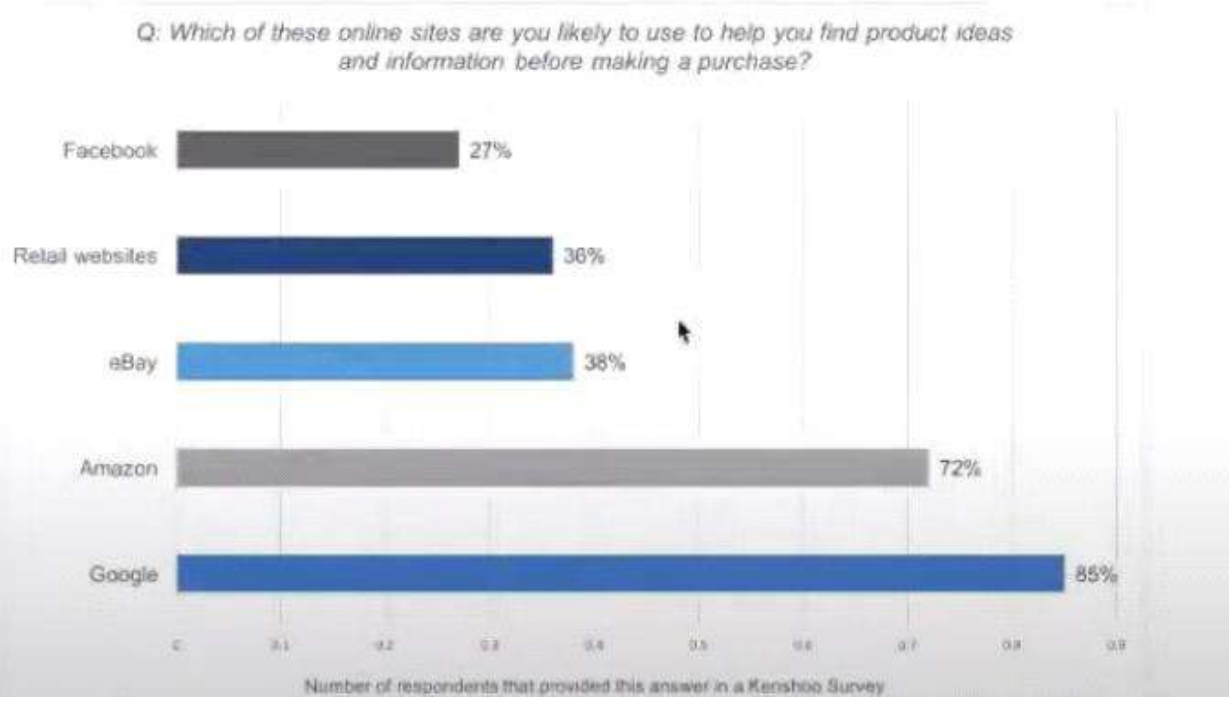
By comparison, browsing an eCommerce site online and making selections is easier, faster, and requires a lot less energy. People are more likely to spend their money when it doesn't feel like work to do so.

Online shopping isn't 100% effortless – but it's pretty close. When you sell your products through an online store, you remove a lot of the barriers to buying that exist with a physical store.

- An online presence gets your found on search.

4. An online presence gets your products found in search.

85% of people turn to Google for product discovery and shopping.



Without an eCommerce website, your products have no chance of showing up when potential customers start looking for what you sell online.

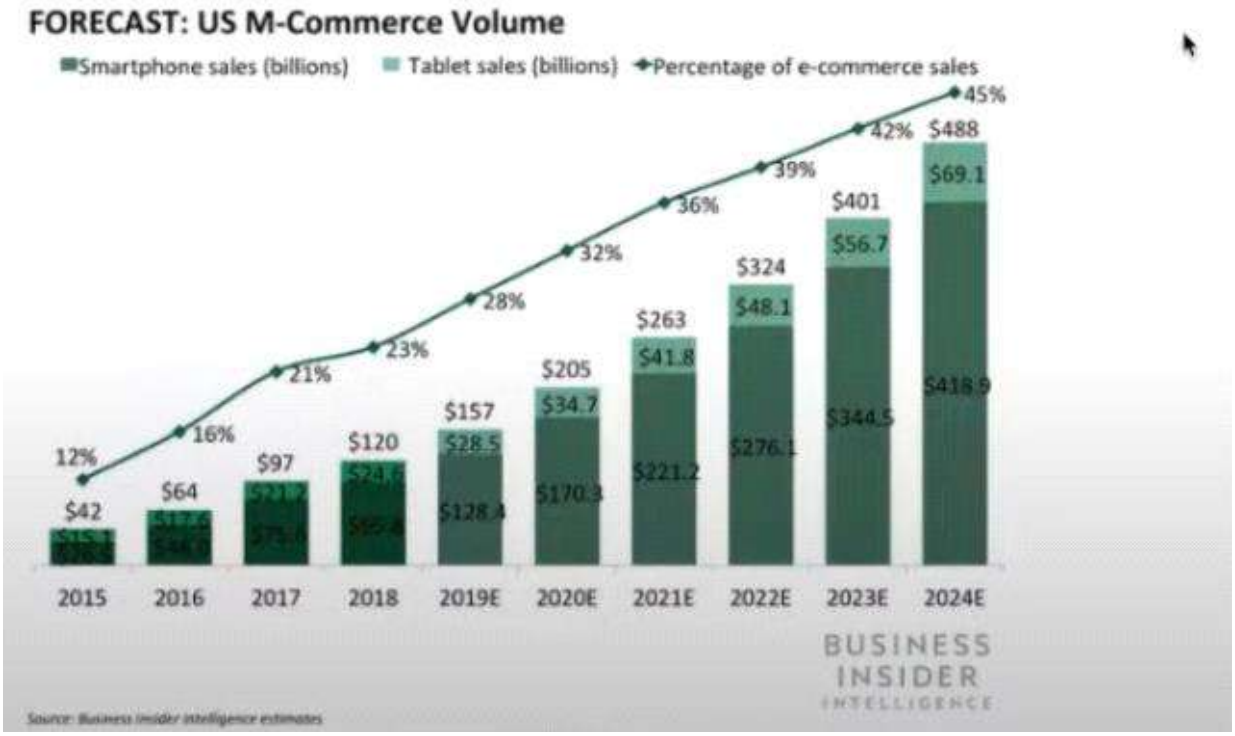
But when every product you sell is listed on a page on your eCommerce site, all of them will be indexed by Google. That's the first step to your products showing up in search results.

If you want those product pages to show up on the first page of Google, that requires an investment in SEO best practices. But before you can even start thinking about that, you need an eCommerce website for your products.

- The growth in mobile means you can buy product from anywhere.

5. The growth in mobile means people can buy from anywhere.

Mobile purchasing trends show that over a quarter of all eCommerce purchases now occur on mobile devices.



Mobile shopping makes it possible for people to make purchases at the exact moment they decide they want something. Whether they're on the couch watching TV, strolling through the park, or lounging on the beach – if they think of something they want, they can make the purchase right then.

That creates more opportunities for sales. You don't face the risk of a potential customer deciding they want an item you sell, only to forget about it by the time they get home to their computer. That's good for your customers, and it's good for your bottom line.

Now it's time to jump into the *how*.

- Before you built your ecommerce course.

Before You Build Out Your eCommerce Store

Before you start the process of building your eCommerce store there are some foundational things you need first. To start, you need a domain name and web hosting plan. Without these, there's no way to actually have a live eCommerce website.

There are dozens of different providers and services you can use to secure a domain name and hosting, and you can either purchase them together or separately. If this is your first time building a website, the easiest course of action is purchasing them together from the same provider. This minimizes the number of technical tasks you need to complete and speeds up the process of getting your site online.

If you're looking for the fastest and most beginner-friendly way to secure a domain name and hosting, then consider purchasing them both right here at HostGator. Just select your web hosting package and add a domain name to your order before you check out.

<u>What You Need To Built An eCommerce Website</u>

- A domain name
- A web hosting provider
- An integrated payment processor
- A platform or tool to build your store
- Initial capital to create your products
- Packaging for product shipping

What You Need to Build an eCommerce Website

Building an eCommerce website will be a little different than creating a standard website or starting a blog. Beyond securing a domain name and hosting you'll also need to consider tools and software for processing payments, along with actually creating and manufacturing your products.

Here's a quick rundown of what's required to build a website from scratch for an eCommerce store:

- A domain name
- A web hosting provider
- An integrated payment processor
- A platform or tool to build your store
- Initial capital to create your products
- Packaging for product shipping

While building an eCommerce store is much cheaper and easier than building a storefront, it does still require some upfront capital and work. Make sure you work out a budget that makes sense for your goals.

- Building the ecommerce website the easy way.

Building an eCommerce Website the Easy Way

Building an eCommerce website is similar to the process of building other websites, except that it needs to be set up to sell physical products. You should anticipate a bit more work upfront to get your website ready.

Here's the process of discovering your market, finding out what you're going to sell, and finally building out your store.

- Market research and figure out your Niche.

1. Market Research and Finding Your Niche

Before you start your eCommerce business, you need a physical product to sell. But you definitely don't want to put the work into building an online store only to find out that no one wants to buy what you're selling.

To avoid this fate, spend time upfront finding the right market and products to sell. You might already know exactly what you want to sell. But if not, no worries, this post will walk you through the process of conducting market research for a small business.

Ideally, you want to uncover a pressing need in the marketplace and fill it with your product.

- Look for trending products

Look for Trending Products

One way to ensure that you'll start selling products right out of the gate is to align yourself with products that are *currently* trending. By getting into a trending niche at the right time, you can ride the wave in the short-term and long-term.

However, you don't want to hop into a trending market when it's on the decline. Instead, you want to find a product or a niche for products that are growing in popularity.

There are tons of different tools you can use to spot trending products and niches, but here are some of the most common.

- Google Trends: This Google tool allows you to look into the past and future of certain product keywords. By using the data this tool provides, you'll be able to see if a market is growing or waning in popularity.
- TrendHunter: This site showcases a lot of different niches and products that are currently trending across the globe. Plus, there's a nifty filter that lets you sort by different industries.
- Instagram influencers: If you spend time on Instagram, then you probably follow a few influencers who regularly promote products. The chances are pretty high that the products they promote tend to sell well and can suggest a viable niche.

- Align with your passion

Align With Your Passion

Hopefully, the resources above will point you toward a handful of products or potential niches to consider. To narrow down that list even further, do some self-evaluation. You'll find it much easier to sustain momentum with your store if you actually care about what you're selling.

Consider the following questions to determine if you have what it takes to sustain yourself in the niche:

- Do you care about the market you've chosen?
- Do you have any pre-existing knowledge you can rely on?
- Can you see yourself working in this space for 5 years or more?
- Do you care about the people that you're serving?

- Explore what's currently selling

Explore What's Currently Selling

Once you've narrowed down your list of potential products and niches to a group you actually care about, go one step further and validate your ideas in the marketplace. The most effective ways to do this are using the Amazon product marketplace and a keyword research tool.

First, head over to the Amazon bestseller list in the product category of your choice. Then, spend some time digging through the products to see what kinds of things people are actually buying. Look for product multiples and dig through the reviews to see if there are any glaring issues with the existing products.

- Getting ready to sell.

2. Getting Ready to Sell

Before you settle on the product you want to sell there are a few more considerations to take into account.

First, make sure the product is light enough that it's easy to ship. During the early days of your eCommerce store, you'll be handling most of the packaging and shipping yourself. Sticking with products that don't weigh much will not only make your life easier, but you'll be able to maintain better product margins, which can be crucial in the early days of your store.

Finally, consider creating a prototype for your product. That means starting with a sample of the product you have in mind that you can test out in the market. If you're building a new product, you probably won't make it perfect on the first time. Prototyping gives you room to iterate and improve based on feedback—before you make a huge investment in inventory. This will help you get to market faster and build a better product over time.

- Choosing your ecommerce platform.

3. Choosing Your eCommerce Platform

When the time comes to choose the platform for your eCommerce store you're going to have a lot of different options at your disposal. For first-time eCommerce entrepreneurs, some of the most popular options include:

- WordPress/WooCommerce
- Magento
- Website Builders

WordPress OR WooCommerce

WordPress/WooCommerce

WordPress is the most popular content management system (CMS) in the world, and for good reason. It's relatively easy to use (although still requires a learning curve for beginners), but is both powerful and versatile because of the massive library of plugins available. To build an eCommerce website on WordPress, you'll need to choose an eCommerce plugin to provide the proper functionality.

The most popular option is WooCommerce, which is free and powered by the team that owns WordPress. That means you'll never have to worry about compatibility issues, and can count on access to plenty of compatible themes and add-ons. WooCommerce will walk you through the process of getting your online store set up, so it's mostly a matter of following the steps in the plugin's setup wizard.

- Magneto and website developer.

Magento

Magento is another popular eCommerce CMS, but one that requires more skill and knowledge. The tradeoff is that it can be more powerful and give you more options for customizing and building out functionality within your eCommerce store. But unless you're a pretty skilled developer or willing to hire someone that is, it will take a lot more work to learn and use.

Website builders

For many new eCommerce owners, the easiest option by far is using a website builder that includes eCommerce options. Most website builders are designed with beginners top of mind, so anyone should be able to put a website together quickly and easily. With a website builder, you start with a pre-designed template, then can easily add in original images, move things around, on the page with drag-and-drop functionality, and fill in your copy.

- Choose element to include on your ecommerce website

- Organise your E-Commerce website based on how people shop.

Organize Your eCommerce Website Based On How People Shop.

One of the first steps to organizing an eCommerce website is figuring out what pages and categories to include. You'll obviously want a home page and pages for each of your different products. But how will you organize all those product pages so that people can find what they're looking for?

Think about how your potential customers are likely to shop and browse and build out the site architecture that makes the most sense based on that.

For example, if you sell pet food and supplies, you might organize your website based on the type of pet your visitors have (dog, cat, fish, mouse, etc.) with subcategories under each for product types (toys, treats, food, etc.). Anyone visiting the site can pretty quickly figure out how to narrow down their search based on their particular needs. And by visiting a page that groups a lot of similar products together, they can browse all the dog treats or cat toys available and pick the one they most want.

.

- Include a search bar

- Include Reviews

Include Reviews.

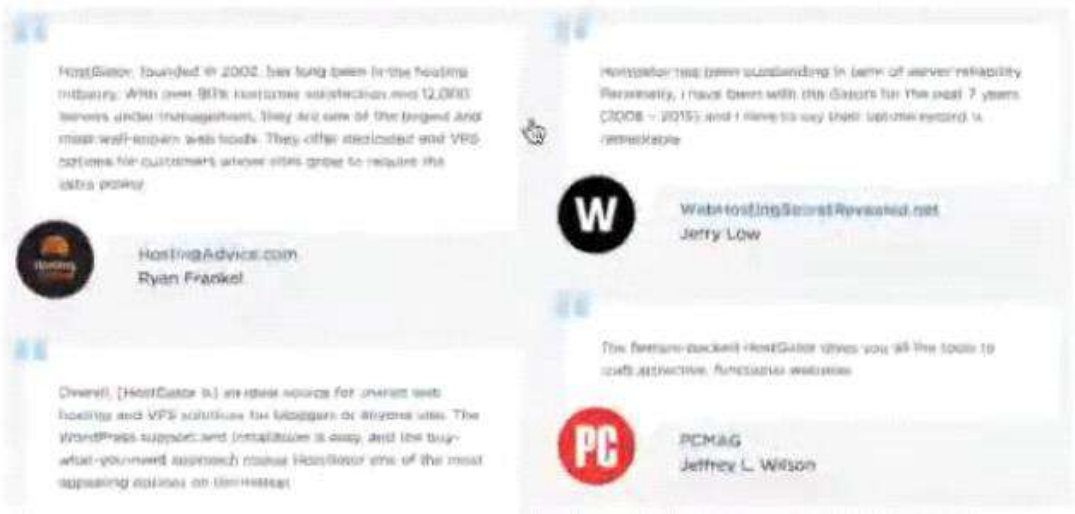
Over half of regular online shoppers say they read reviews almost every time they shop online. Looking at customer reviews has become a regular part of the purchasing process.By including a review feature on your website, you give your customers a way to hear directly from each other – which is more powerful than what you can tell them.

Enabling reviews from third parties shows your customers you're confident that your products can stand up to customers' honest feedback. And if the reviews are positive (which they should be if your products are solid), they'll increase your sales.

As an added benefit, reviews can help you gain valuable feedback about the products you offer and the service experience your customers have. You may be able to pick up some tips that help you improve your eCommerce business results over time.

For example, at HostGator, we feature reviews on a dedicated page on our site:

For example, at HostGator, we feature reviews on a dedicated page on our site:



- Having as shopping cart

Have a Shopping Cart.

Your ideal customer isn't going to see one item they like and go straight to the purchase process. They'll spend some time browsing and choosing several items they want.

A shopping cart is another standard feature of eCommerce stores because it allows visitors to save the items they want to purchase while they continue browsing, and then go through the purchasing process once for all of them.

And for visitors that add items to a shopping cart but don't buy right away, you can follow up with a reminder email to your customers to help nudge them back to the site toward a purchase.

- Include wishlist functionality

Include Wish List Functionality.

Including wish list functionality accomplishes a few different things:

- It gives people an easy place to point friends and family in the lead up to a gift-giving event – a situation people regularly make purchases for.
- For people who prefer to do research before they buy a product, it gives them a chance to save the items they like for possible future purchase.
- For anyone avoiding impulse buys for budgetary reasons, they can mark now the things they want to come back for later.

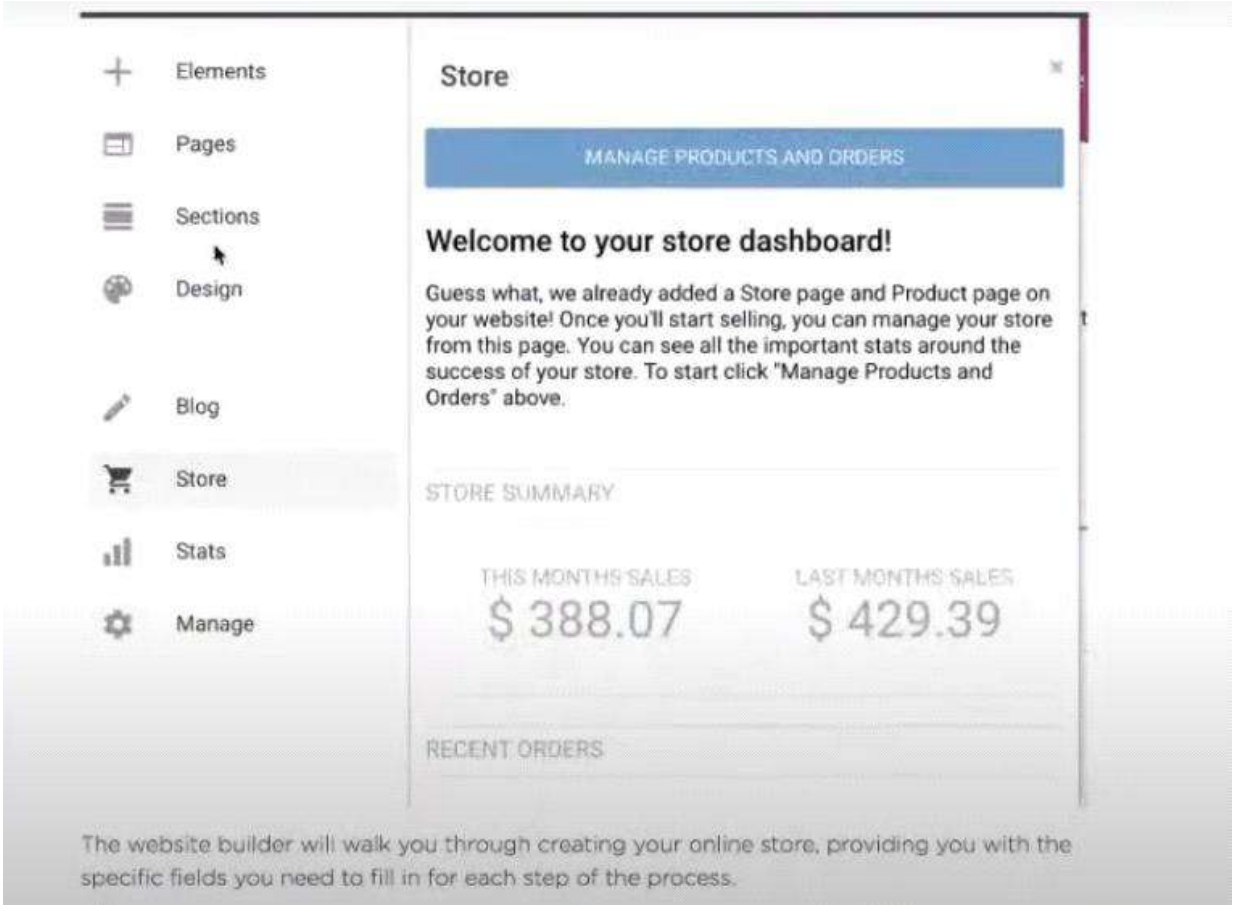
Giving people a way to use your website to create a list of items they want creates a few opportunities where those items are more likely to become future sales.

- Building your online store.

5. Building Your Online Store

The step-by-step details of building your online store will vary based on the platform you choose. But in each case, creating product pages that effectively highlight the items you're selling is the most important part.

If you're building your website using the Gator Website Builder, start by making sure you have the eCommerce plan. Once you do, Store will be an option in the menu on the left-side of the screen. When you select it, you'll see a button labeled Manage Products and Orders.



- Launch your store and selling products.

6. Launching Your Store and Selling Products

If you've done everything above, then congratulations, you're just about ready to launch your store. However, it's important not to launch your store before it's fully polished. After all, you only have one chance to make a first impression.

Spend some time double-checking every page on your site for typos, broken links, or other things that don't work. You'll also want to run through a few sample purchases to make sure the checkout process works smoothly.

Now that you've learned to create a website to sell products, it's time to launch your store and start selling!

- Marketing your ecommerce website.

Marketing Your eCommerce Website

The internet is vast, and you have a lot of competition. If you don't make an effort to get your website in front of people, they're unlikely to find you at all, much less buy from you. To be successful, you need to invest in marketing your eCommerce store to the world. Here are a few of the most common eCommerce marketing methods:

- Content marketing

1. Content Marketing

Content marketing is a valuable marketing tactic for building up your website's search engine optimization (SEO) authority, attracting more relevant traffic to your site, and building trust with your target audience. Content marketing requires a considerable investment in time and money, but it's a good way to grow the online influence of your business.

Content marketing can take a number of different forms:

- **Blog posts –** A blog is usually the go-to starting place for content marketing. Regularly publishing high-value blog posts about topics important to your audience is a great way to raise your profile and demonstrate your trustworthiness.
- **Longform content –** Longform content like in-depth guides or ebooks can be published to your website as a way to build up SEO, or placed behind a form as a lead generation tactic. If you create something valuable enough that your audience is willing to give you their address in return for it, that's a good way to build up your email list and identify potential customers.

Get the ebook!

Where should we send it? Give us your email and we'll send you the free guide to launch your home business!

Email

By submitting this form, you are consenting to receive marketing emails from: HostGator, 1500 North Priest Drive, 2nd Floor, Tempe, AZ, 85281, US, http://www.hostgator.com. You can revoke your consent to receive emails at any time by using the SafeUnsubscribe® link, found at the bottom of every email. Emails are serviced by Constant Contact.

- Social media marketing.

2. Social Media Marketing

Social media can be a great way to increase brand awareness while driving traffic back to your eCommerce site. For eCommerce businesses, some of the best channels to grow your audience and promote your products are Instagram, Twitter, Facebook, and Pinterest.

Instagram and Pinterest are image-driven platforms, so they align well with selling products.

Instagram will give you a way to showcase your product and demonstrate how it's commonly used. And Pinterest gives you the opportunity to create and share content that's related to your niche and drive targeted traffic back to your site.

Twitter and Facebook are better for growing your community and promoting your content. You can use them to tap into the community that's talking about your industry, connect with influencers in the space, and directly reach your audience. You can also get a lot out of community-building tools like Facebook groups and Twitter chats.

In addition, you can monitor social channels to stay on top of what people are saying about your products. Negative tweets are useful for learning how to improve your products. And positive ones are powerful social proof.

Get the guide: *HostGator Web Pros break down everything you need to know about Facebook and Instagram Stories for your online business. Read now!*

- Influence marketing.

3. Influencer Marketing

Influencer marketing has been used to build some wildly successful eCommerce stores. It's the process of reaching out to influencers (most often on social media channels like Instagram) and having them promote your brand and products to their audience.

You have the option of doing paid shoutouts, but you might also be able to find accounts who will mention you for free in exchange for your product. Influencer marketing plays on the idea that these people have already built a loyal following, so your product gains exposure and trustworthiness by association.

If you're interested in exploring this further, then check out the resources below:

- 5 Excellent Influencer Marketing Ideas for Any eCommerce Store
- The Ultimate Guide to Growing Your Business with Influencer Marketing

SEO

SEO stands for "search engine optimization." In simple terms, it means the process of improving your site to increase its visibility for relevant searches. The better visibility your pages have in search results, the more likely you are to garner attention and attract prospective and existing customers to your business.

4. SEO

Search engine optimization is the collection of tactics that increase your website's authority according to the factors valued by search engine algorithms. If that definition sounds a bit technical, it just means that SEO is how you increase your chances of showing up on the first page of Google (and Bing and Yahoo, but mostly Google).

SEO consists of doing a few main things:

- **Keyword research** – All of SEO is based on targeting the specific keywords you want to rank for, so this is an important step to start with. Use SEO tools to learn what terms your audience is commonly searching for.
- **On-site optimization** – This step includes making sure your website works on all the important technical levels, like loading quickly. It also means ensuring that each page is optimized for your chosen keyword.
- **Content marketing** – Content marketing involves creating a lot of great content that keeps your website fresh, useful, and gives you the chance to target more keywords. For all those reasons, it helps with SEO.
- **Link building** – The search engine algorithms see links as a measure of how valuable other people see your website. As such, they're an important ranking factor. Link building involves encouraging other websites to link back to yours.

- Email Marketing

5. Email marketing

Email marketing is arguably the best way to turn website visits into relationships. Every consumer that opts into your email list is telling you they want to hear from you. It's an invitation to keep the relationship going, which is a big deal!

Email marketing gives you a good way to keep driving traffic back to your site by promoting new content and sharing company announcements. And it gives you a loyal audience to promote special deals to. For eCommerce stores, earning new customers is definitely important. But getting those customers to come back again and again is how you really build a sustainable business.

For more information on how to use email marketing to grow your business, check out:

- 5 Tips for Launching Your First Email Marketing Campaign
- 7 Powerful Strategies to Fine-Tune Your Email Marketing

- PPC advertising

6. PPC advertising

All of the marketing tactics we've described so far are useful, but require a significant amount of time and work before you start seeing results. They're all about playing the long game. If you want to start driving traffic to your website sooner rather than later, adding paid advertising methods to your list will help with that.

