

**FULL STACK DEVELOPMENT**

**CREATION OF AN UI WITH RANDOM USER API**

***Submitted by***

|  |  |
| --- | --- |
| **STUDENT NAME** | **ROLL NO** |
| KRISHNAKUMAR.R | 727621BEC060 |

VIGNESH.J.N 727621BEC076

KRISHNA.A 727622BEC302

***in partial fulfilment for the award of the degree of***

**BACHELOR OF ENGINEERING**

***in***

ELECTRONICS AND COMMUNICATION ENGINEERING

# Dr MAHALINGAM COLLEGE OF ENGINEERING AND TECHNOLOGY

**An Autonomous Institution Affiliated to ANNA UNIVERSITY CHENNAI - 600 025**

**Dr. MAHALINGAM COLLEGE OF ENGINEERING AND TECHNOLOGY, POLLACHI -642 003**

**(An Autonomous Institution Affiliated to Anna University, Chennai - 600 025)**

**BONAFIDE CERTIFICATE**

Certified that this project report titled **“ Creation of an random user API ”** is the bonafide work of

**STUDENT NAME ROLL NO**

KRISHNAKUMAR.R 21BEC060

VIGNESH.J.N 21BEC076

KRISHNA.A 22BEC302

who carried out the project under my supervision.

|  |  |
| --- | --- |
| **HEAD OF THE DEPARTMENT**  Department of Dr.Mahalingam college of y Engineering and Technology Pollachi- 642003 | **SUPERVISOR**  Department of  Dr. Mahalingam college of  Engineering and Technology Pollachi- 642003 |

Submitted for the Autonomous End Semester Mini project Examination held on

# INTERNAL EXAMINER EXTERNAL EXAMINER

**TABLE OF CONENTS**

|  |  |  |
| --- | --- | --- |
| **S.NO** | **TITLE** |  |
|  |  |  |
| 1  2  3 | OBJECTIVE  INTRODUCTION  PROGRAMMING LANGUAGES USED  3.1 JAVA SCRIPT  3.2 CSS  3.3 HTML |  |
| 4 | WHAT WE HAVE LEARNED  4.1 HOW TO CREATE REACT APP  4.2 BASIC CONCEPTS  4.2.1 GRID AND EXAMPLE  4.2.2 FLEX BOX AND EXAMPLE  4.2.3 HOOKS AND EXAMPLE  4.2.4 PROMISES |  |
| 5 | HOW CREATION IS TO BE DONE  5.1 CODES |  |
| 6 | FINAL OUTPUT |  |

7 CONCLUSION

**1.OBJECTIVE**

The main objective of the project is to create an random user API with more than two user in up scroll type which depicts the user information containing the following details of the user,

* Name
* Gender
* Date of Birth
* E-mail
* User name
* Password

**2.** **INTRODUCTION**

In this project, we will be utilizing the Random User API to create a user interface (UI) that displays random user information such as name, email, address, and profile picture. The Random User API is a free web service that generates random user data, including names, addresses, phone numbers, and more. By utilizing this API, we can easily create a UI that showcases a variety of different user profiles. Our UI will be designed to be visually appealing and easy to use, with a focus on providing an intuitive user experience. To create our UI, we will be using a combination of HTML, CSS, and JavaScript. Our project aims to showcase the functionality and capabilities of the Random User API while also demonstrating our skills in UI design and development. We will be utilizing web technologies such as HTML, CSS, and JavaScript to create a sleek and modern UI that is both user-friendly and visually appealing.

**3. PROGRAMMING LANGUAGES USED**

**3.1 Java Script**

JavaScript is used to create a dynamic and interactive user interface that allows users to browse through a large collection of randomly generated user profiles from the Random User API and to communicate with the Random User and handle user interactions such as clicks and scrolls.

**3.2 CSS**

CSS (Cascading Style Sheets) is used to style and layout the content of our UI to make it visually appealing

**3.3 HTML**

HTML (Hypertext Markup Language) is to structure and organize the content of our UI and Creating links and navigation

**4.** **WHAT WE HAVE LEARNED**

**4.1 How to create react app?**

**Steps to create react app**:

Install Node.js:

React requires Node.js, so make sure you have it installed on your machine. You can download it from the official Node.js website (https://nodejs.org) and follow the installation instructions for your operating system.