Pay-per-click (PPC) advertising describes any marketing you do on platforms that charge for ads based on how often people click on them, rather than how many times they're shown. Most of the ads you see on Google employ this model, as do most social media platforms.

- A Developer guide to headless CMS

Let's talk about modern e-commerce.

Many call it *Jamstack* or *headless* commerce. They're umbrella terms—helpful yet confusing.

I try my best not to indulge in cynicism, so I won't label them buzzwords.

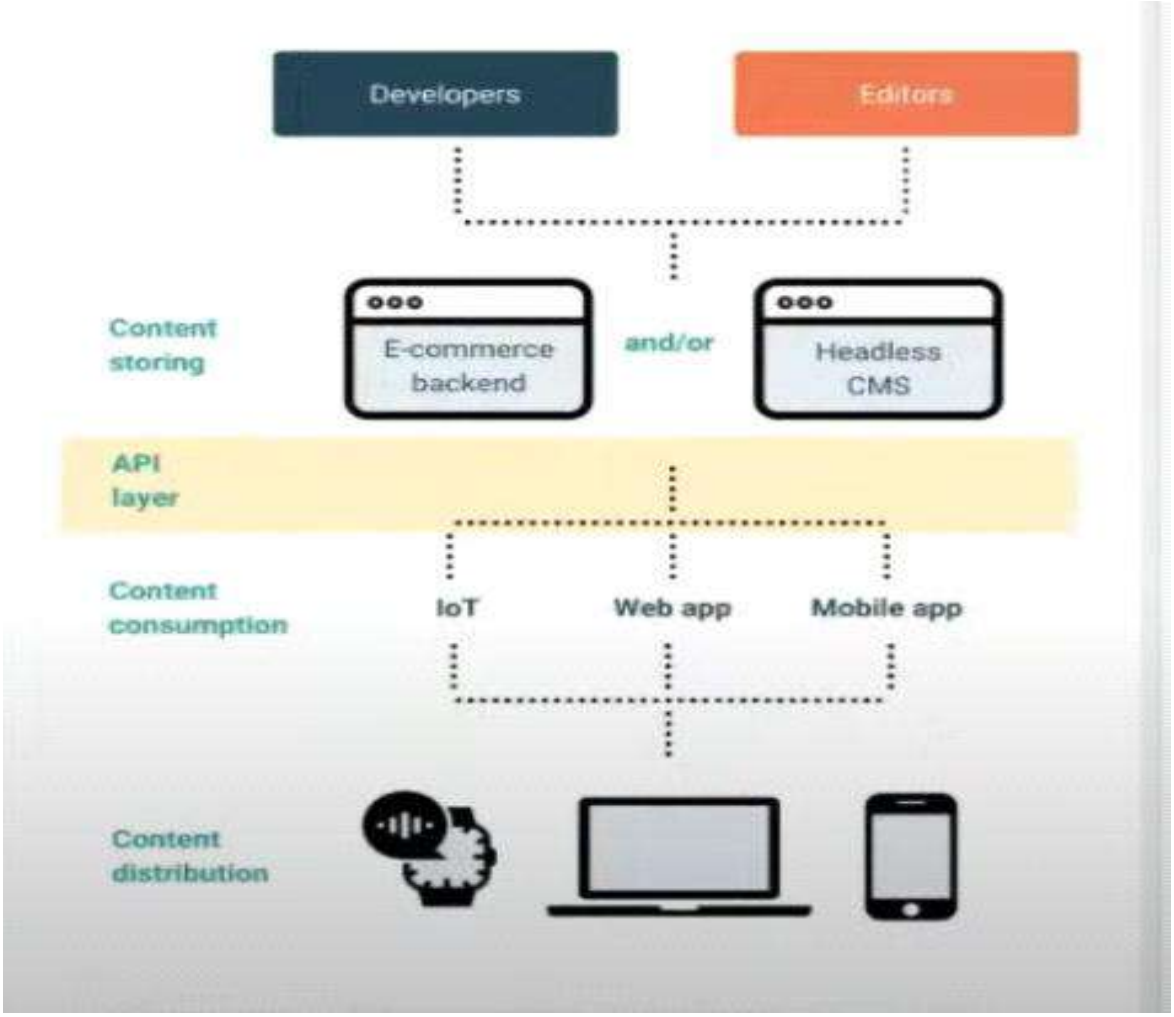
This post will unpack these concepts. It will also contrast traditional approaches with modern ones.

My goal is to help you better navigate e-commerce as a developer in 2020.

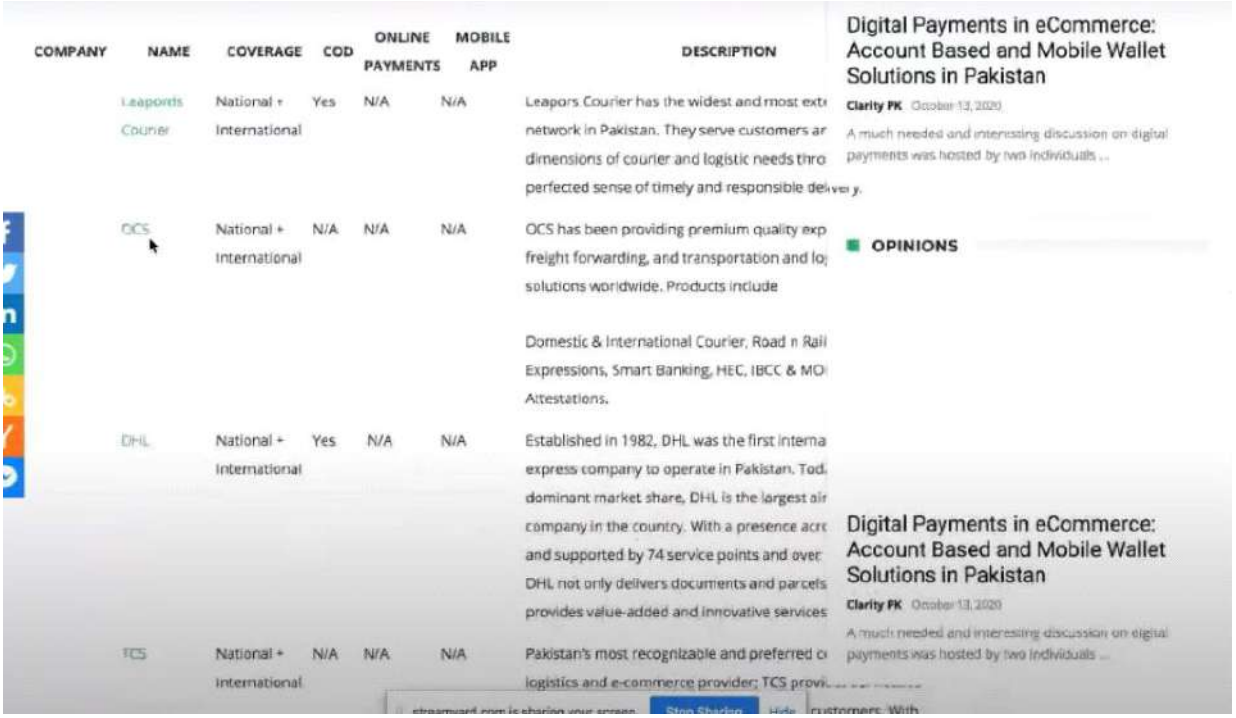
By the end of this post, you'll be able to make better decisions when it comes to picking an e-commerce stack.

Full disclaimer: I'm the CEO at Snipcart, an e-commerce solution for developers. I'll talk about all platforms, including ours. If your next online store ends up being powered by Snipcart, great. If not, no biggie. We're independent and profitable, so we have the luxury of recommending other solutions. I do it all the time. ✌

- What is headless ecommerce?



- List of delivery service



COMPANY	NAME	COVERAGE	COD	ONLINE PAYMENTS	MOBILE APP	DESCRIPTION
	Leaports Courier	National + International	Yes	N/A	N/A	Leapors Courier has the widest and most ext network in Pakistan. They serve customers ar dimensions of courier and logistic needs thro perfected sense of timely and responsible deliv y.
	OCS	National + International	N/A	N/A	N/A	OCS has been providing premium quality exp freight forwarding, and transportation and lo solutions worldwide. Products include Domestic & International Courier, Road n Rail Expressions, Smart Banking, HEC, IBCC & MO Attestations.
	DHL	National + International	Yes	N/A	N/A	Established in 1982, DHL was the first interna express company to operate in Pakistan. Tod dominant market share, DHL is the largest air company in the country. With a presence acr and supported by 74 service points and over DHL not only delivers documents and parcels provides value-added and innovative services
	TCS	National + International	N/A	N/A	N/A	Pakistan's most recognizable and preferred c logistics and e-commerce provider; TCS provi customers. With

Digital Payments in eCommerce: Account Based and Mobile Wallet Solutions in Pakistan

Clarity PK October 13, 2020

A much needed and interesting discussion on digital payments was hosted by two individuals ...

OPINIONS

Digital Payments in eCommerce: Account Based and Mobile Wallet Solutions in Pakistan

Clarity PK October 13, 2020

A much needed and interesting discussion on digital payments was hosted by two individuals ...

- E-COMMERCE investment in Pakistan and developing countries

Ecommerce Investments in Pakistan and Developing Countries

Retailo raises USD 2.3 million pre-seed for its B2B ecommerce marketplace in Saudi & Pakistan

Miniso has acquired ToSharing for around $7 million to expand eCommerce operations in Pakistan

Pakistani e-commerce platform PriceOye announces six-figure funding in recent round

Pakistani baby products ecommerce platform Baby Planet raises $250,000 led by High Output Ventures

Pakistan's Tajir raises $1.8 million for its B2B ecommerce marketplace that helps mom-and-pop stores source inventory

Two Pakistanis on mission to revolutionize e-commerce industry

Want to become an ecommerce millionaire? These eight business models are up for grabs

US-based venture capital fund backs Pakistani e-commerce platform PriceOye.pk

Dastgyr Pakistani Startup Raises Six-Figure Investment For Its B2B E-Commerce Marketplace

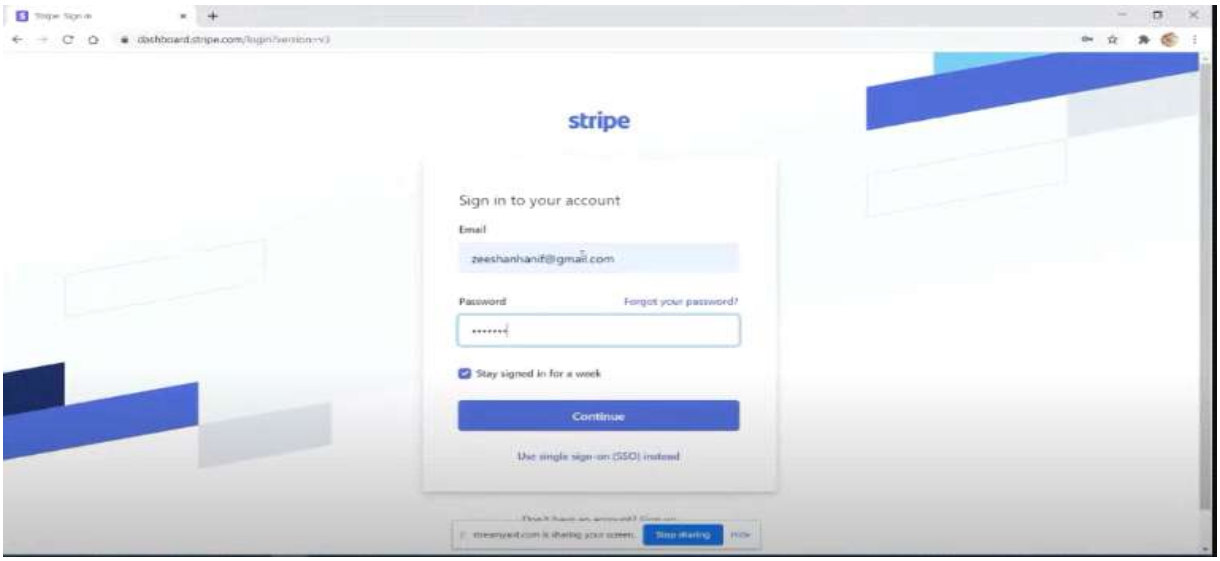
Sequoia India, Flourish Ventures lead $22.5m round in Bangladeshi B2B commerce startup

Session- 43

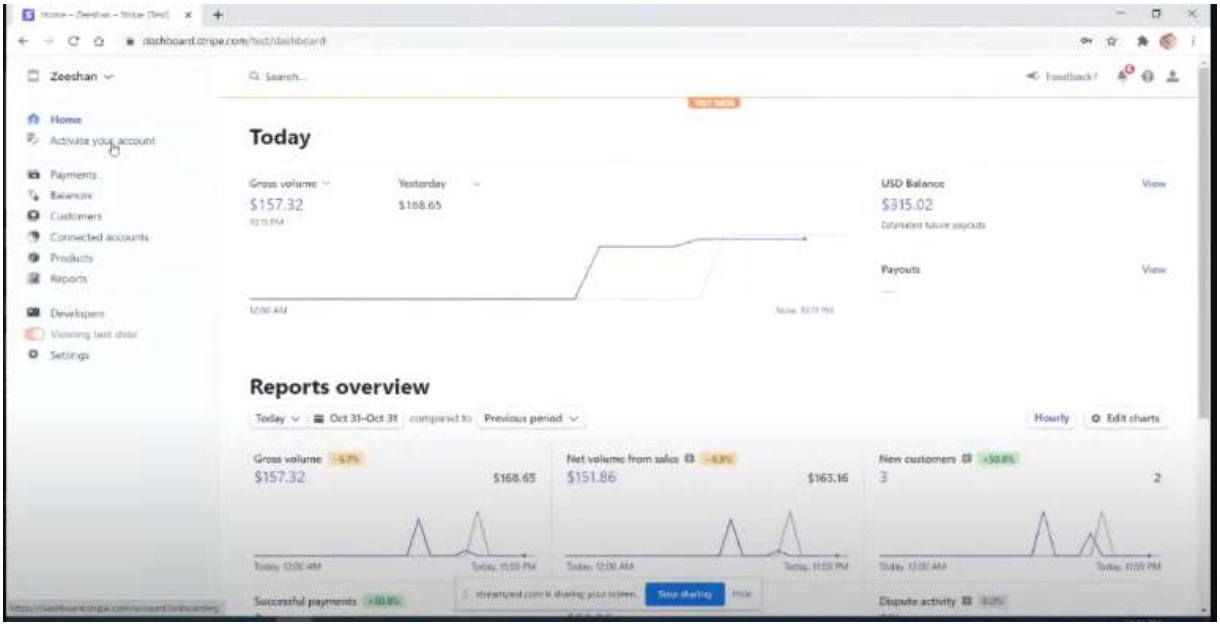
Payments Using Stripe Checkout

Project “0” (Gatsby Stripe Checkout)

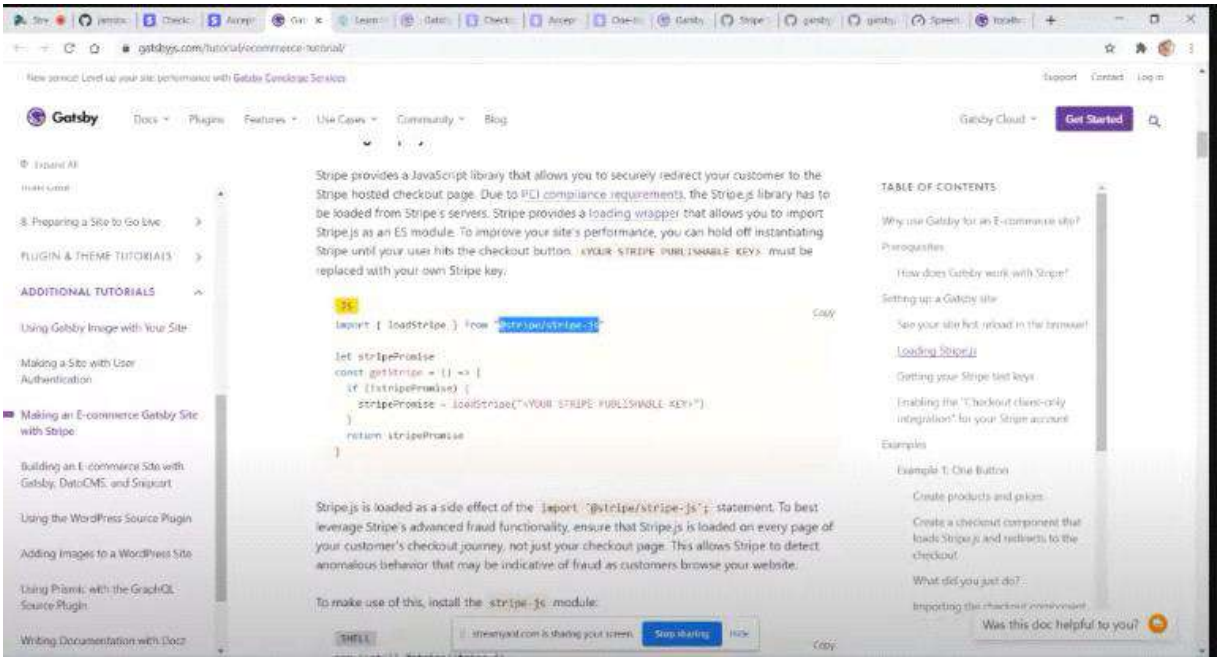
- First of all start the project “0”
- Go to Stripe.com
- If you have an account, you just need to sign in and if you don’t have an account, you have to create an account



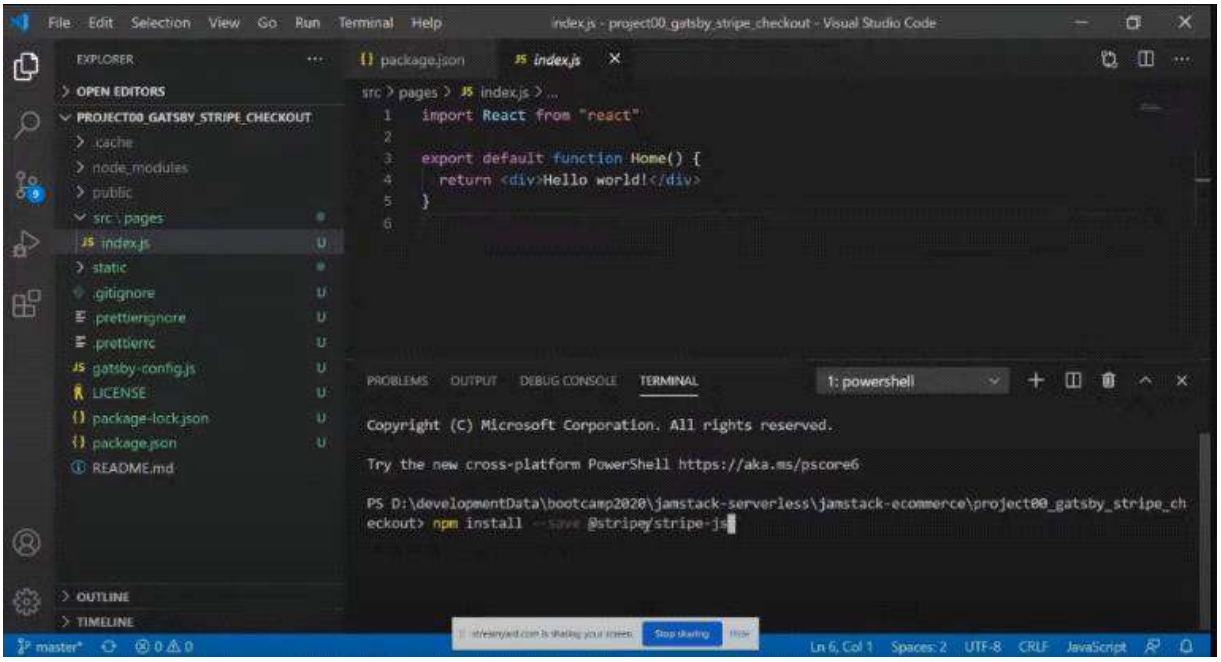
- Then activate your account



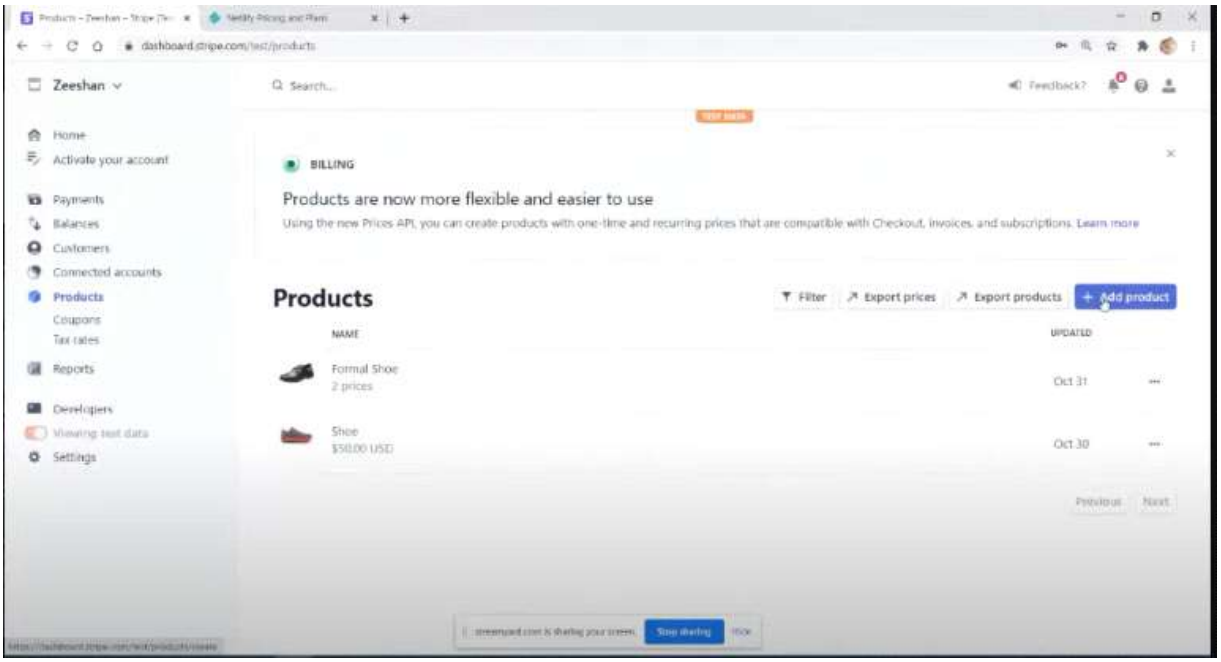
- Go to Gatsby E-Commerce Tutorial
- Install the stripe.js



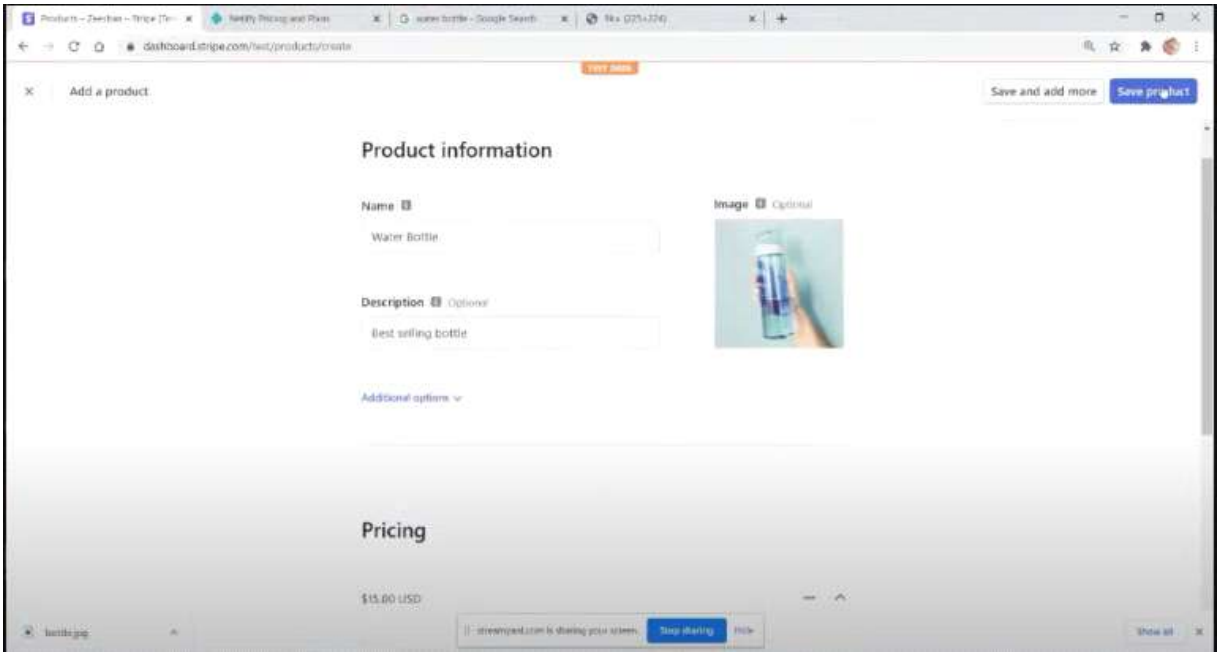
- Go to your project
- Install strip.js here



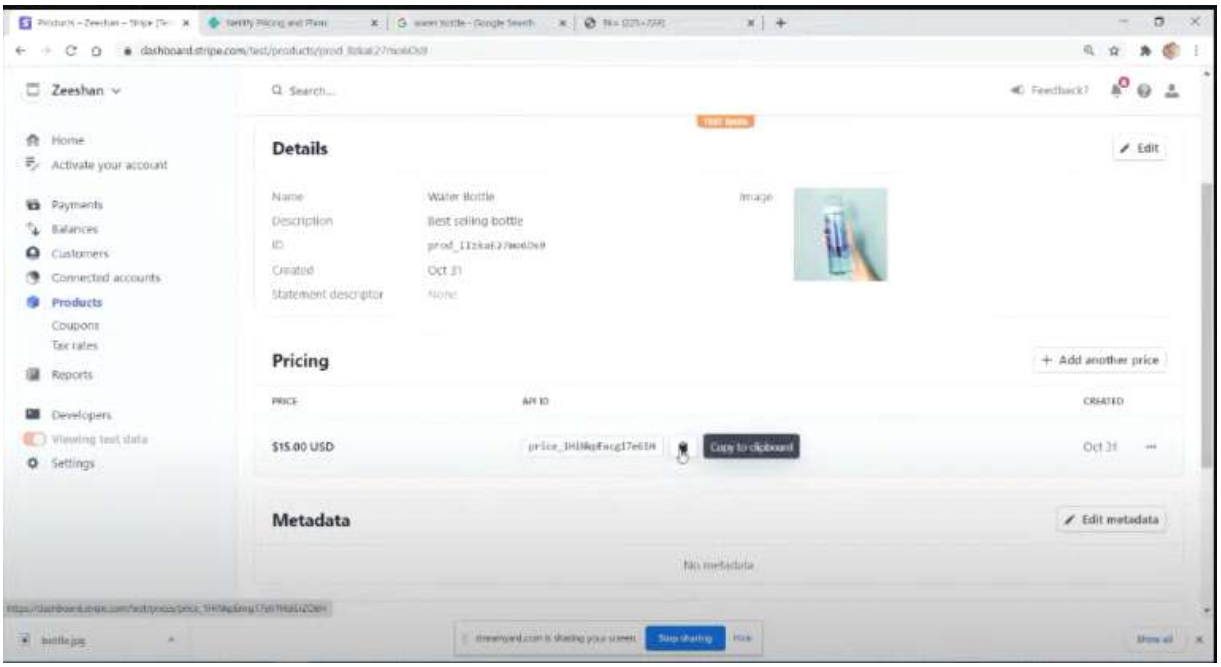
- Go to products in stripe.com
- Add a product



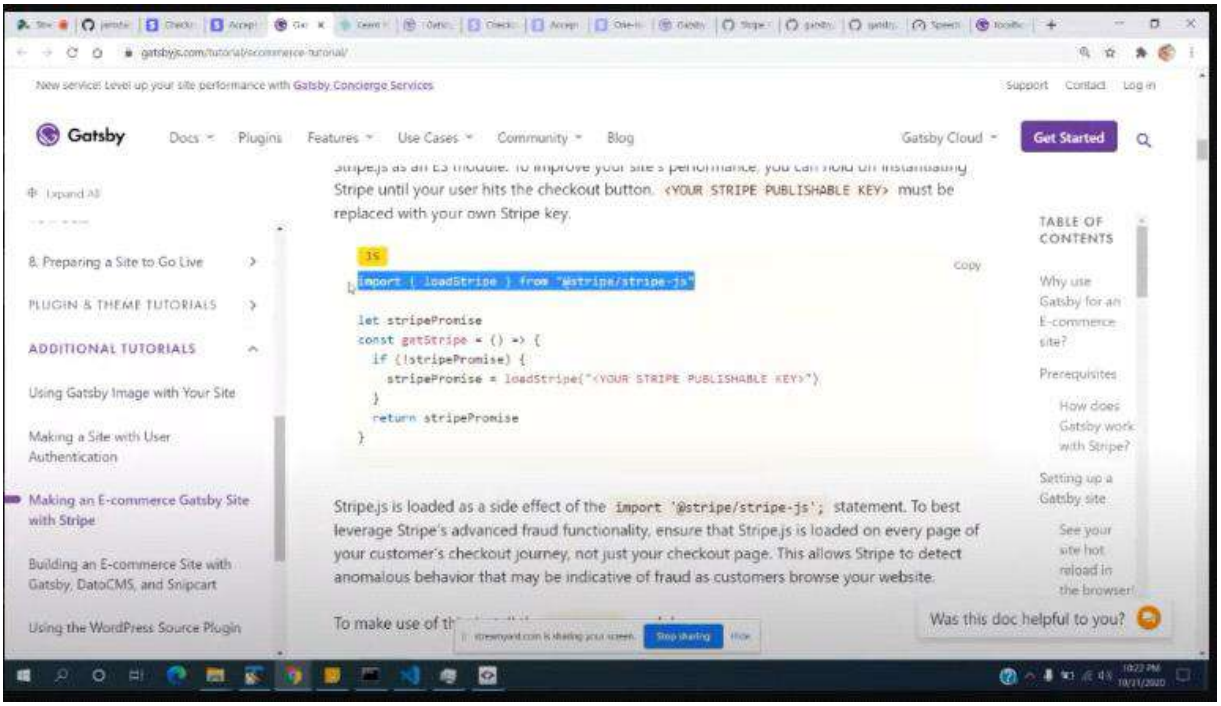
- After adding product information
- Save the project



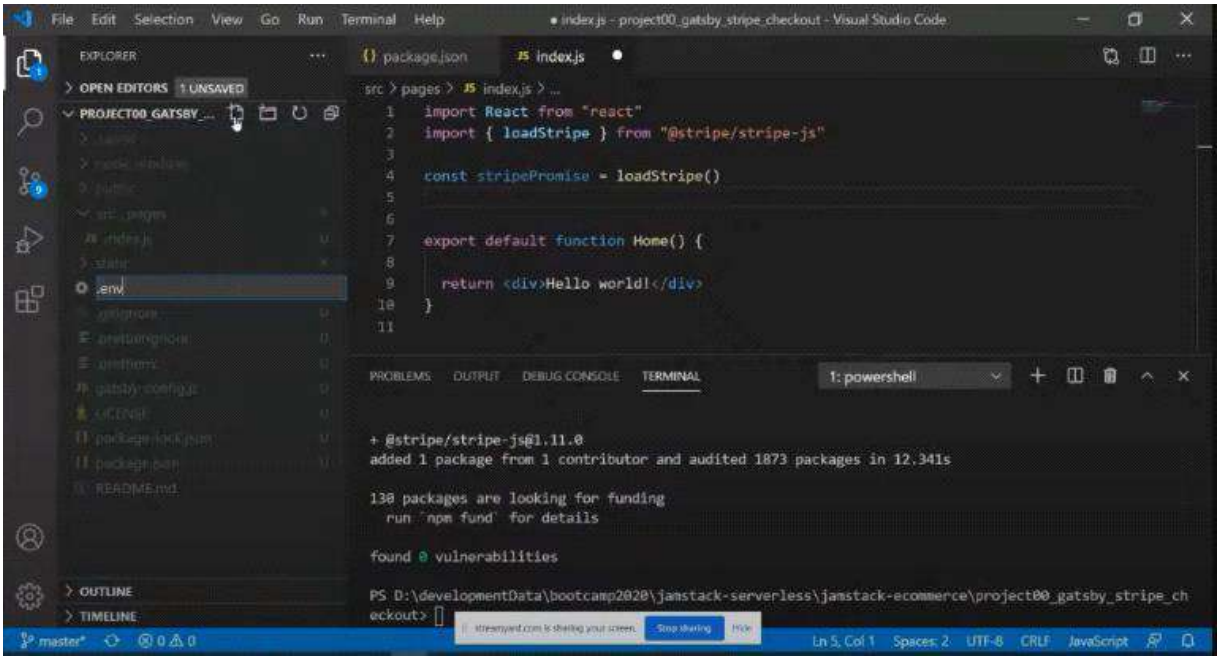
- Copy the pricing ID of your product from main page



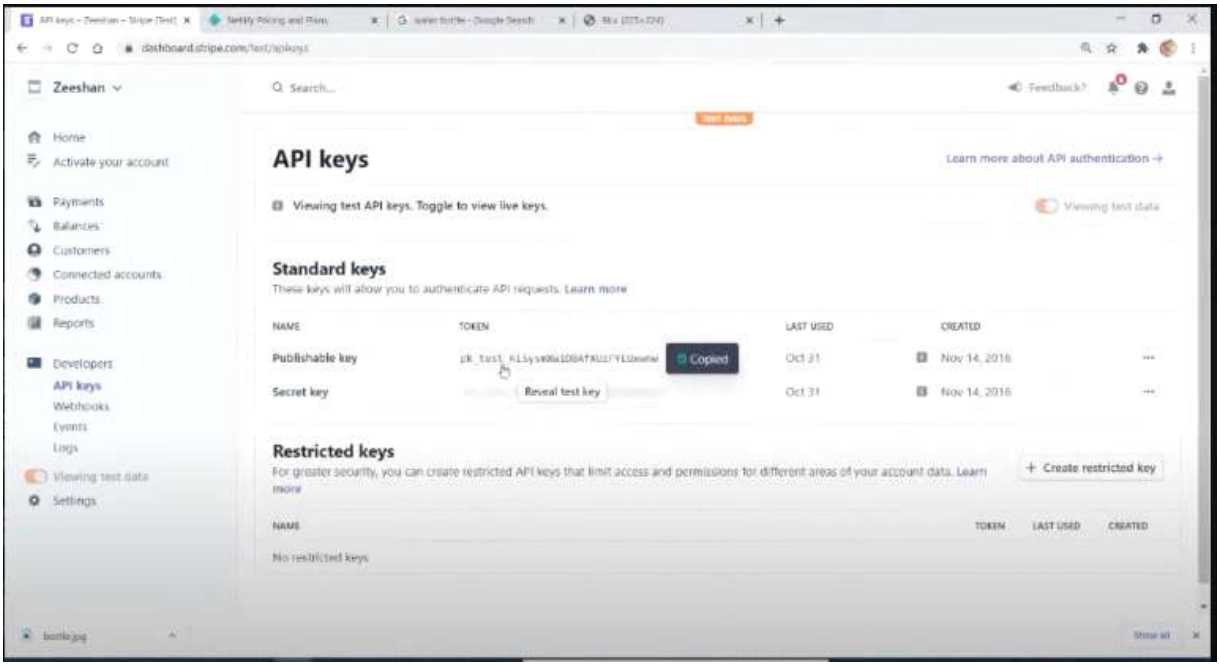
- Import loadstripe from Gatsby tutorial page



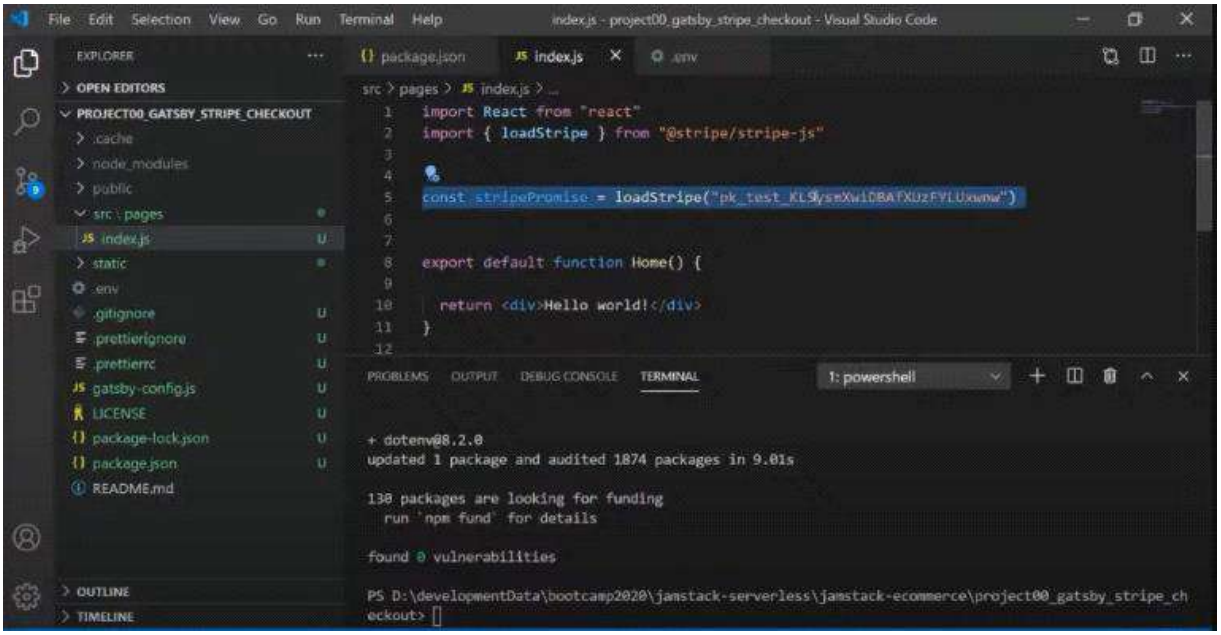
- Then add this in to your project
- Create a new folder ".env"



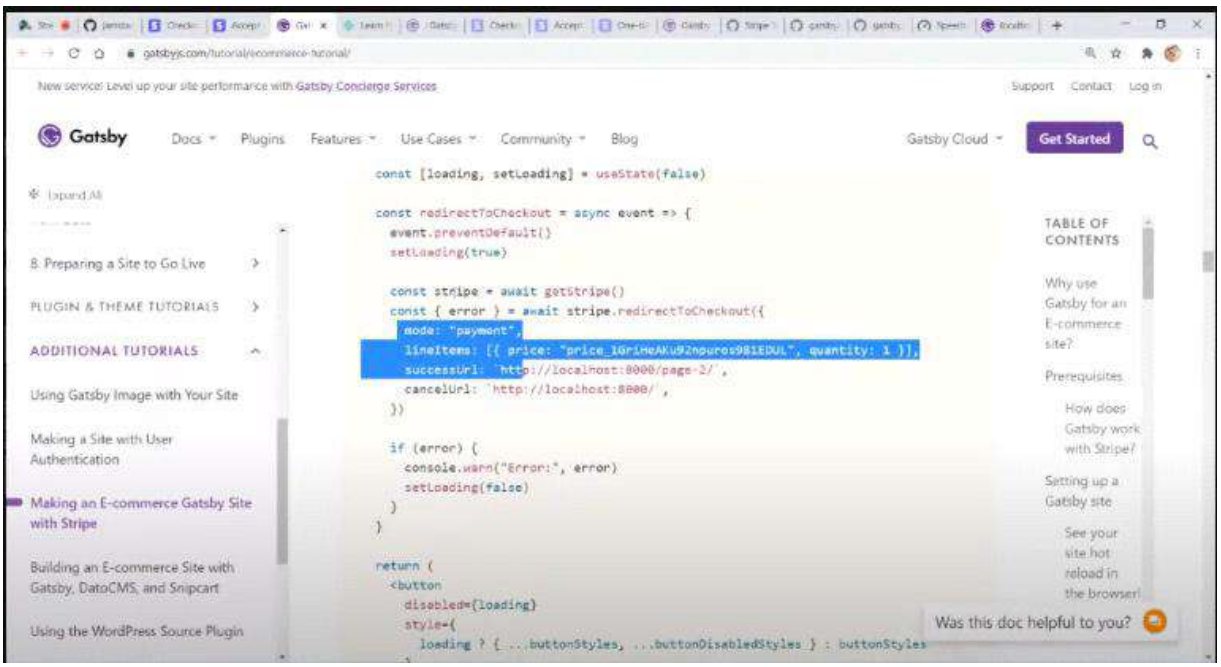
- Then install dotenv
- Go to developers
- Copy the API key



- Put this key here in your project



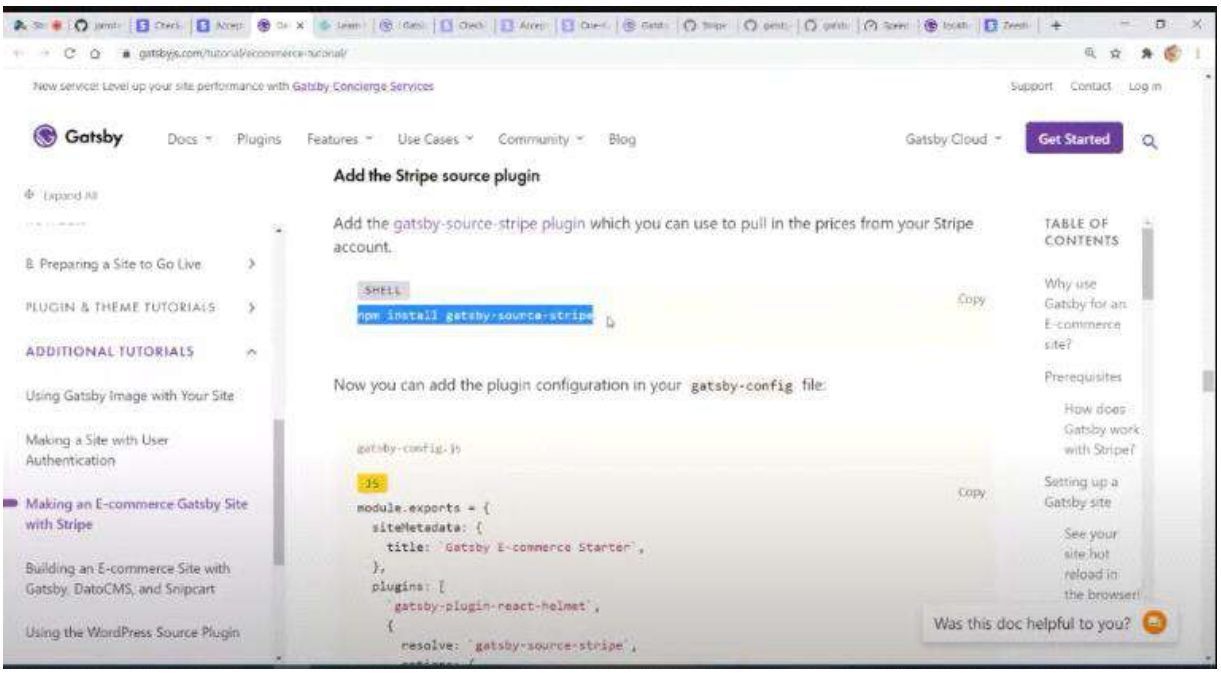
- Then add const stripe
- Add checkout button
- Add async function
- Copy the code from additional tutorial



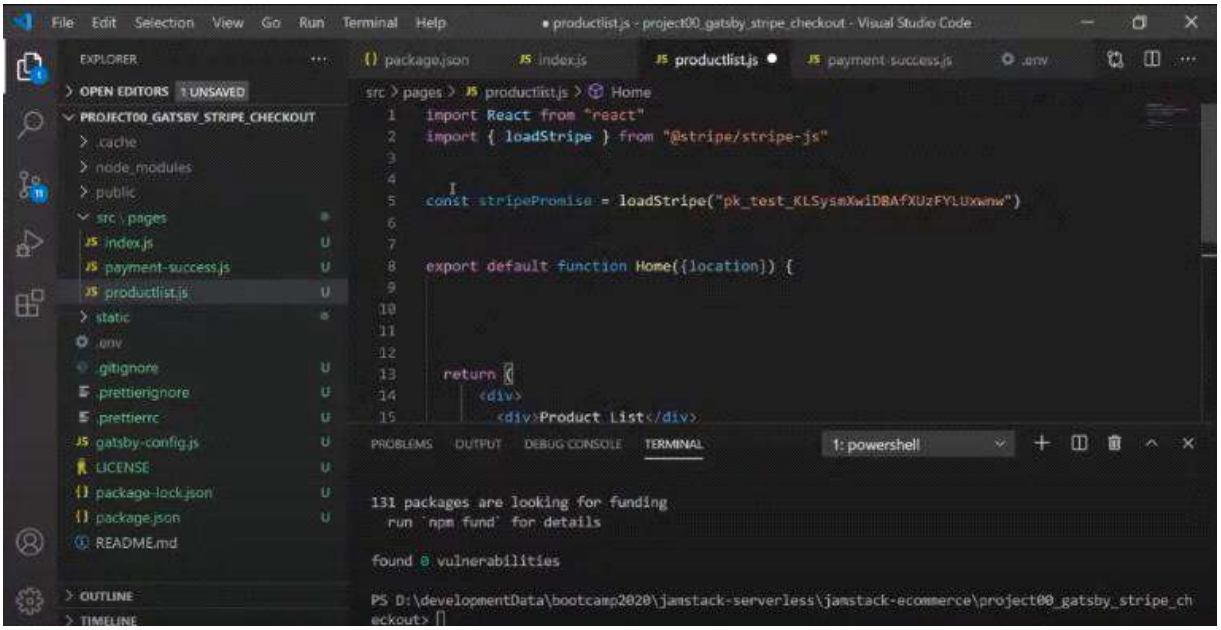
- Put the code in your project

```
export default function Home () (
const stripe = await stripePromise;
const redirectToCheckout = async () => {
 const result = await stripe.reDirectToCheckout({
   node: "payment:,
   lineitems: [( price //// , quantit: 1}],
   successURL: (Add URL)
   cancelURL: (Add URL)
 });
}
```

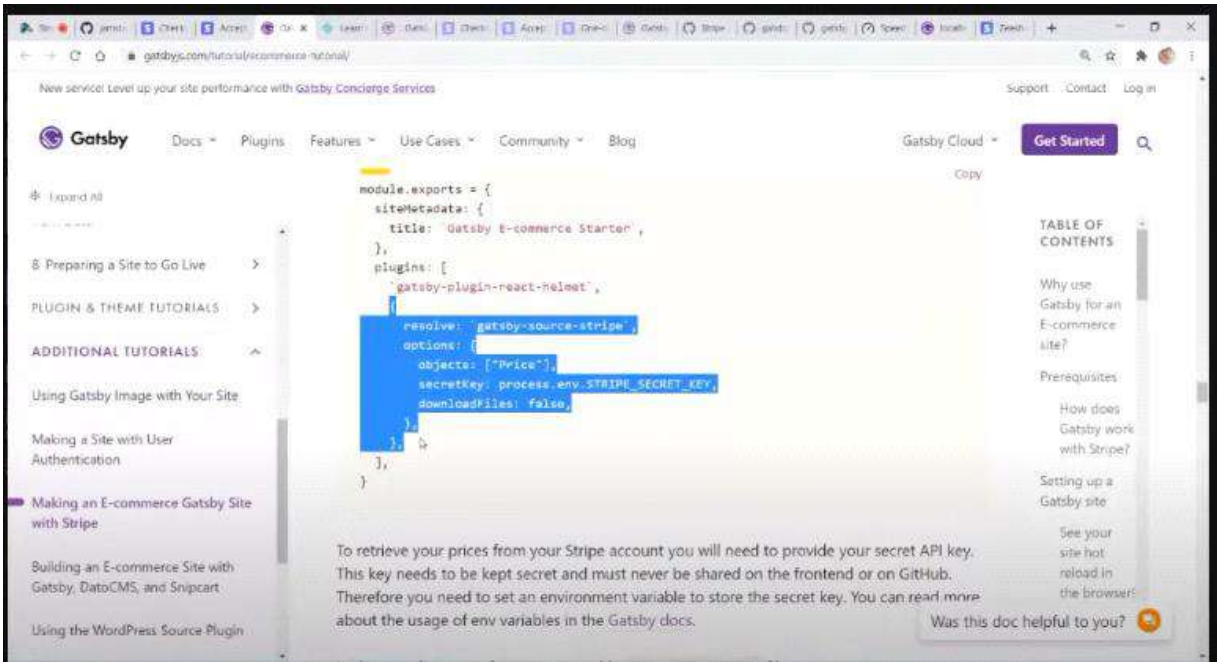
- Use price product key
- Then go to "localhost8888" and check your reults
- Go to your project
- Add a new file “payment-succes”
- If you add multiple products, you just have to repeat the same method
- You can add more products
- And to change the products id from the pag, you just need to copy the other products pricinig key and put it to your project
- Add a location in your project
- Then go to Gatsby source stripe
- Install the plugin



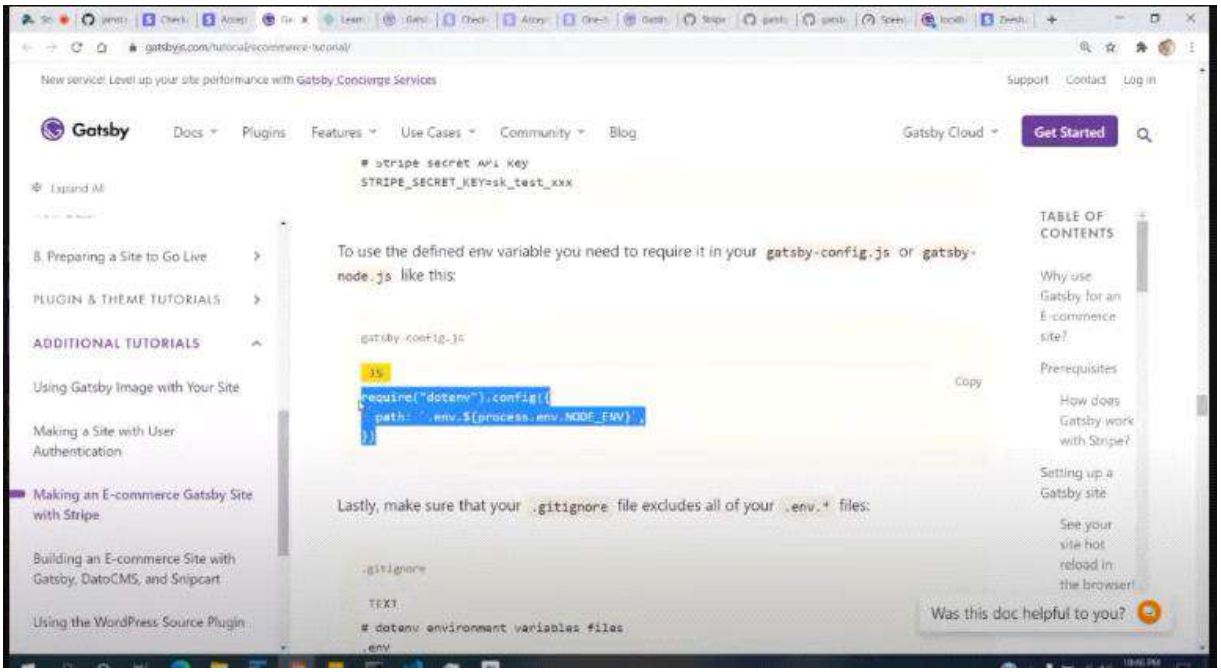
- Install the plugin in your project
- Create a page “productlist.js”
- Copy the code from index.js
- Put the code in product.js
- Make the change in the code



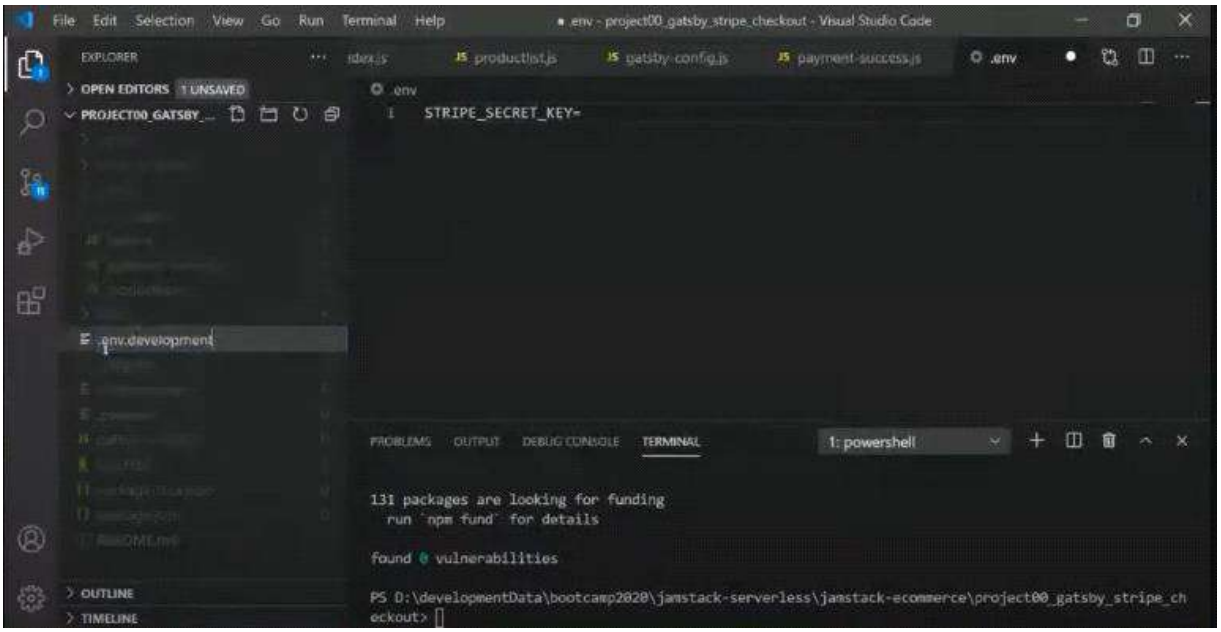
- Copy the code from Gatsby



- Then go to gatsby-config in your project and put the code here
- Now copy the code



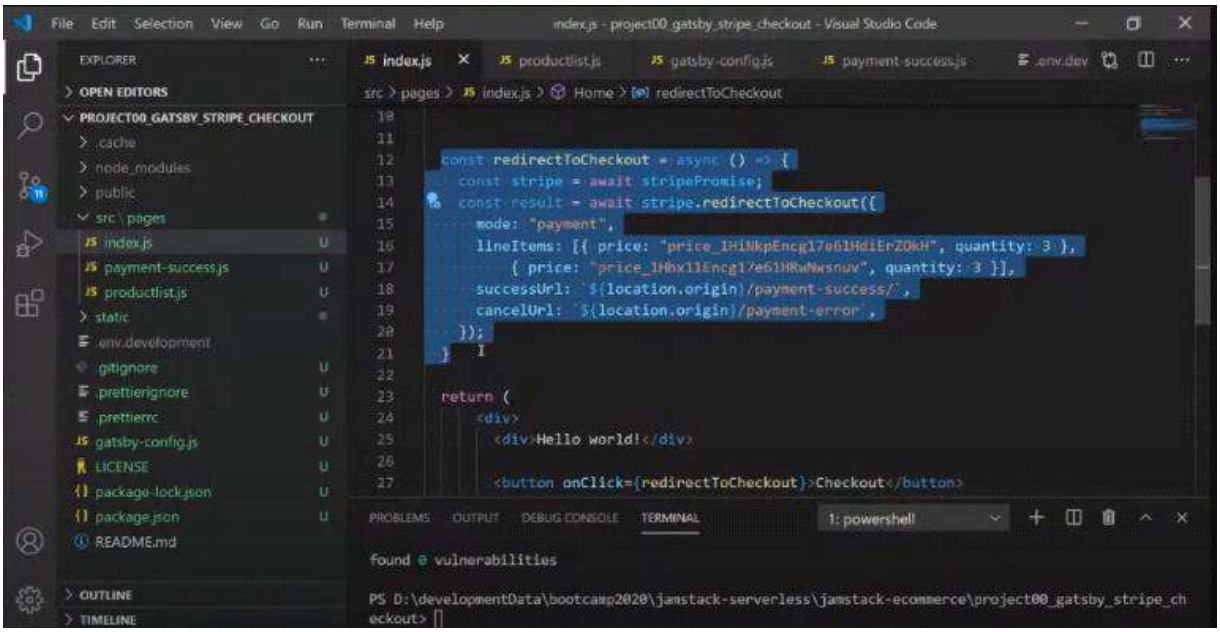
- Put the code in gatsby-config.js
- Change the file name .env to “.env.development”



- Then go reveal the secret key
- And put that key in ..env development
- Update the code in productlist.js

```
graphql
    query productprices {
      prices : allstripeprice {
         edges {
             id
             active
             currency
             unit_amount
             product {
                  id
                  name
           }
       }
}
```

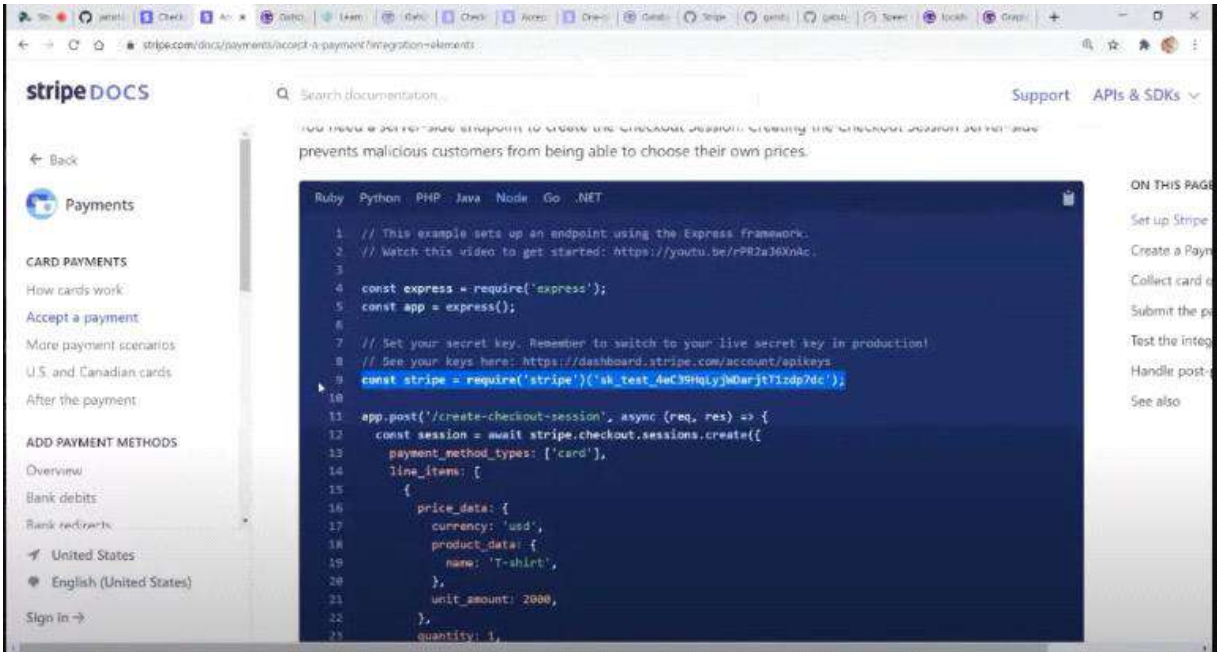
- Then go to "localhost8888" and check your updates
- Make some other additions in productlist.js
  Copy the code from index.js



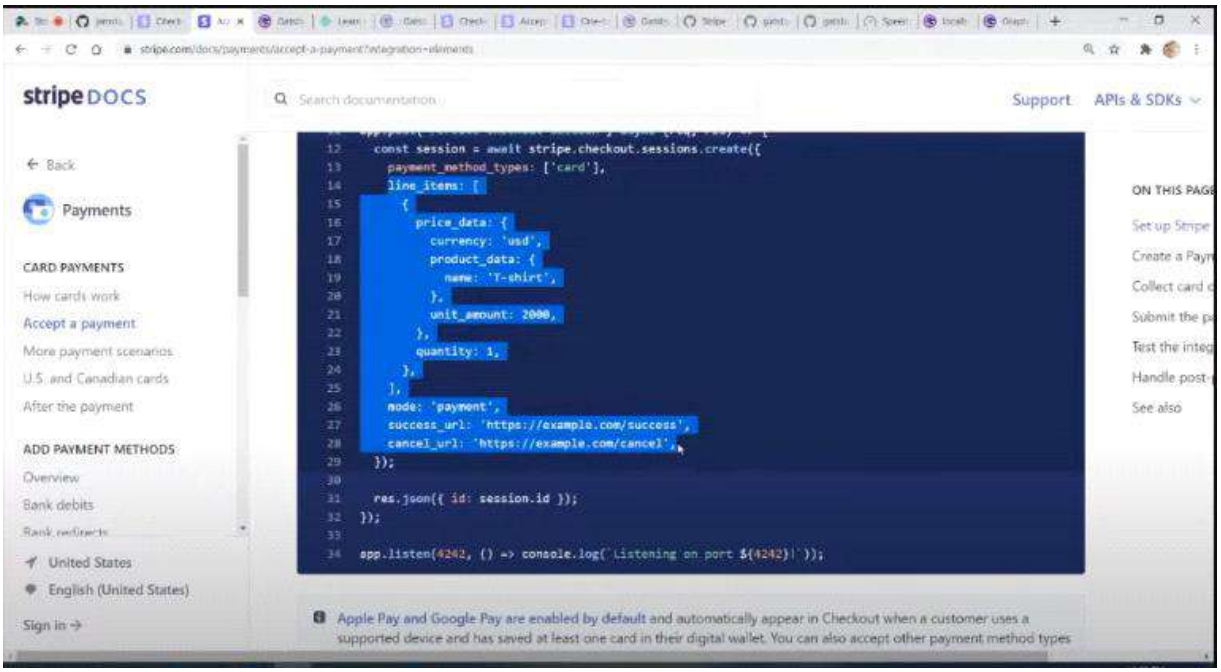
- Put the code in productlist.js
- Make some changes in that code
- Then add a new file “netlify.toml”
- Develop this in netlify.toml

```
[build]
  command="gatsby build"
  function="functions"
```

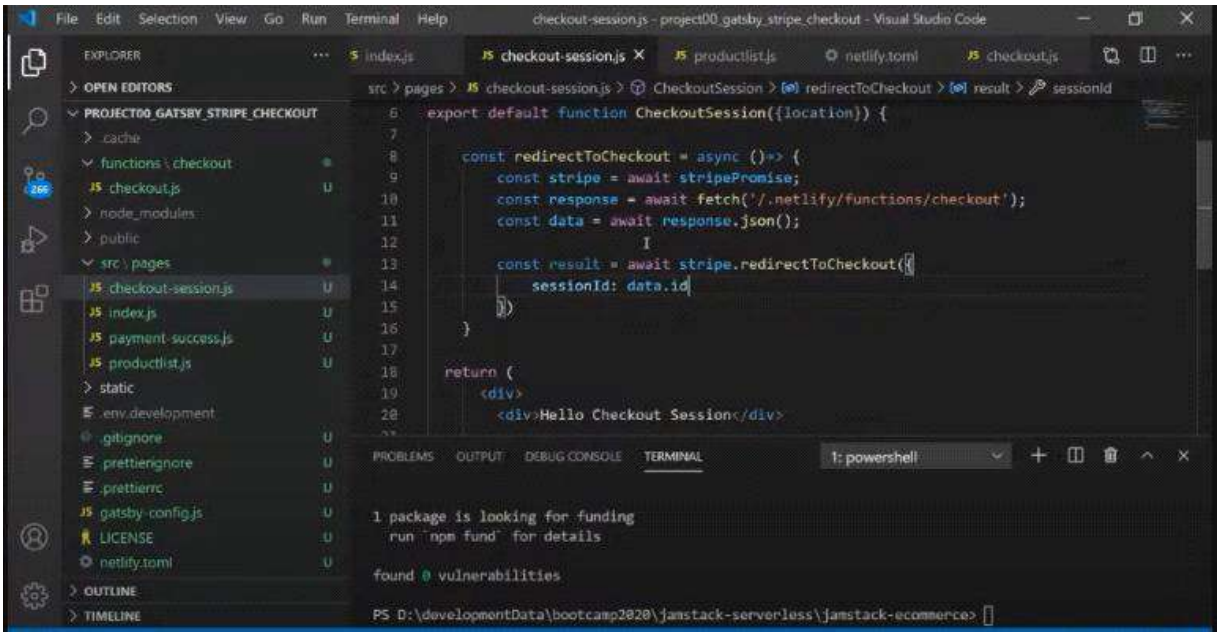
- Then add a new file “functions”
- Then run the command function in terminal
- Copy the code



- Add the code in checkout.js
- Add cons session
- Then copy the line item code



- Remove the price data from checkout.js code
- Create a new file “checkout-session.js”
- Add index.js code in this folder
- Make some changes in the code



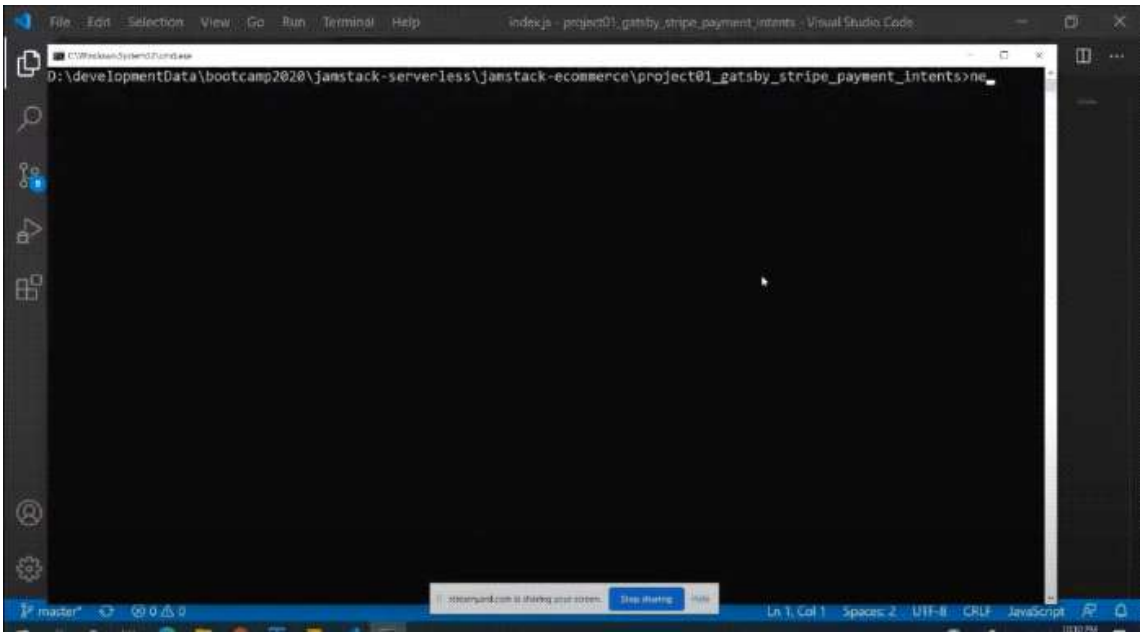
- Then go to "localhost8888" and check your development

Session- 44

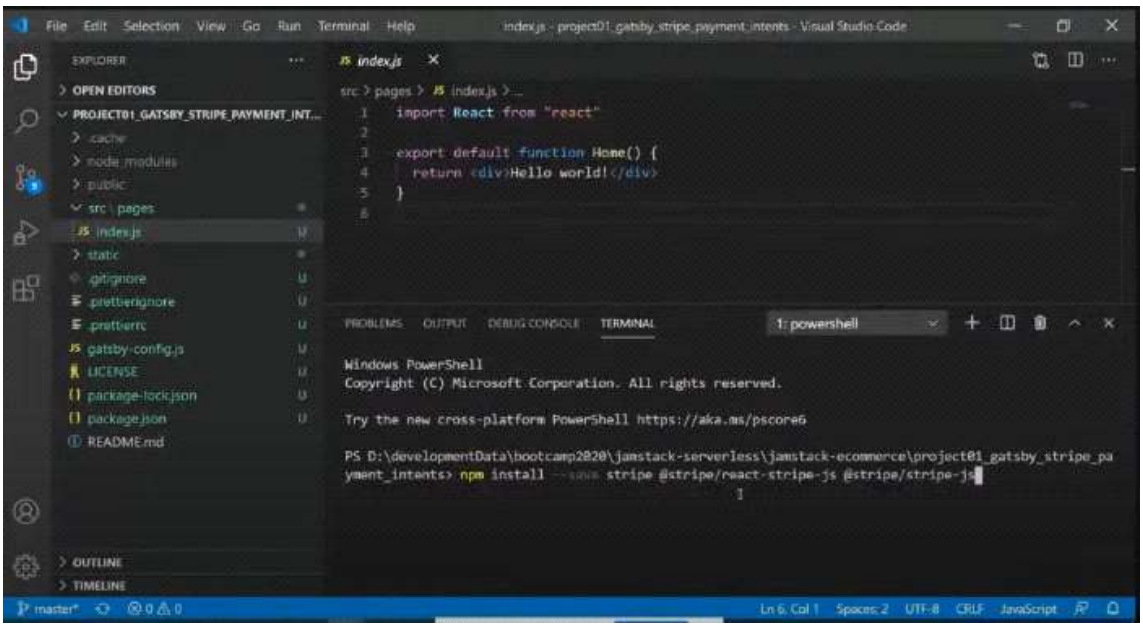
Payments With Stripe API

Steps:

1. Open cmd.exe and type netlify dev to run the program.



2.Open terminal in index.js and type npm install --save stripe and other libraries



3. Create a new toml file and name it as netlify and information to it.

[build]

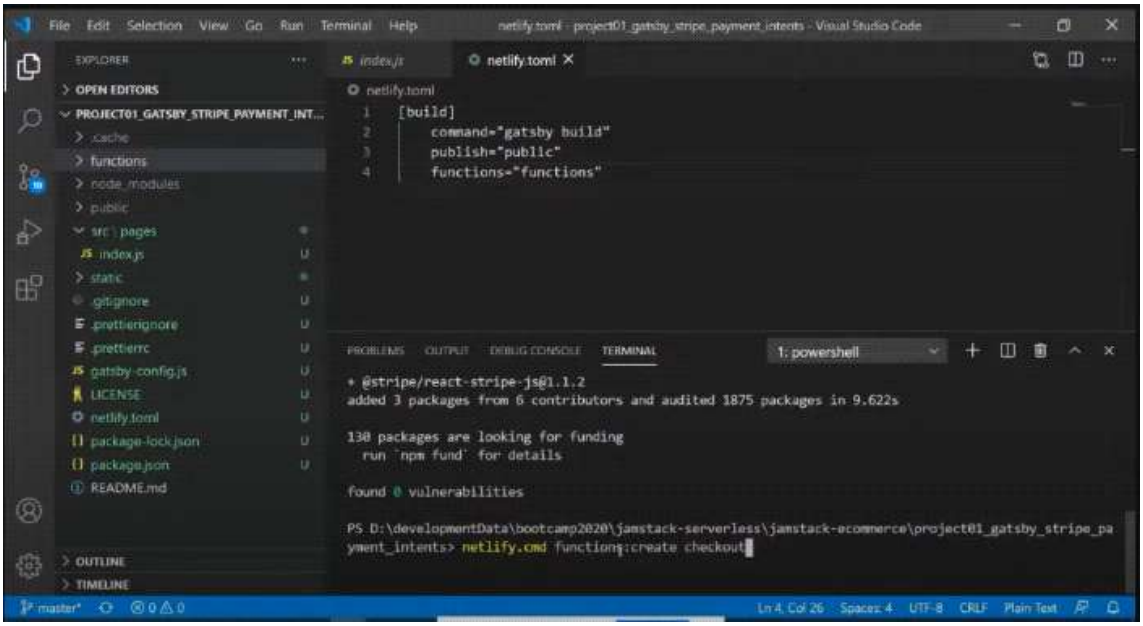
Command= “Gatsby build”

Publish= “public”

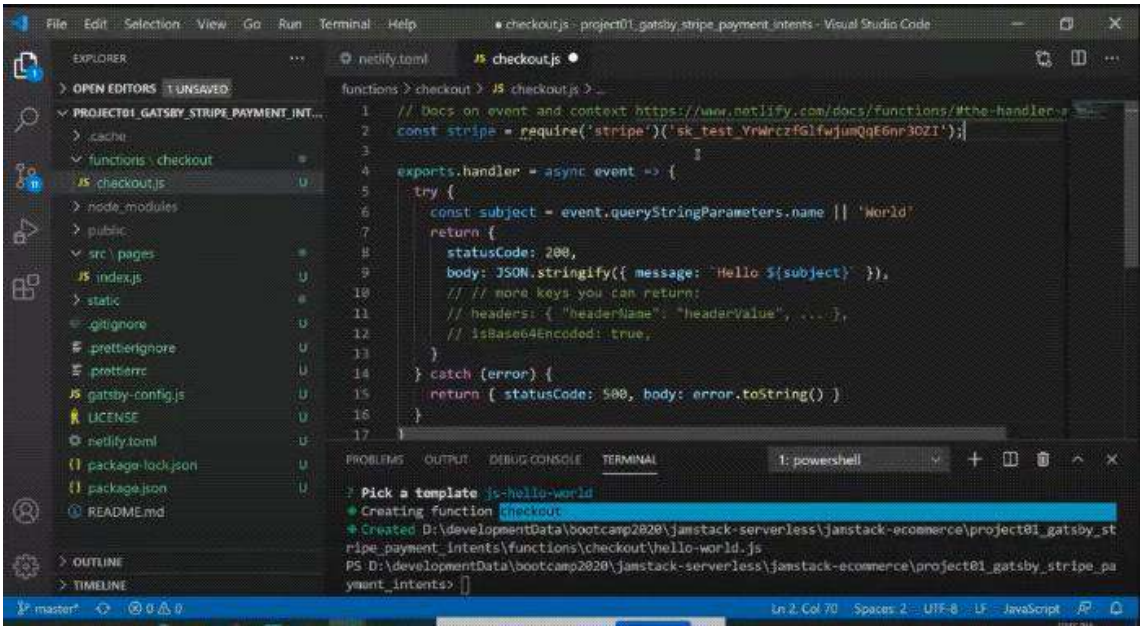
Functions= “functions”

4. Create a new folder and name it as Functions.

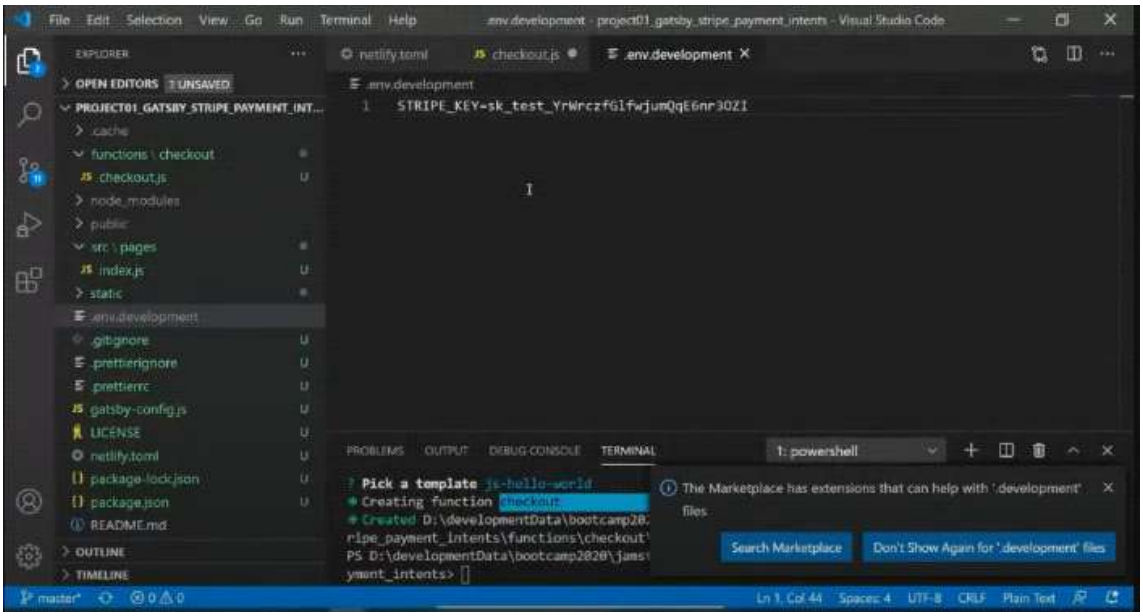
5. Add netlify.cmd functions: create checkout in terminal.



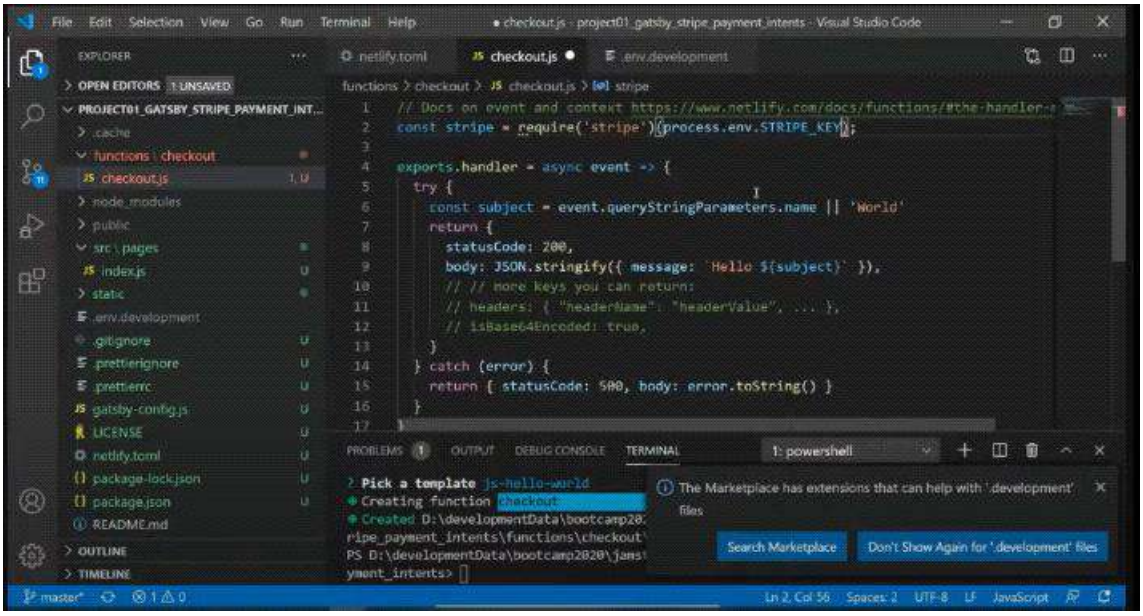
6. Choose checkout from terminal and and paste the required stripe in checkout.js



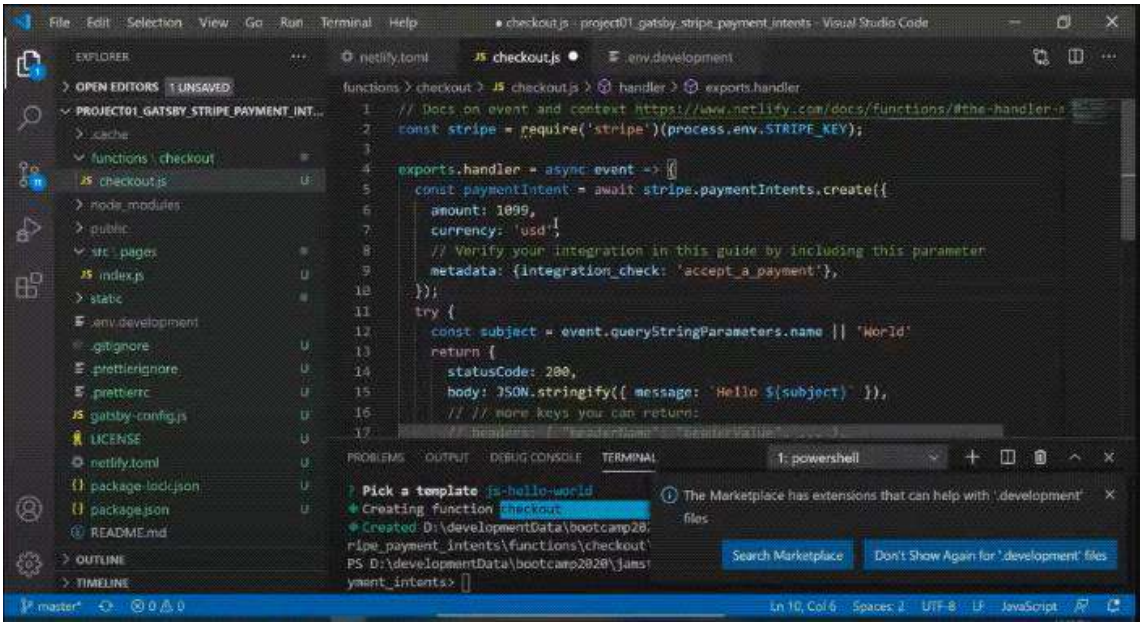
7. Create a file and name it .env.developement and add any stripe key.



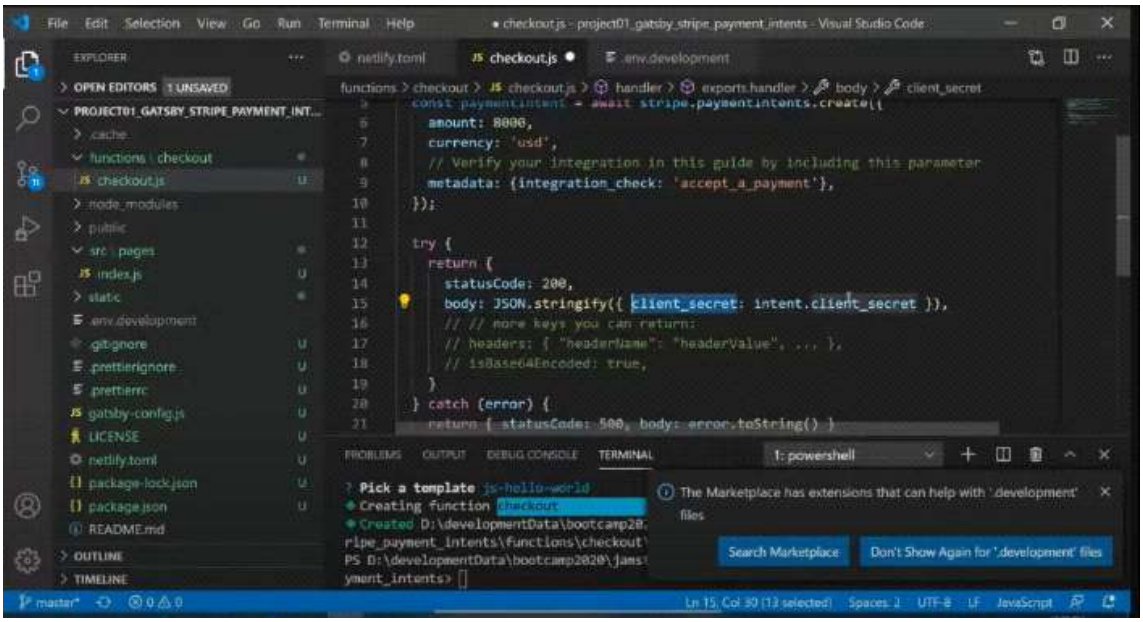
8. Open checkout.js and add process.env.STRIPE_KEY



9. Copy and paste the payment intent in checkout.js



10. In checkout.js remove const subject and add client secret in body: JSON



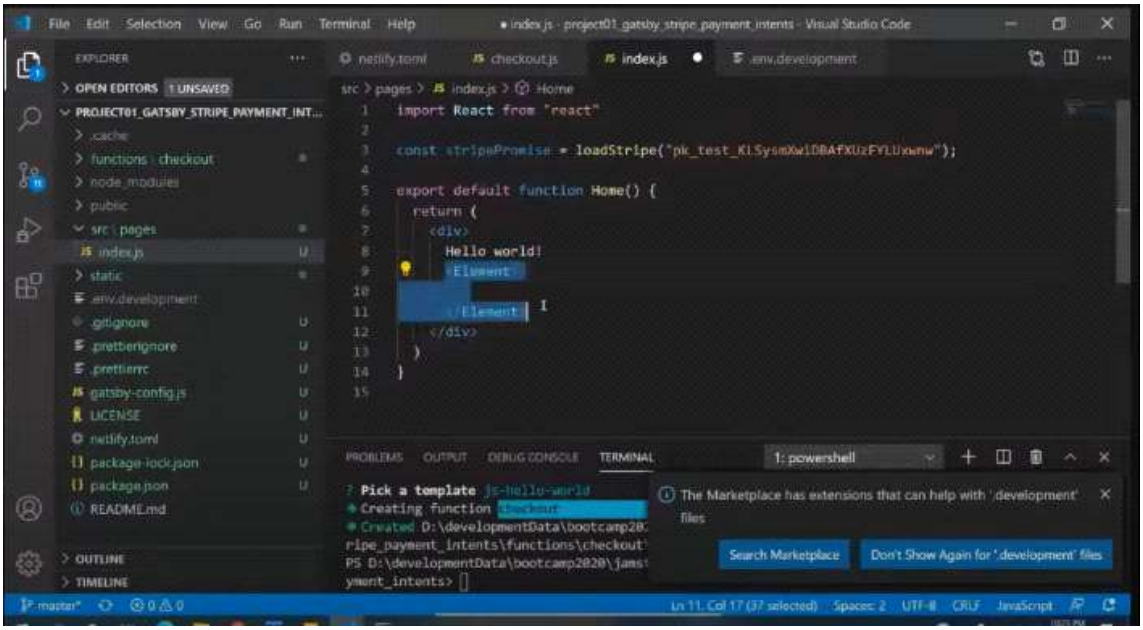
11. Open src/pages, then index.js and add a div.

```
import React from "react"

export default function Home () (
  return (
    <div>
      Hello world
    <div>
  )
)
```

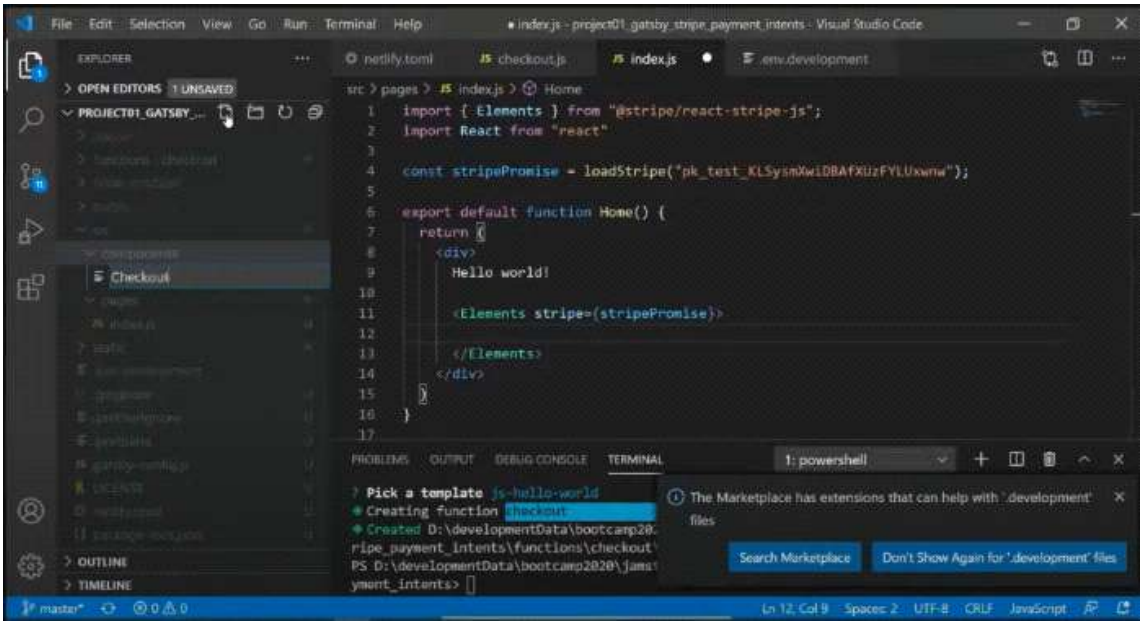
12. Add const stripePromise in index.js

```
import React from "react"
const stripePromise = loadstripe ( add stripe );
export default function Home () (
  return (
    <div>
      Hello world
    <div>
  )
)
```

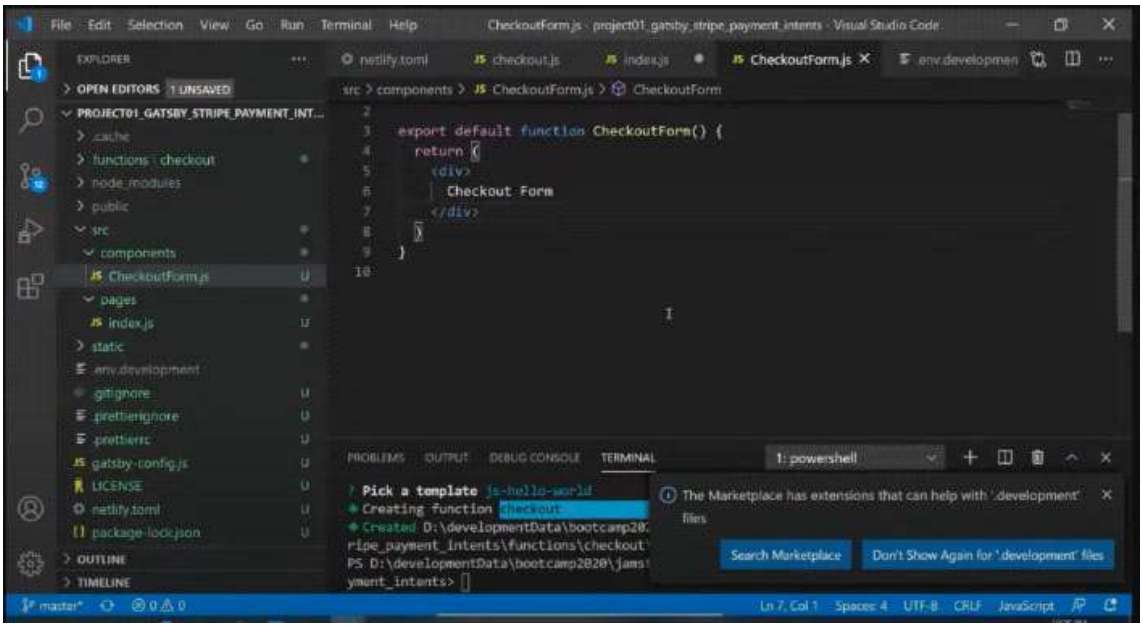
13. Add stripe element in index.js and add stripe promise.



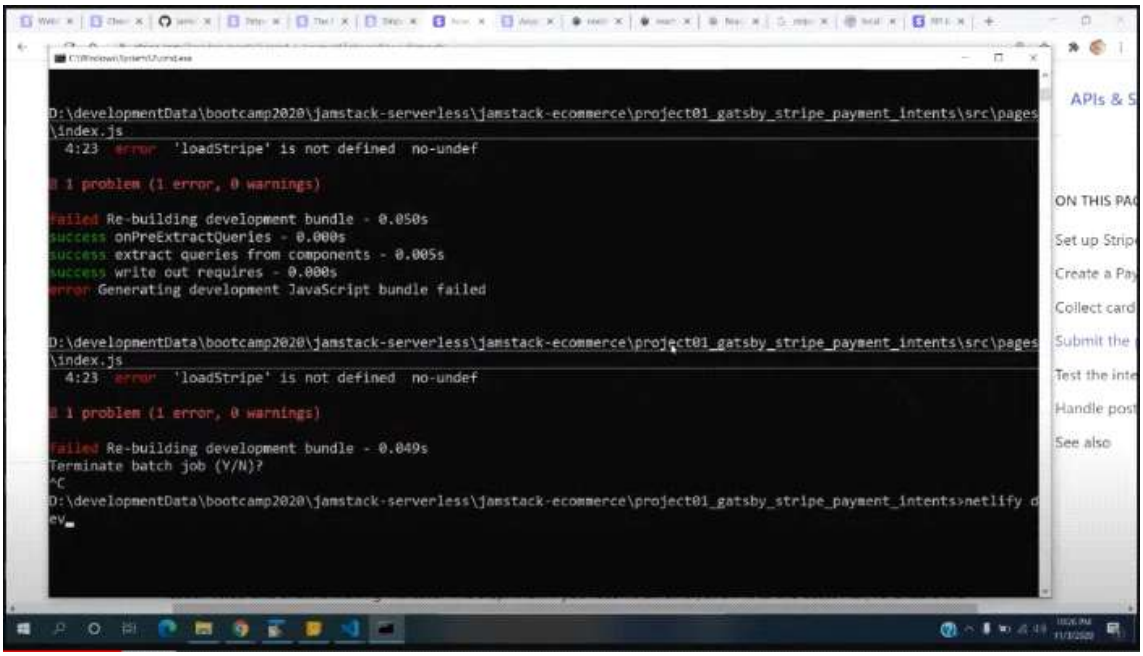
14. Create a new folder and name it as components then create a new file named as checkout form.js



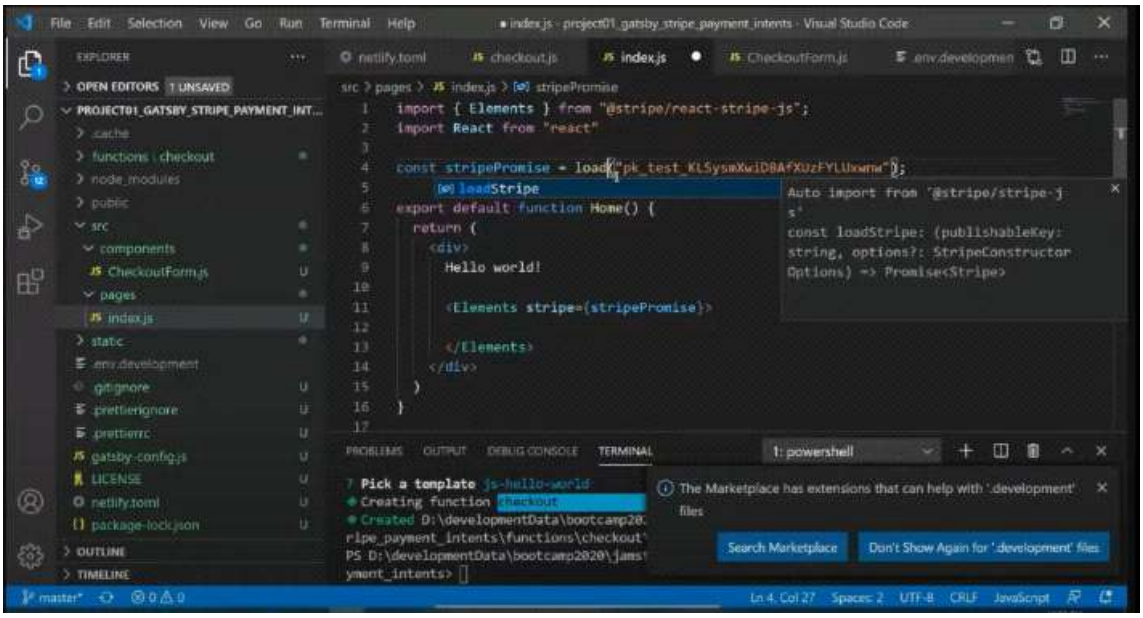
15. Add checkout form in function and div.



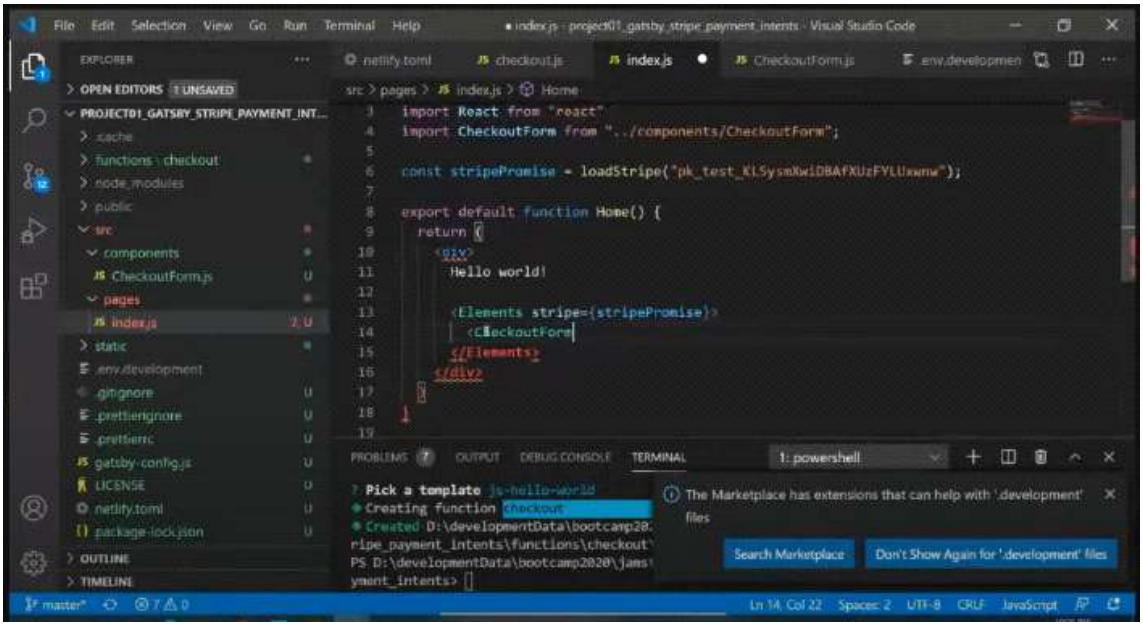
16. Run the code in cmd.exe



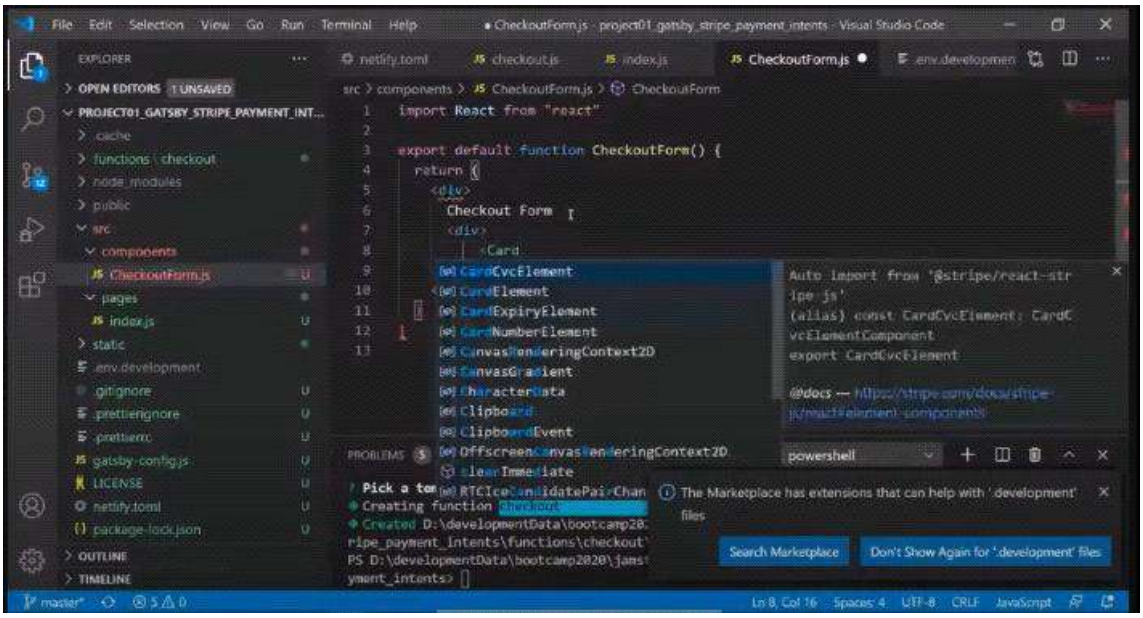
17. Open index.js and replace load stripe with load.



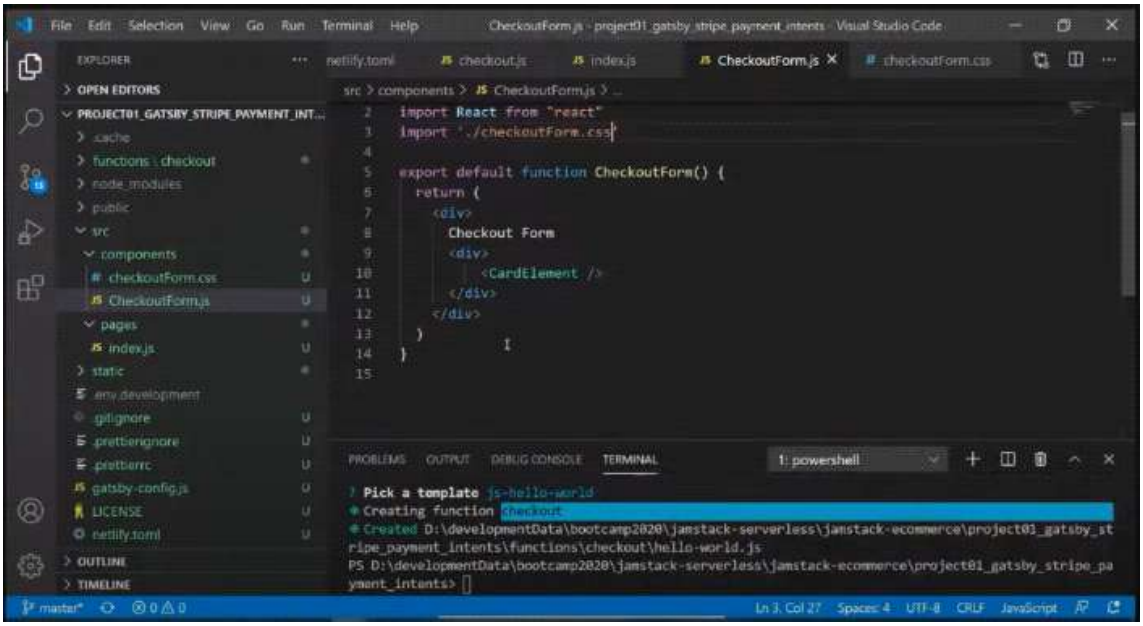
18. Open index.js and add Checkout form element.



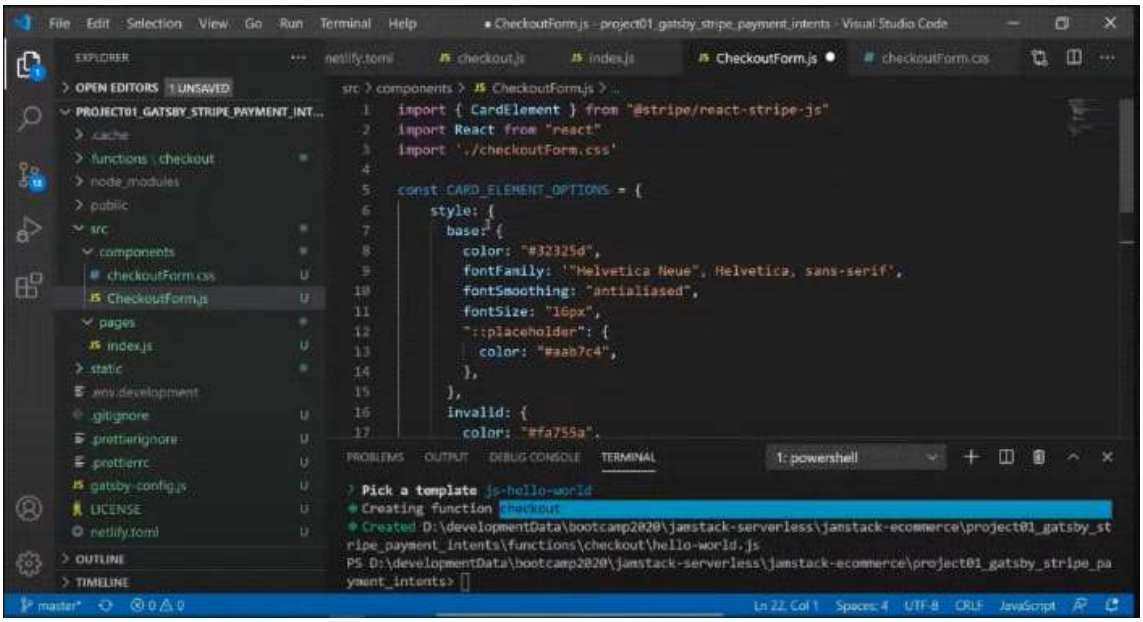
19. Open checkoutform.js and add a div.



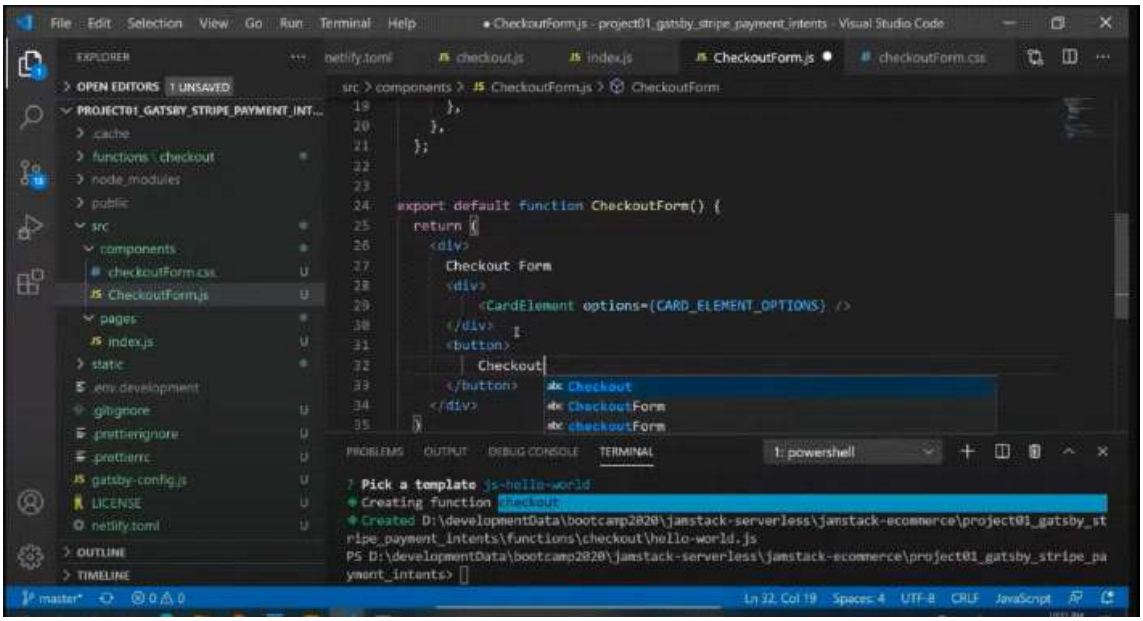
20. Create a new css file and name it as checkout form.css and import it in checkout form.js



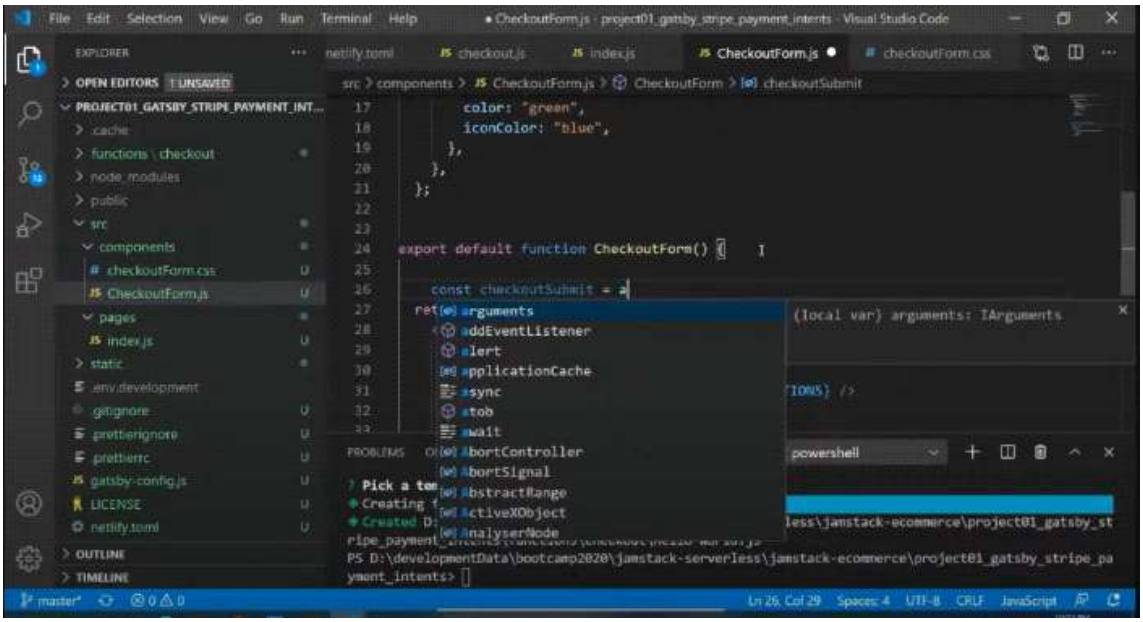
21. Copy card element options and paste in checkoutform.js and make the required changes.



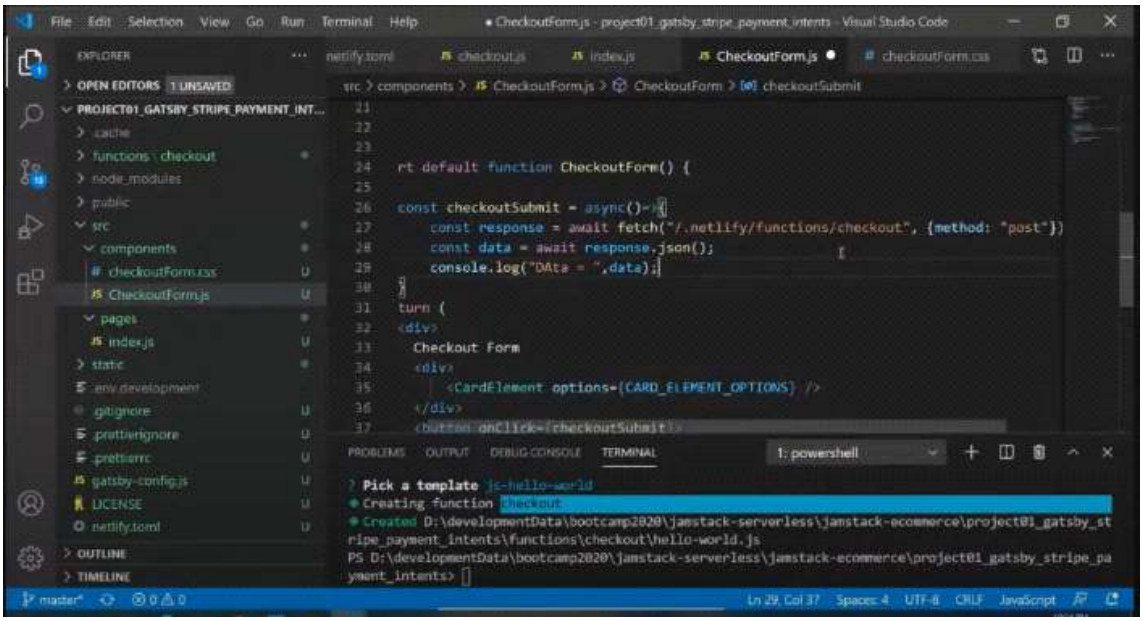
22. Create a button in checkout.js



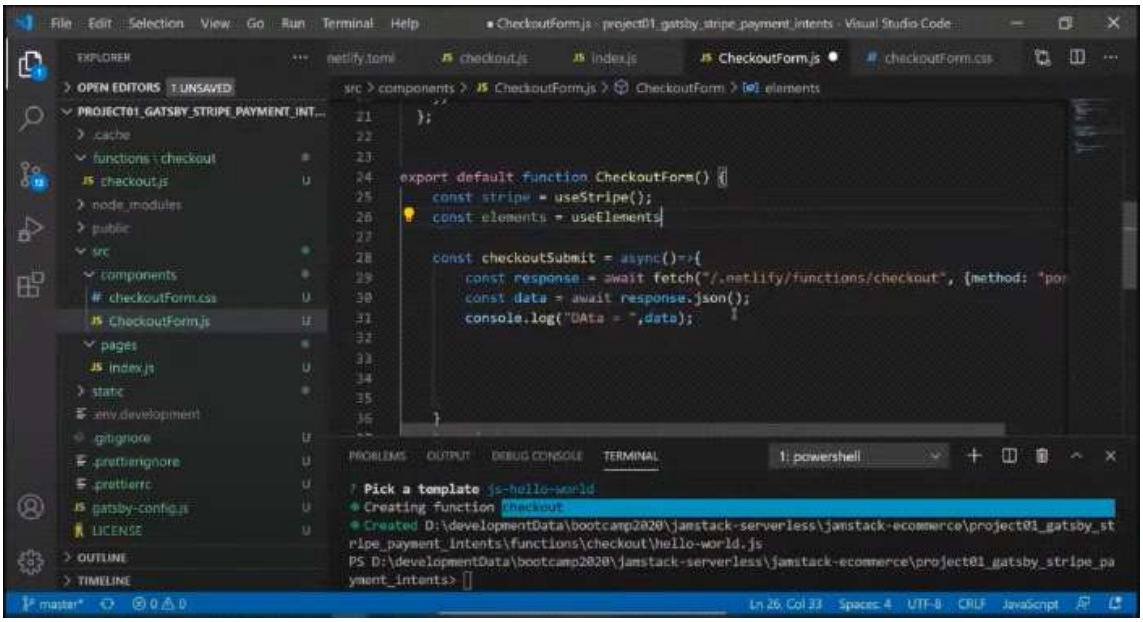
23. Make further required changes in the codes



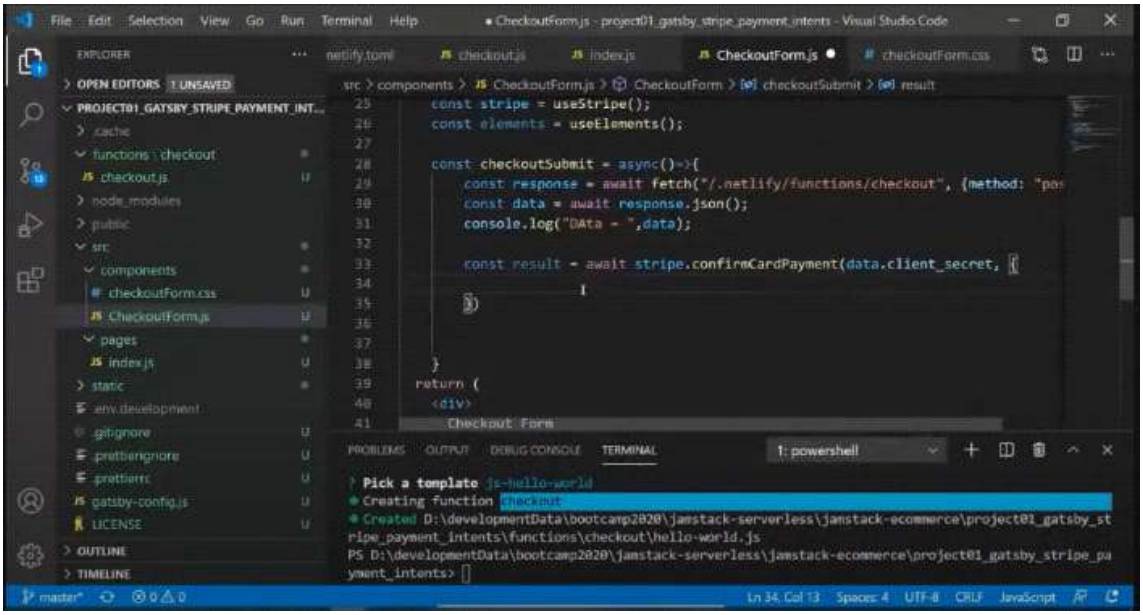
24. Open checkouform.js and make a code to make the checkout button work.



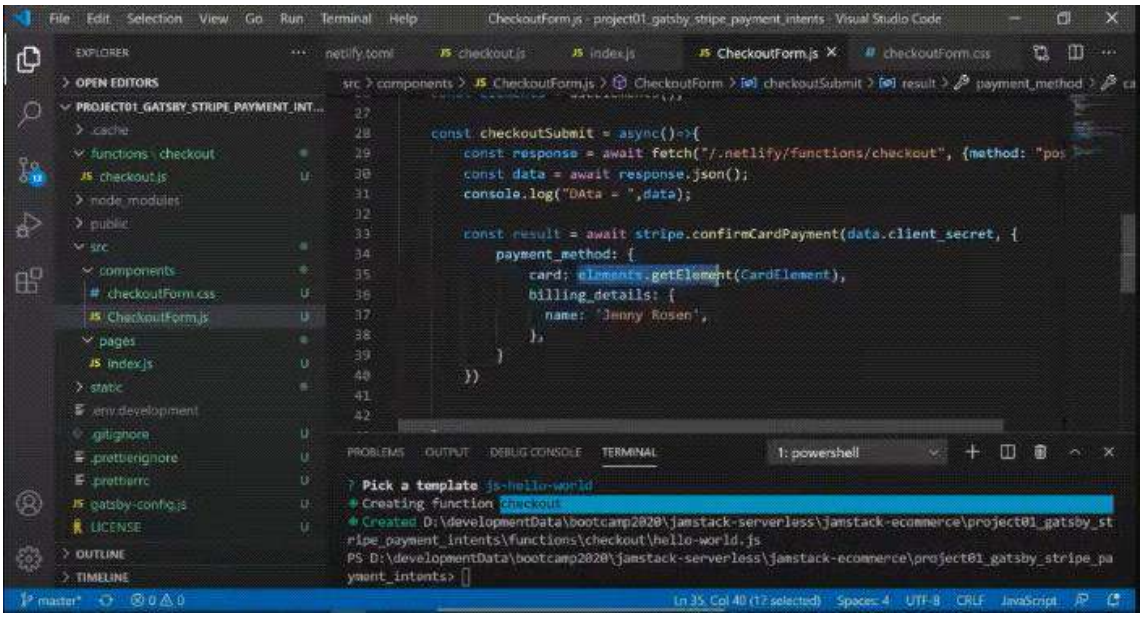
25. Open checkoutform.js and add stripe and element



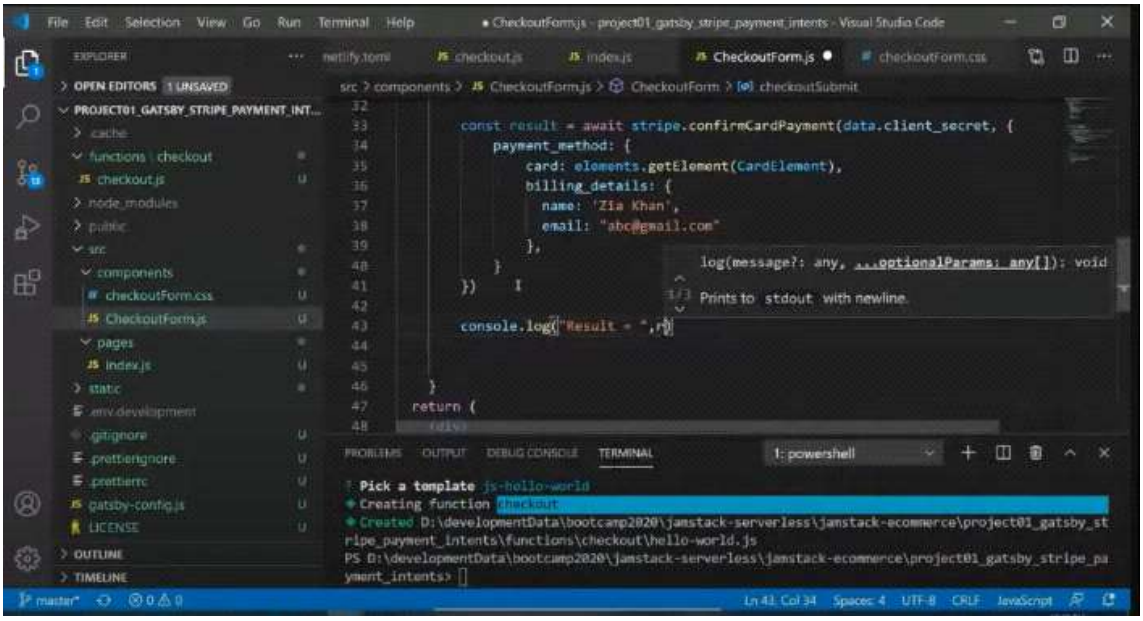
26. Open checkoutform.js and add result



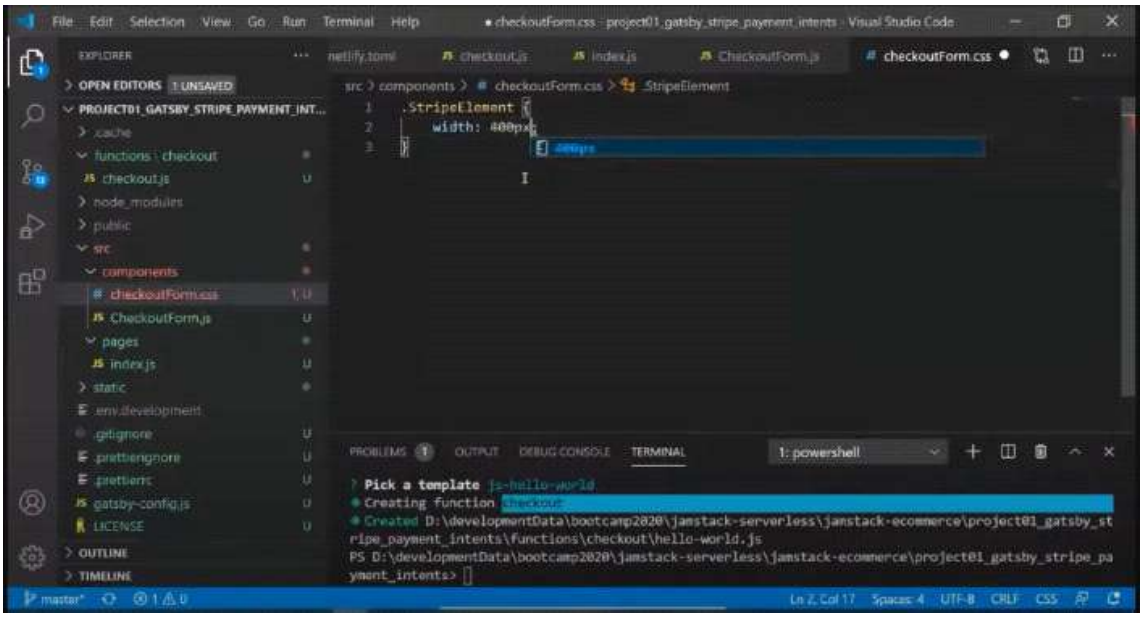
27. Copy payment methond code and paste it in checkoutform.js



28. Add console log in checkoutform.js

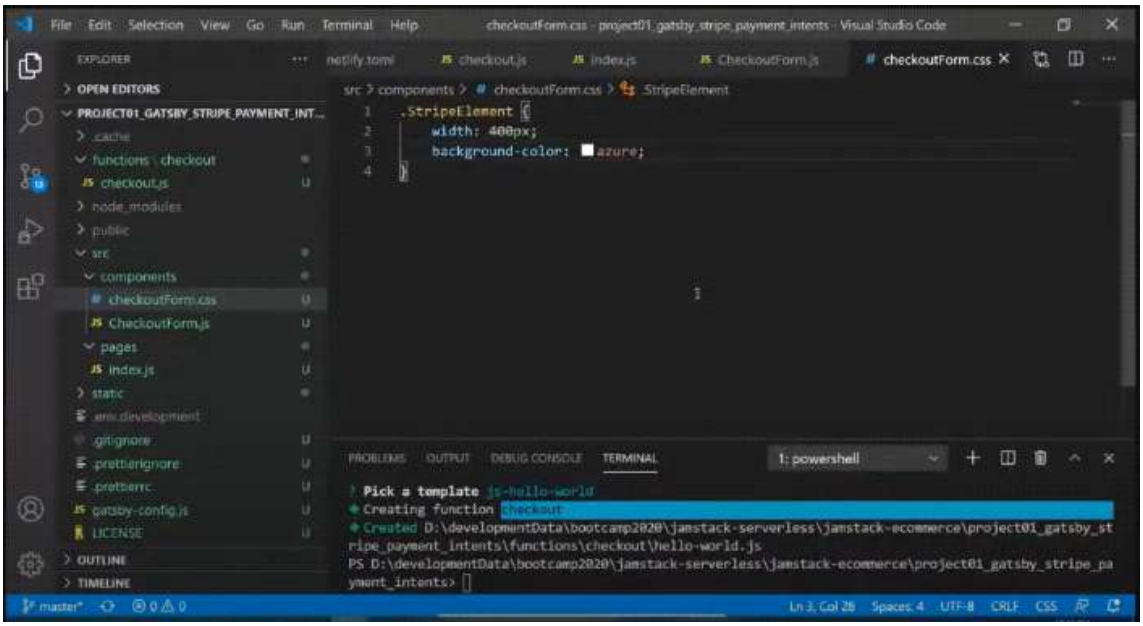


29. Open checkoutform.css and adjust width.

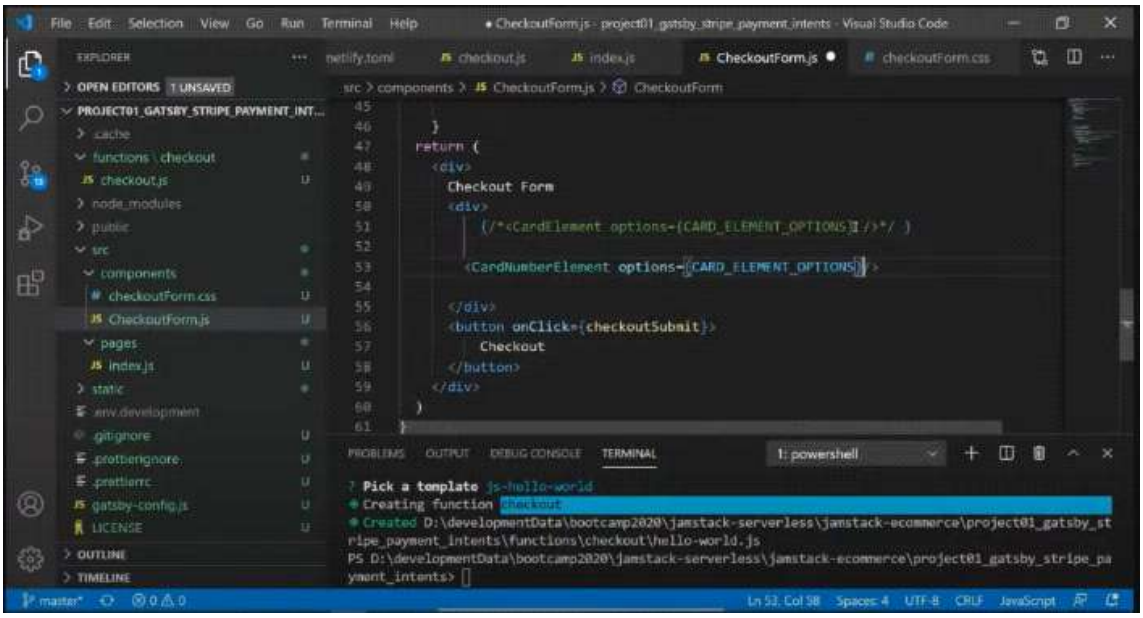


30. Change the backgroundcolour according to your liking

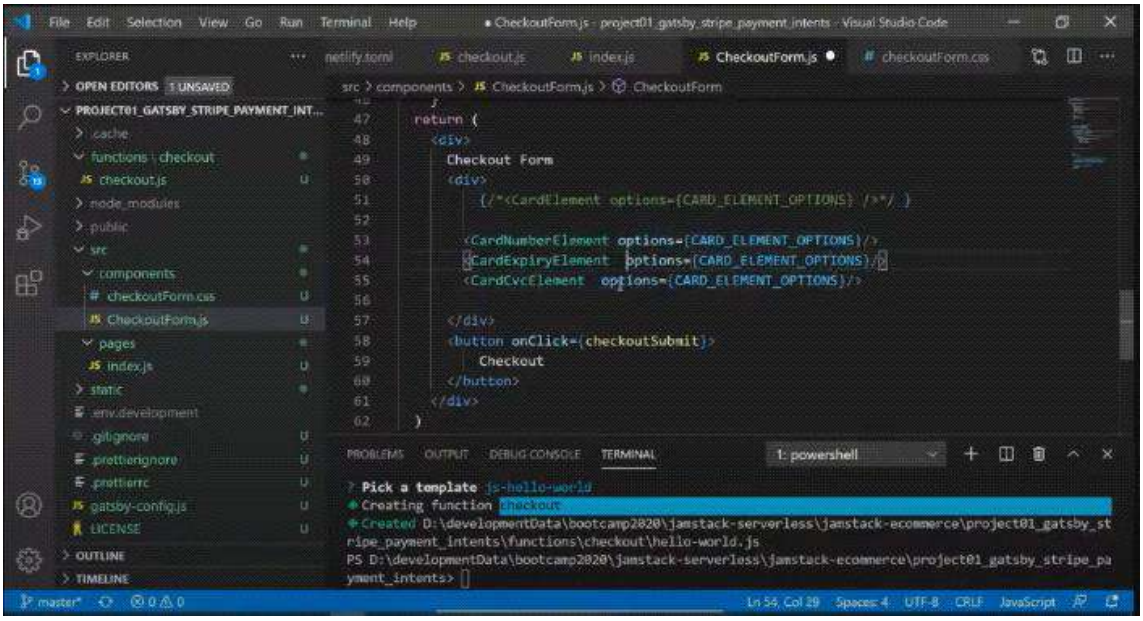
in checkoutform.css



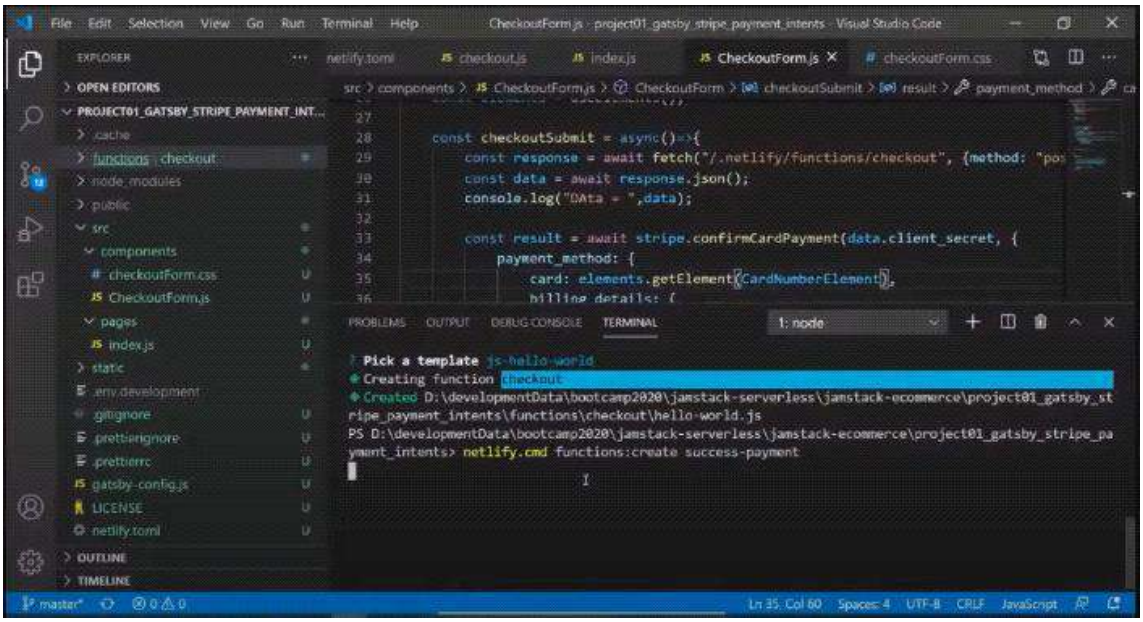
31. Open checkoutform.js and add a Cardnumber element.



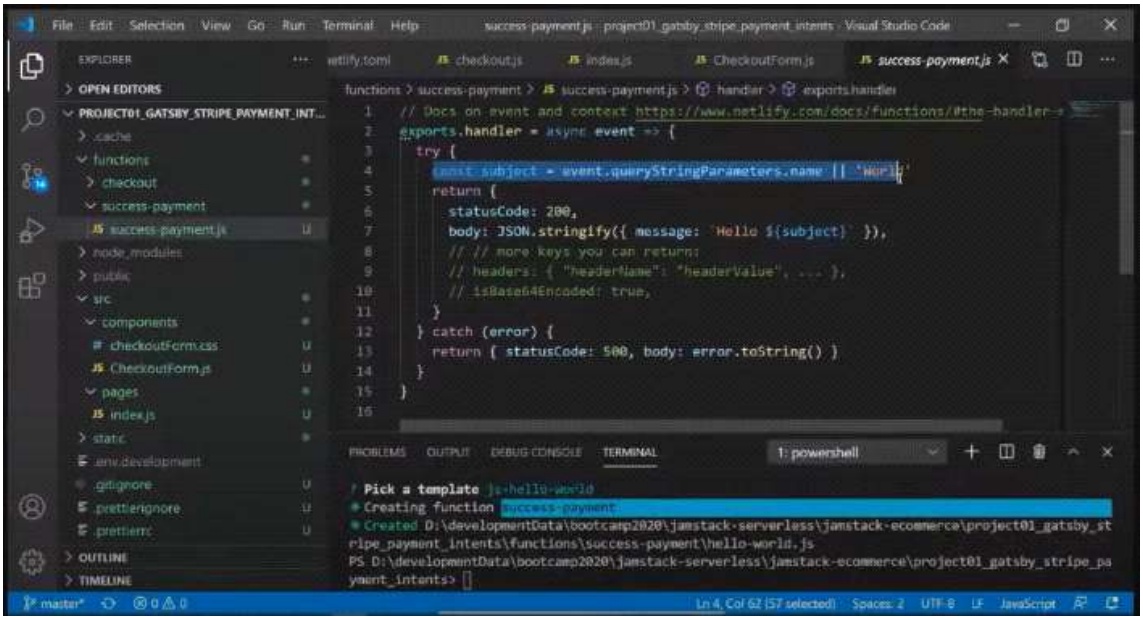
32. Open checkoutform.js and add card expiry element and card cvc element.



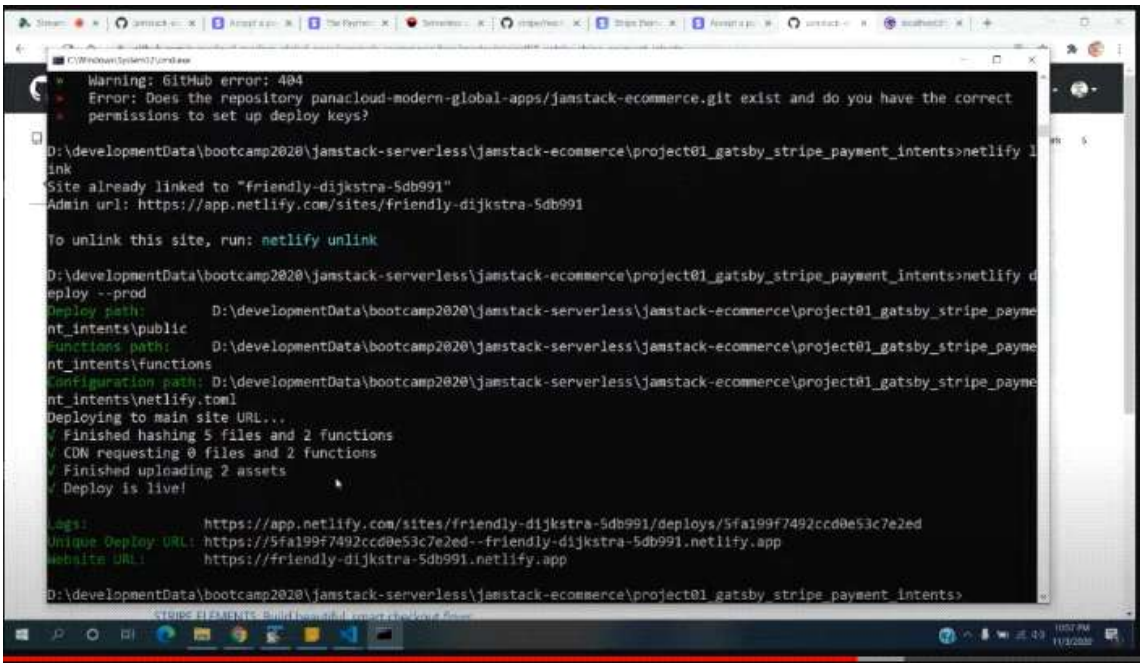
33. Create a success payment function in terminal.



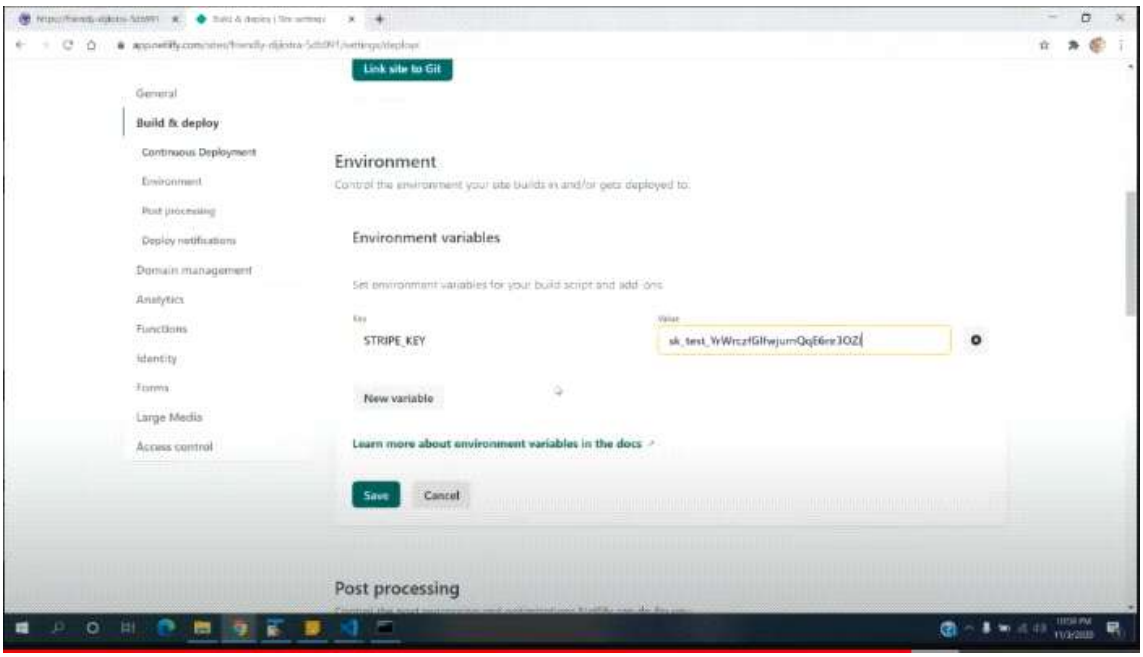
34. Open success-payment.js and forget the const subject.



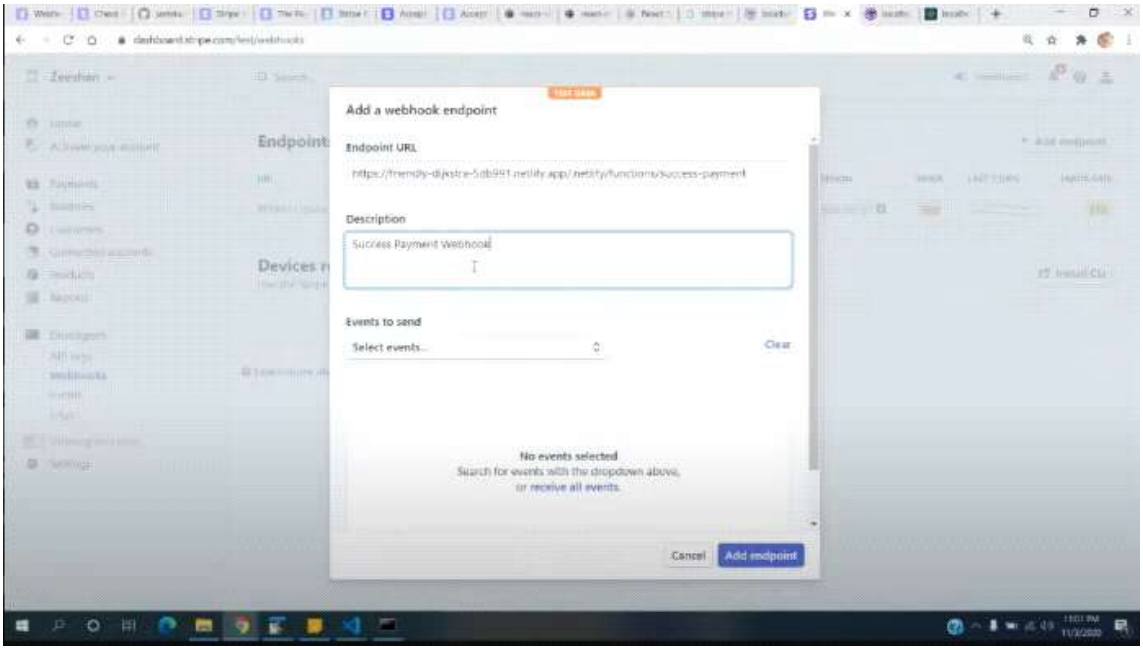
35. Deploy the application in cmd.exe



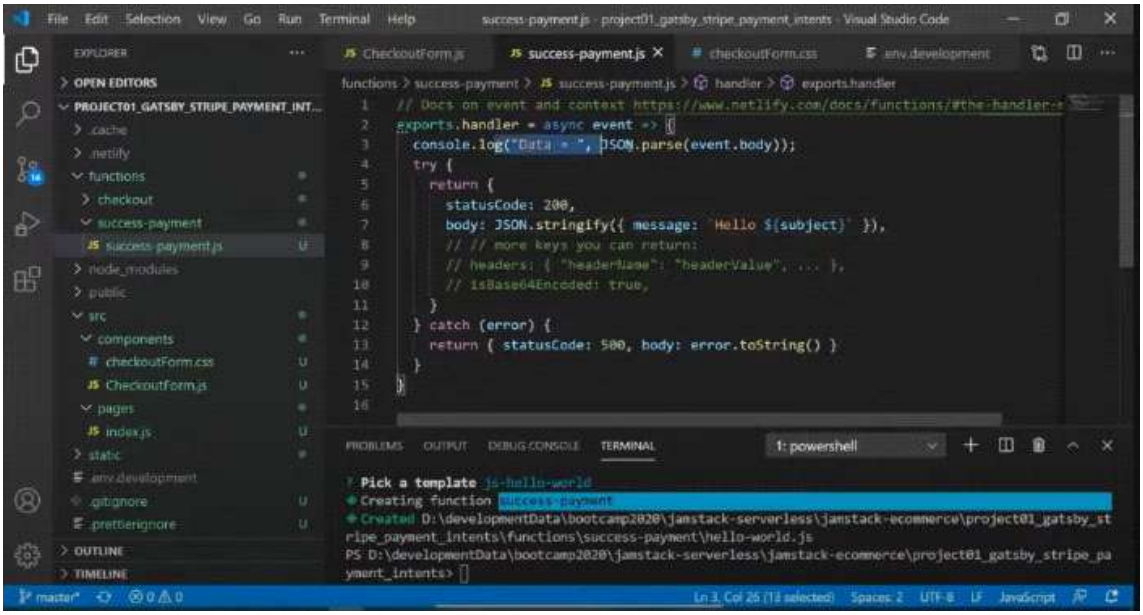
36. Open .env.developement and copy STRIPE_KEY variables and paste it on app.netlify.com to deploy.



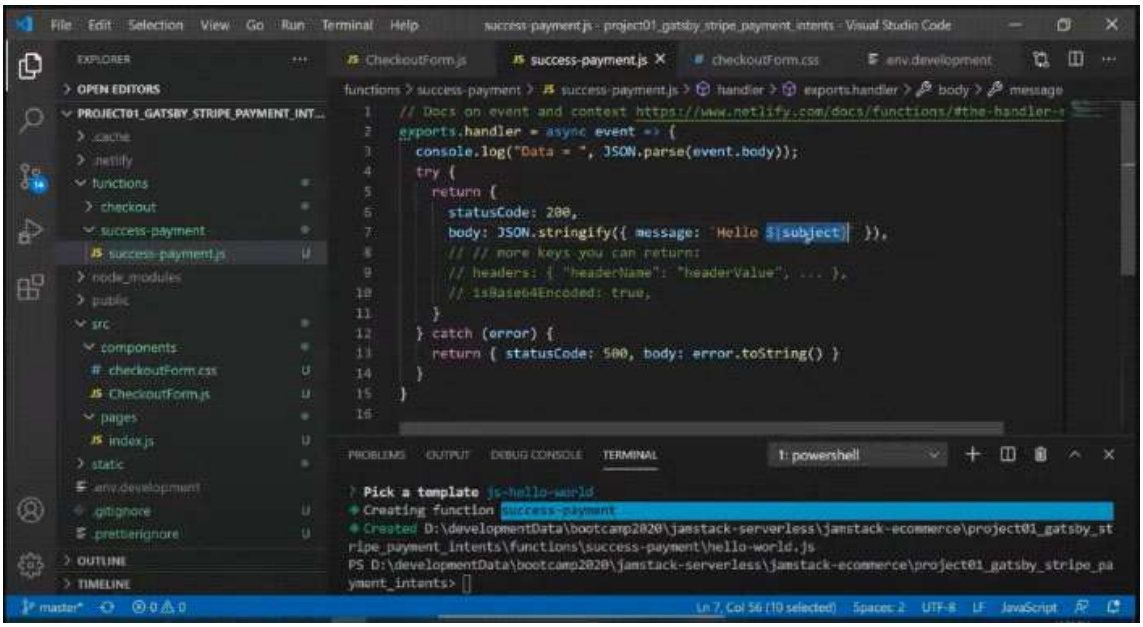
37. Add an endpoint URL and description.



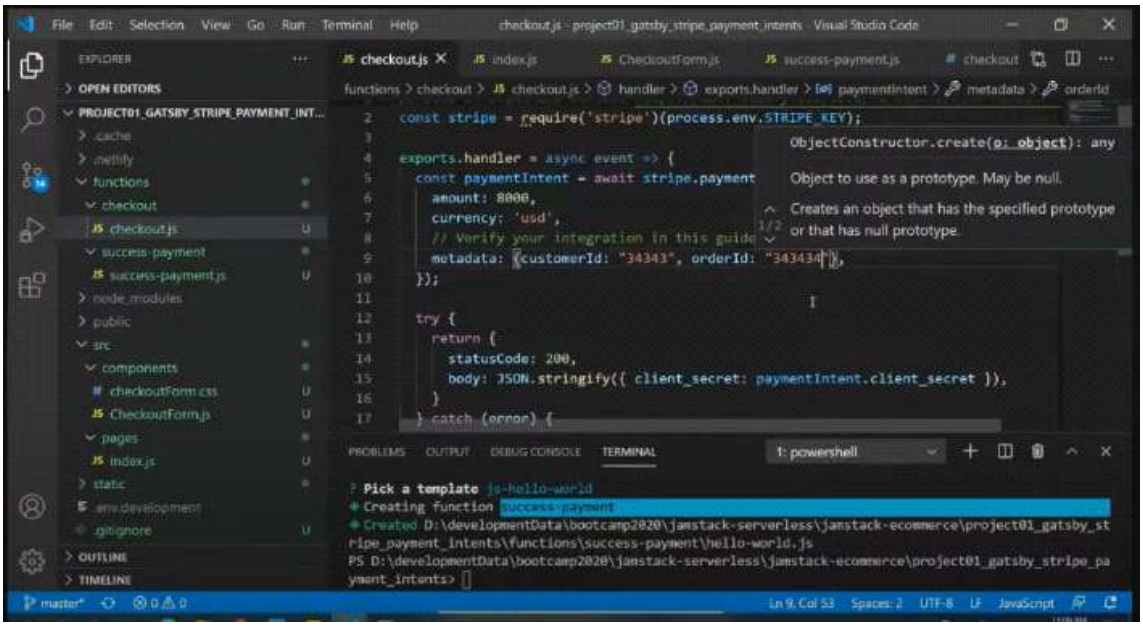
38. Add console log code in success-payment.js



39. Open success-payment.js and change subject.



40. Change the metadata to customer ID and order ID

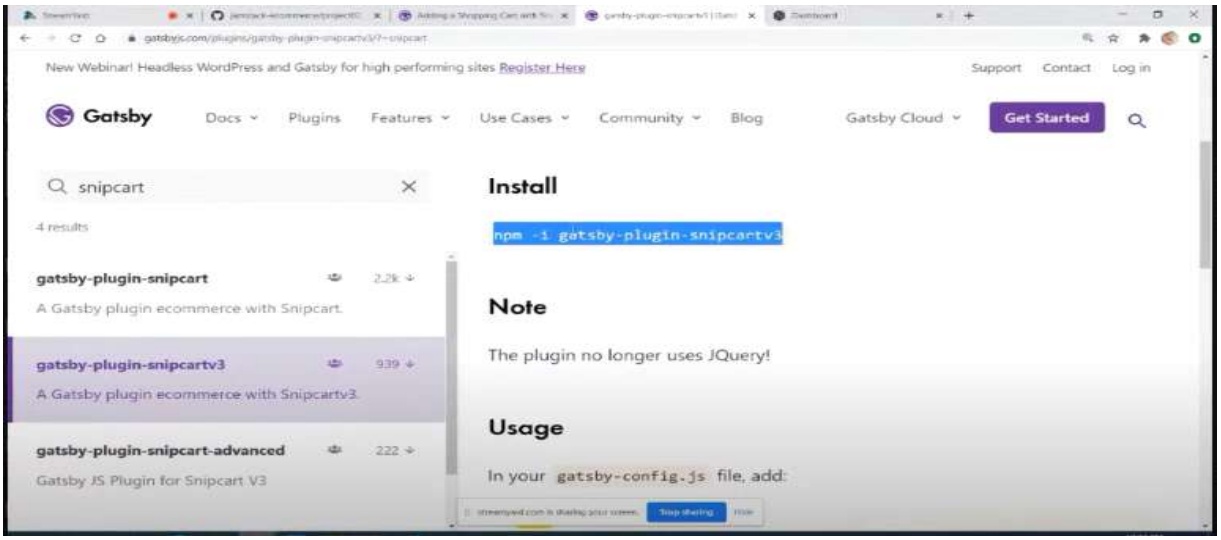


Session- 45

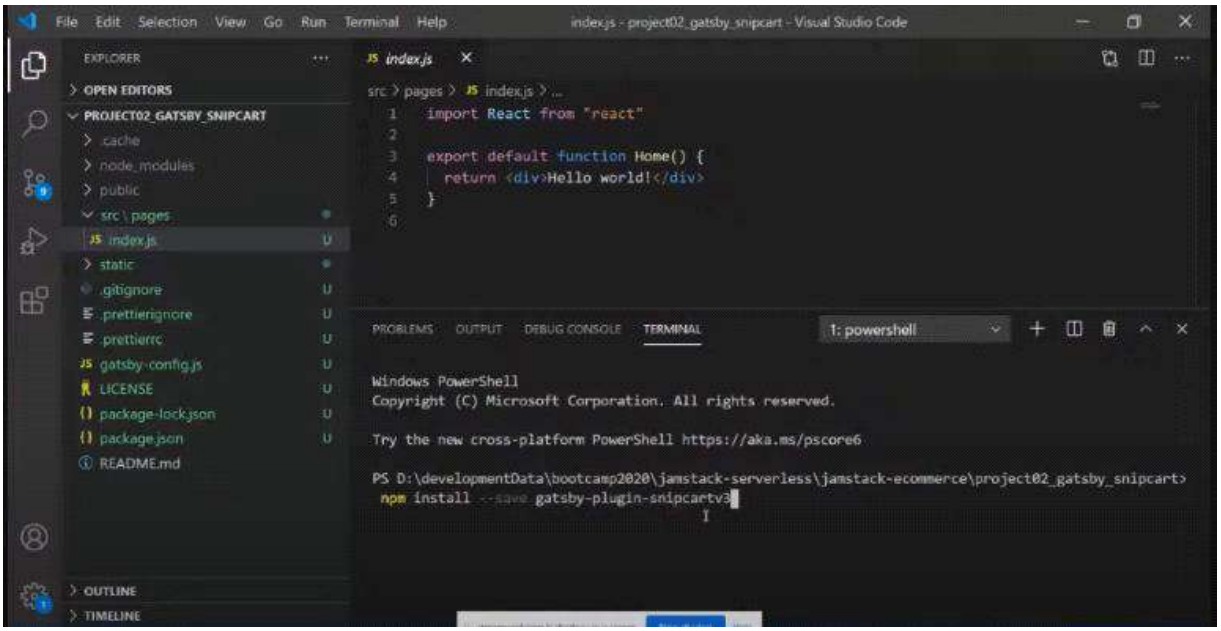
Building Shopping Cart Using Snapcart

Project - 2 (Gatsby Snipcart)

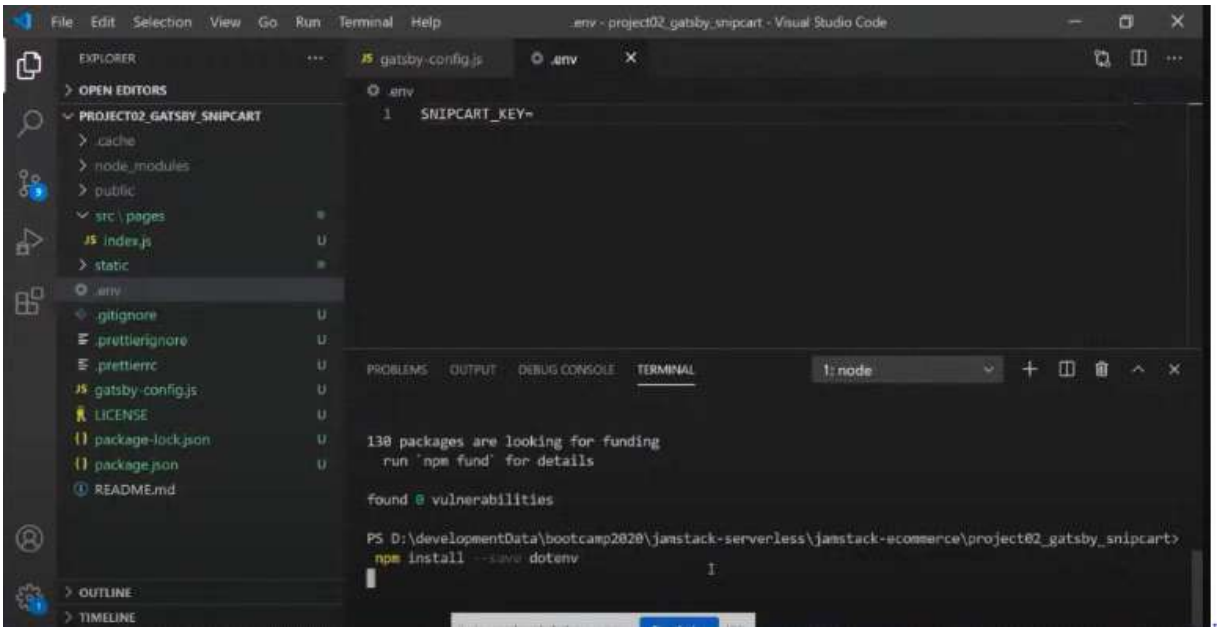
- Go to snipcart.com
- Sign up the account
- If you have account, so just login
- Go to Gatsby plugin snipcart
- Search for snipcartv3
- Install this



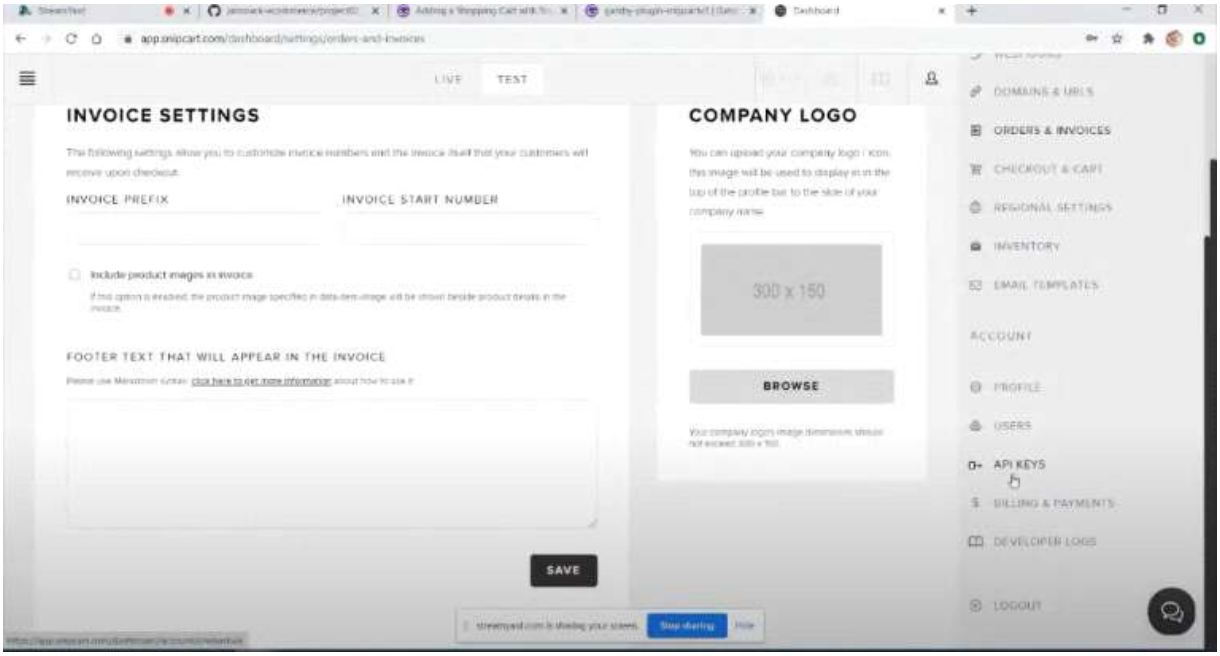
- Install plugin in terminal



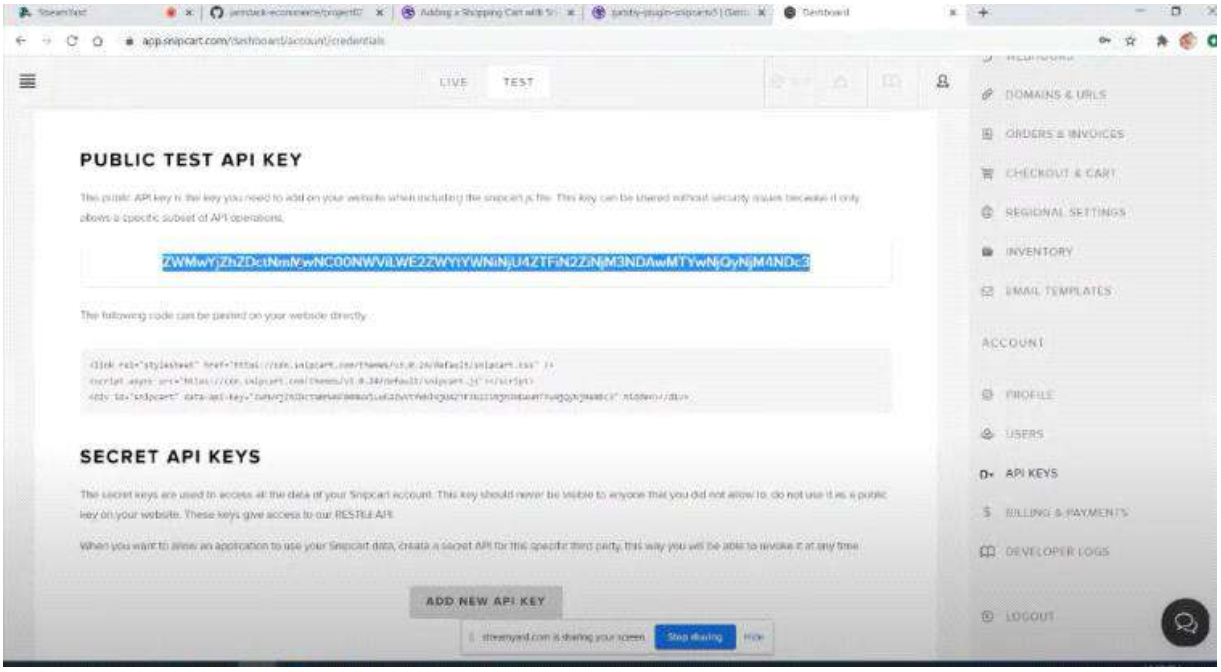
- Copy this code and add in gatsby-config.js
- Create a new folder ".env"
- Inside .env add snipcart kry
- Install dotenv in terminal



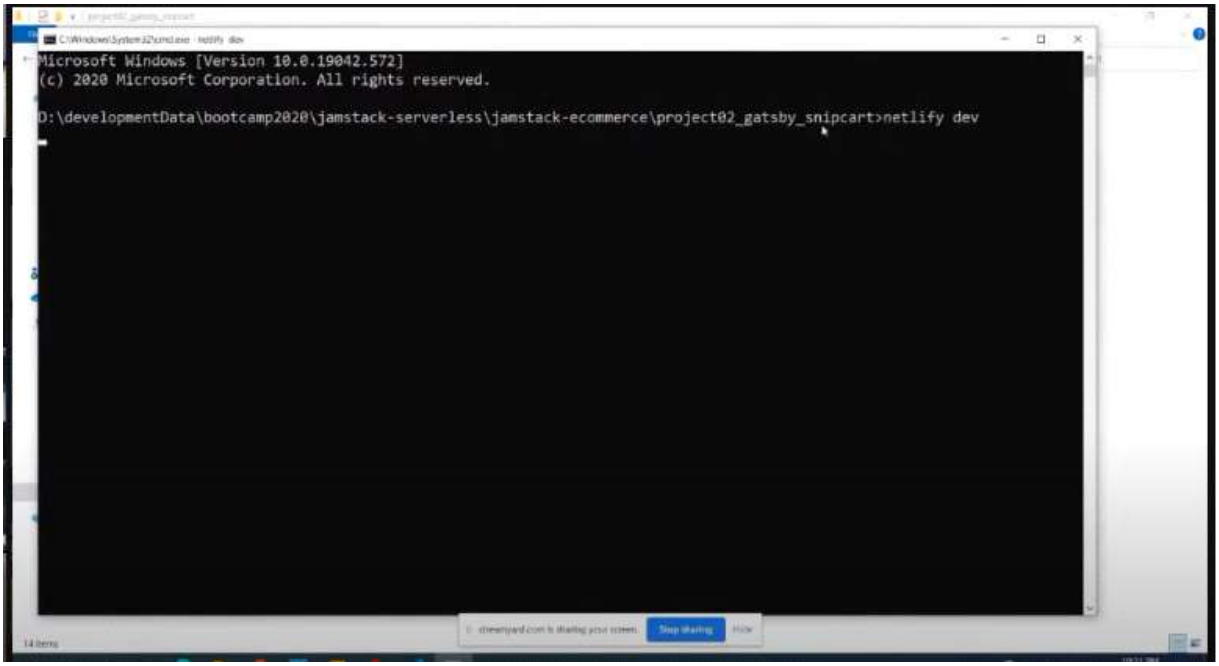
- Update the gatsby-cnfig.js code
- Go to API keys



- Copy this key



- Paste the API key in .env
- Add netlify dev in create app



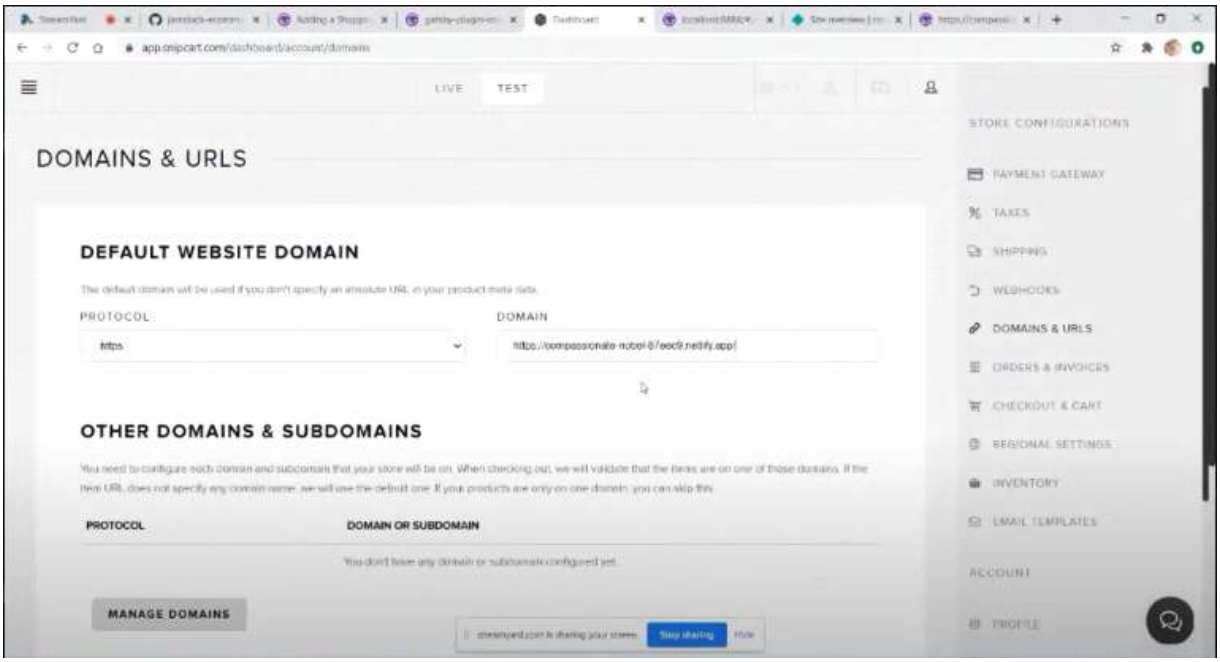
- Then go to "localhost8888" and check your results
- Then add some code in index.js for checkout button

```
button class="snipcart-add-item"
  data-item-id="formal -shoe"
  data-item-price="25"
  data-item-description="This is best shoe"
  data-item="/"
  data-item-image (image URL)
data-item-name="The starry Night"
>
    Checkout
  </button>
 </div>

)
```

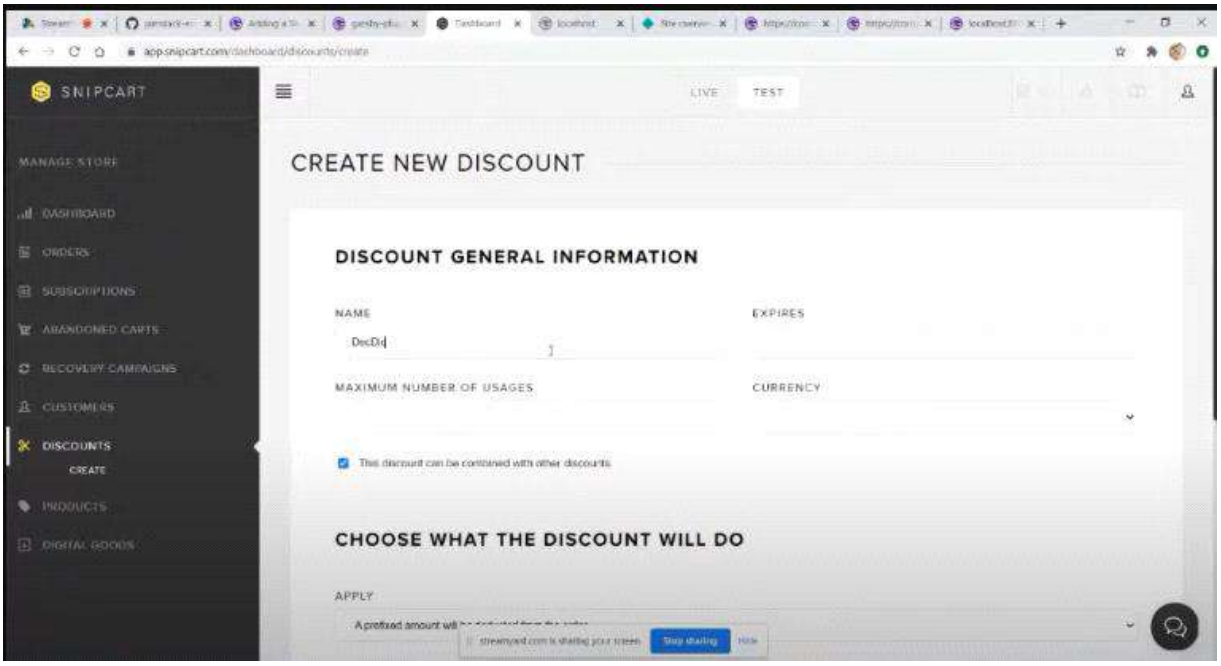
- Add a new file (netlify.toml)
- Inside this file make this code

```
[build]

   command="gatsby build"

   publish="public
```

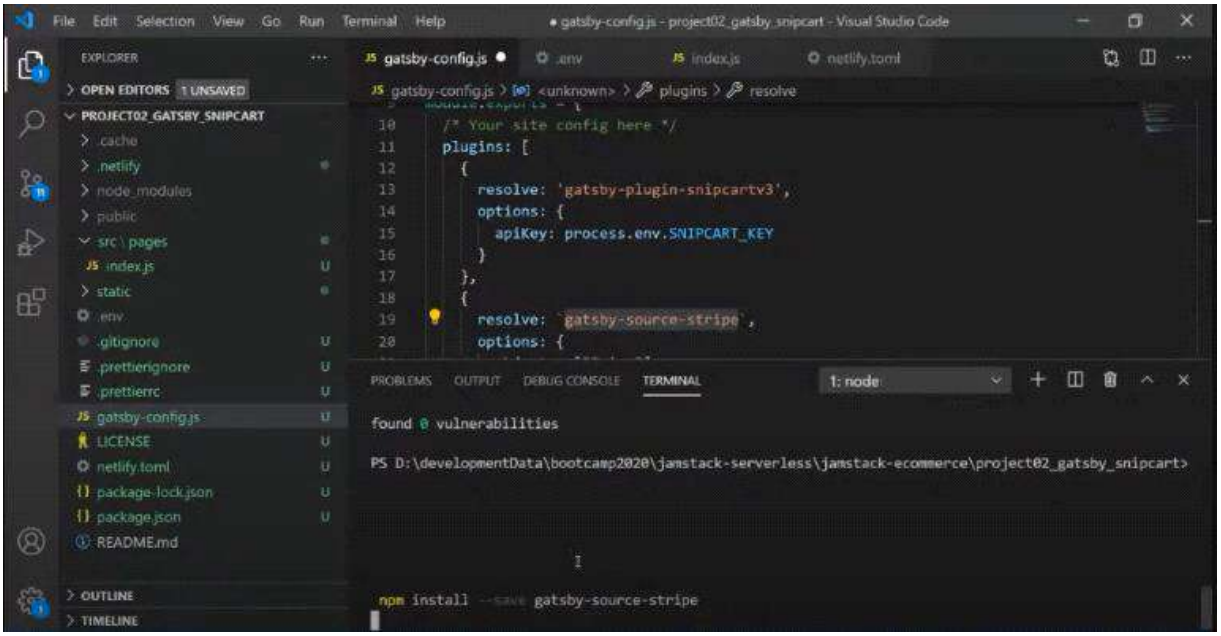
- Go to Netlify site
- Go to your application and copy the URL
- Paste it inside the domain



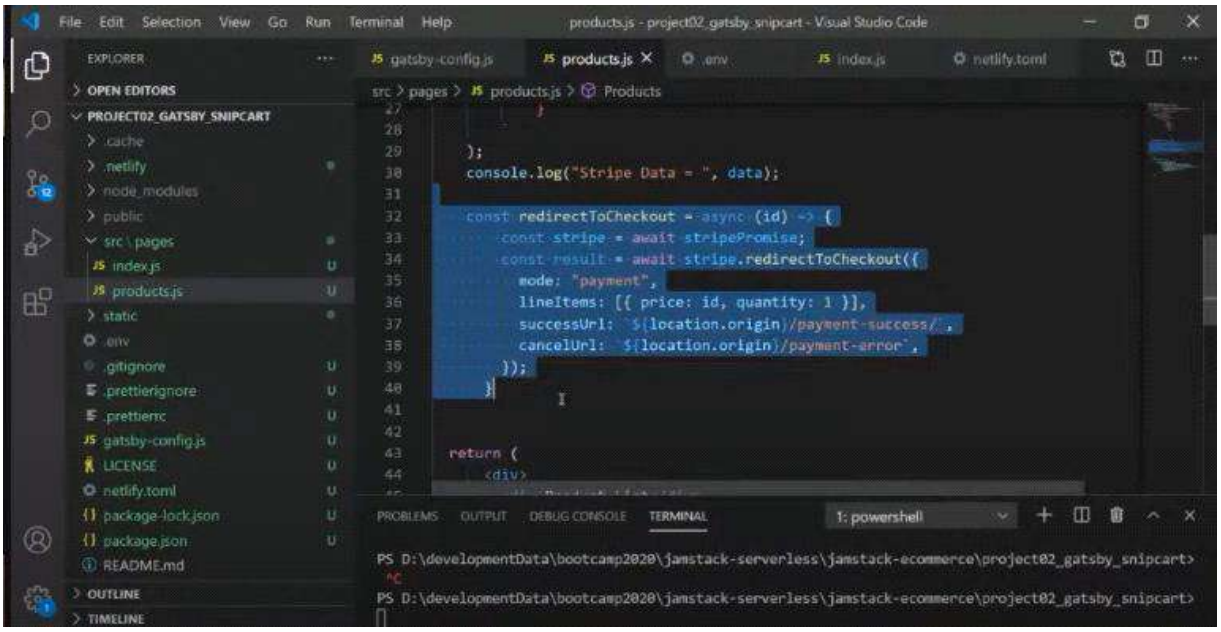
- Then save it
- Then check your result at localhost8888
- Add data item in index.js
- To add discounts
- Go to create discount



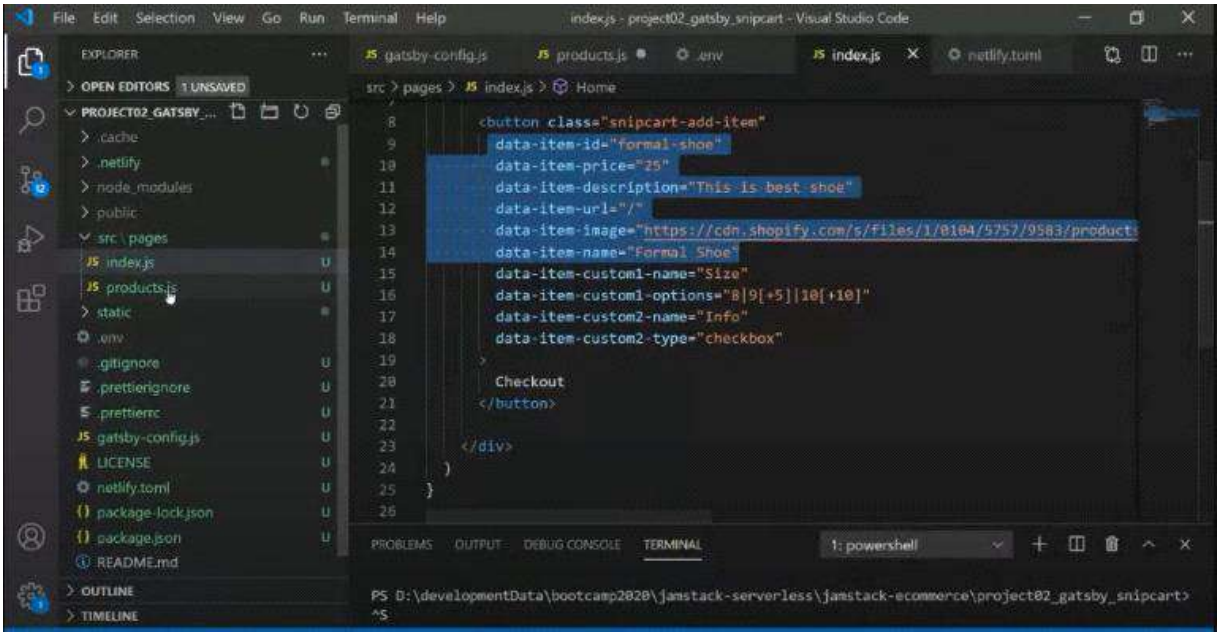
- Then go to "localhost8888" and check the discount
- Install the source stripe in gatsby-config.js



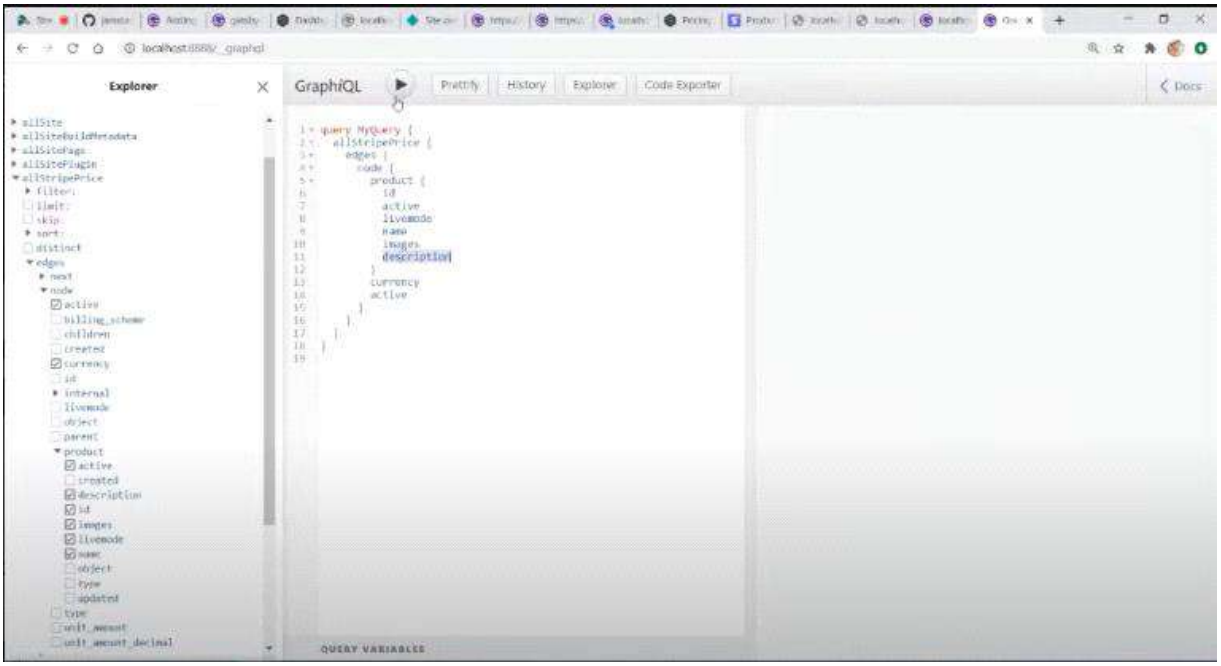
- Copy the productlist.js code that we created in previous Session
- Then create a new page “product.js
- Paste the code inside this page
- Remove redirect checkout from the code



- Install stripe.js
- Go to product.js
- Instead of checkout, use add to cart
- Copy the code from index.js

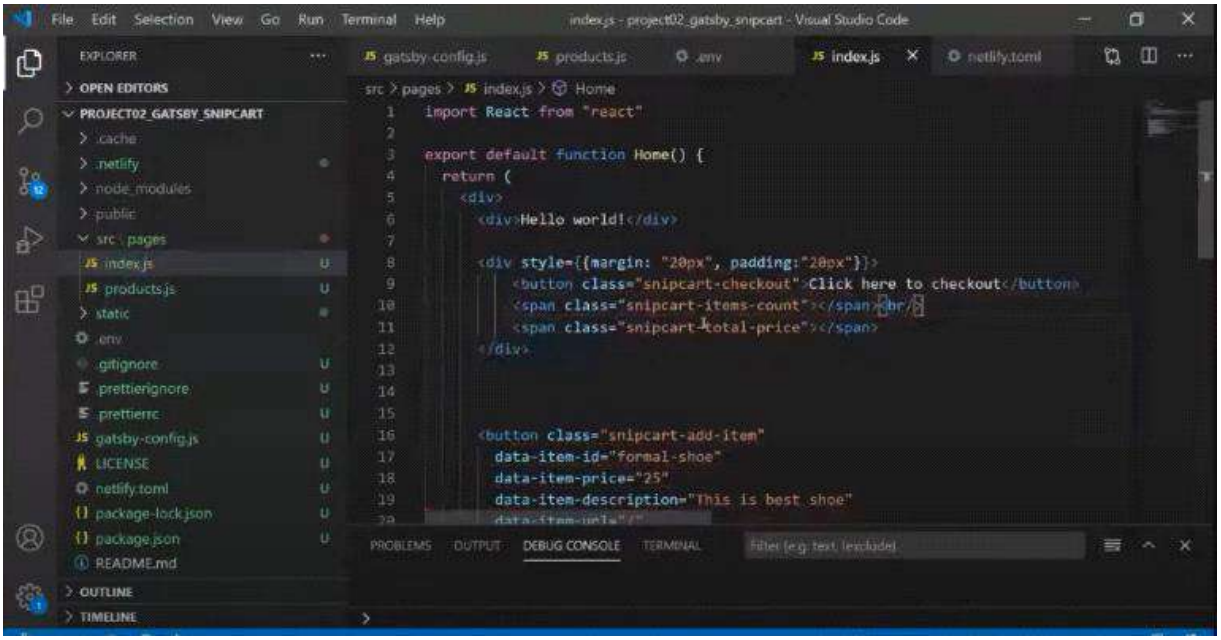


- Paste that code in product.js
- Then make some changes in code
- Then go to GraphiQL and copy the description from there

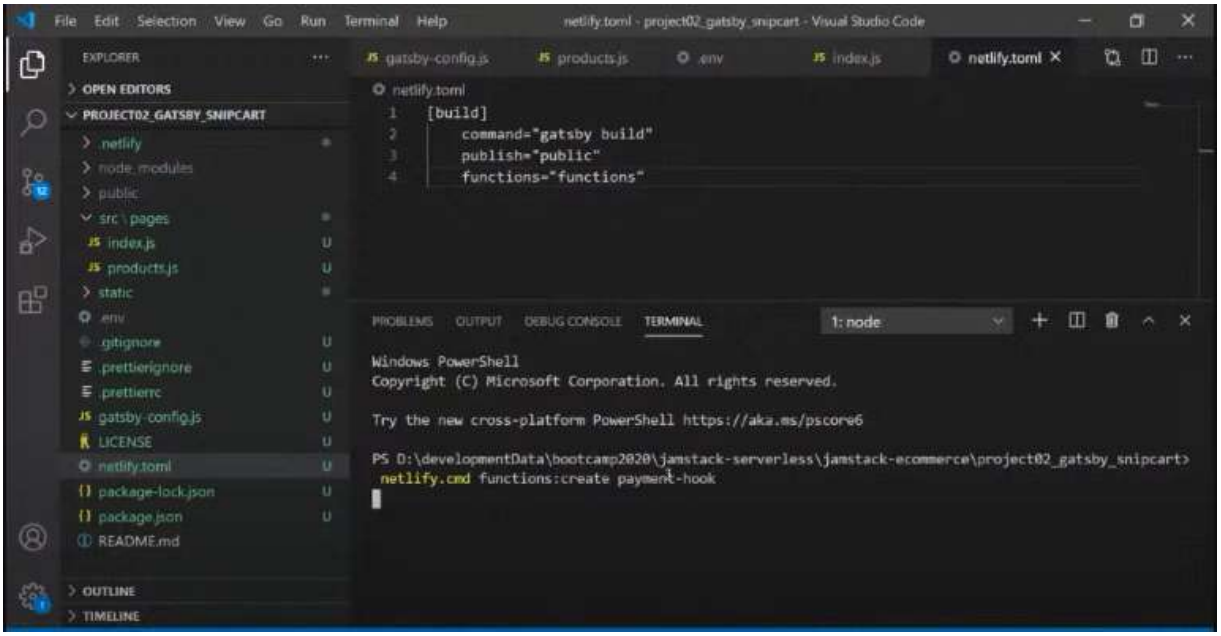


- Paste it in product.js code
- Then deploy the page

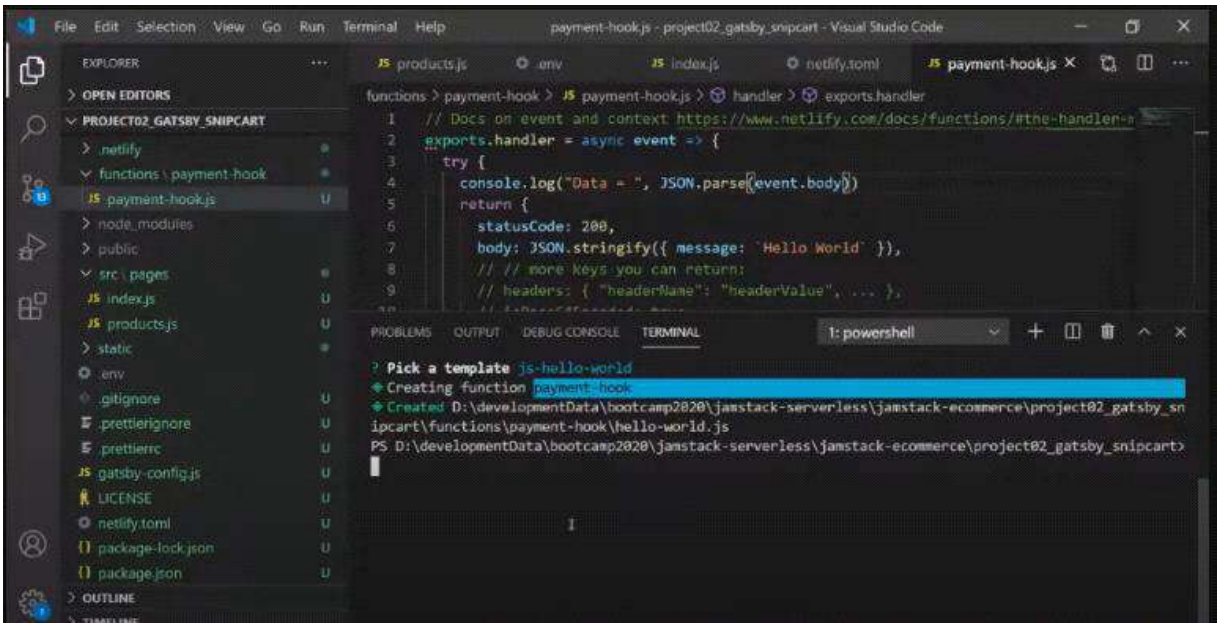
- Add snipcart checkout button in index.js



- Then check your results in "localhost8888"
- Create a function in netlify.toml
- Install create payment-hook in terminal



- You have payment-hook.js
- Inside it make some changes



- Then check your updates at "localhost8888"