Install Create React App: Create React App is a command-line tool that sets up a new React project with the necessary configurations and dependencies. Open your terminal or command prompt and run the following command to install Create React App globally:

npm install -g create-react-app

Create a new React app: Once Create React App is installed, navigate to the directory where you want to create your React app using the terminal or command prompt. Run the following command to create a new React app:

npx create-react-app my-app

Replace "my-app" with the name you want to give to your app. This command will create a new directory with the specified name and set up a basic React project structure inside it.

Navigate to the project directory: After the project is created, navigate to the project directory by running the following command:

cd my-app

Replace "my-app" with the name of your app.

Start the development server: To start the development server and see your React app in the browser, run the following command:

npm start

This command will compile the React code and start a local development server. You can access your app by opening a web browser and visiting http://localhost:3000.

Customize your React app: You can now open the project directory in a code editor of your choice and start customizing your React app. The main file you'll be working with is src/App.js, where you can define your React components and logic.

Build your app for production: When you're ready to deploy your React app, you can build an optimized and minified version using the following command:

npm run build

This command will create a build folder with the compiled and optimized files ready for deployment.

**4.2 Basic concepts:**

**4.2.1 Grid :**

CSS Grid helps in writing quick and efficient layouts using the in-browser grid capabilities. Before the arrival of CSS Grid, you need to devise your custom grid system or go with front-end frameworks such as Bootstrap. Though you can rely on frameworks, CSS Grid makes front-end development even easier while developing simple and complex layouts.

Basic Terminologies

The basic terms associated with CSS Grid are as follows:

=>Columns

=>Rows

=>Cells

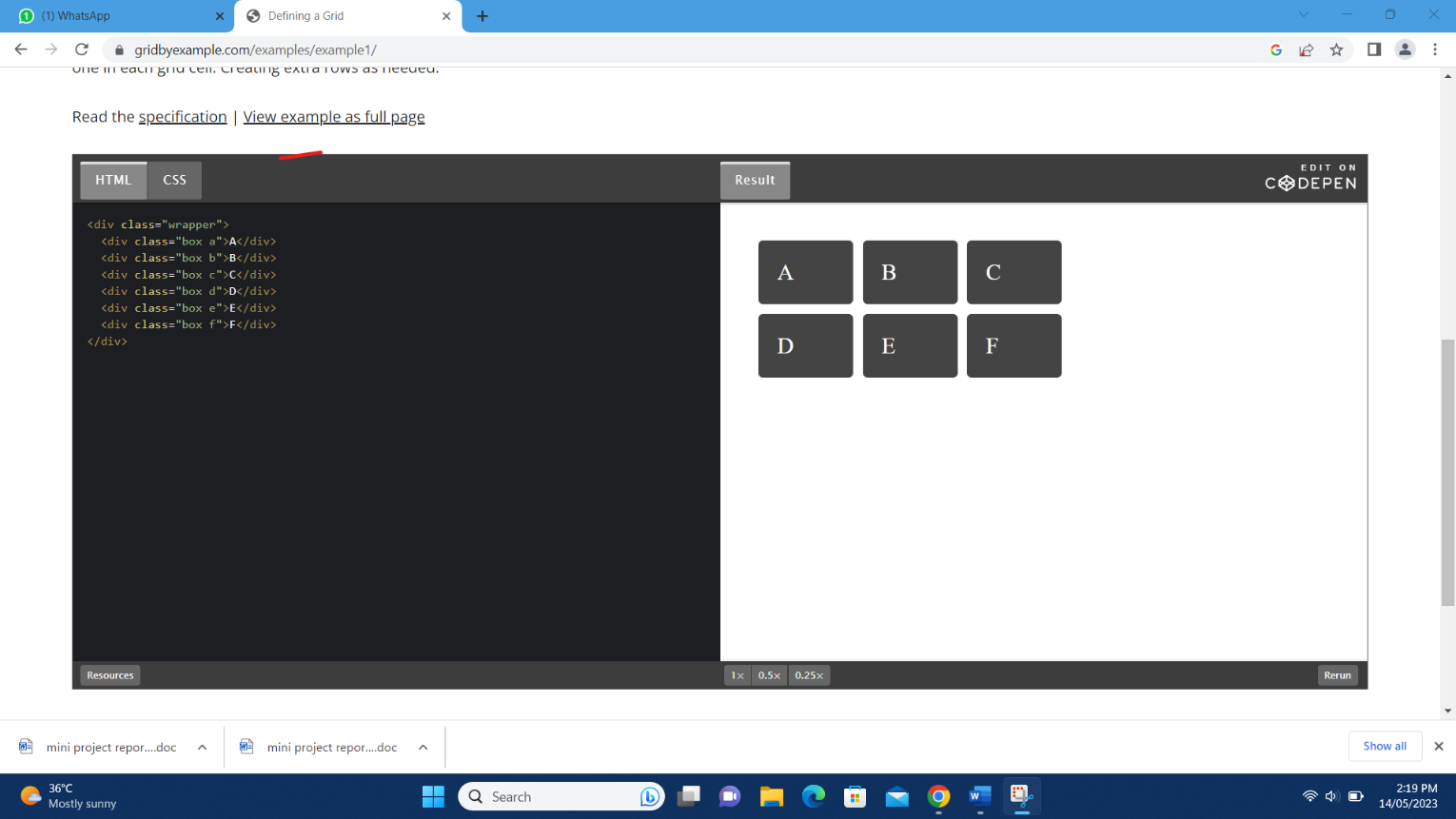
=>Grid Lines

=>Gutter

We can define columns and rows on the grid by using grid-template-rows and grid-template-columns properties.

A grid container can be created by declaring the display: grid or display: inline-grid on an element. Grid container contains the items of a grid that are placed inside the rows and columns.

**Example with output:**

****

**4.2.2 Flex Box:**

Flexbox is a powerful layout system in CSS (Cascading Style Sheets) that allows developers to create flexible and responsive designs for web pages. It provides a straightforward way to align, distribute, and reorder elements within a container.

To use flexbox, you need to define a flex container (parent element) and its child elements. Here's a step-by-step explanation of how flexbox works, along with an example:

Creating a Flex Container:

To make an element a flex container, you need to apply the CSS property display: flex; or display: inline-flex; to it. The display: flex; property makes the container a block-level element, while display: inline-flex; makes it an inline-level element.

HTML

<div class="flex-container">

<!-- Child elements go here -->

</div>

CSS

.flex-container {

display: flex;

}

Specifying Flex Direction:

By default, flex containers have a row-based layout, meaning their child elements are arranged horizontally. However, you can change this behavior by setting the flex-direction property. The available options are row, row-reverse, column, and column-reverse. For example, flex-direction: column; arranges the child elements vertically.

CSS

.flex-container {

display: flex;

flex-direction: row; /\* Default value \*/

}

Aligning Flex Items:

Flexbox offers various properties to control how flex items align within the container. Here are some commonly used alignment properties:

justify-content: Defines how flex items are distributed along the main axis. Options include flex-start, flex-end, center, space-between, and space-around.

align-items: Determines how flex items are aligned along the cross-axis. Options include flex-start, flex-end, center, baseline, and stretch.

align-self: Overrides the alignment set by align-items for an individual flex item.

CSS

.flex-container {

display: flex;

justify-content: center;

align-items: center;

}

Controlling Flex Item Sizing:

Flex items have properties that control their sizing behavior:

flex-grow: Specifies how flex items grow to occupy remaining space in the flex container. A value of 1 means the item grows to fill the space proportionally, while 0 prevents growth.

flex-shrink: Determines how flex items shrink when there's not enough space available. A value of 1 allows the item to shrink, while 0 disables shrinking.

flex-basis: Sets the initial size of the flex item before any remaining space is distributed.

flex: A shorthand property that combines flex-grow, flex-shrink, and flex-basis in one declaration.

CSS

.flex-item {

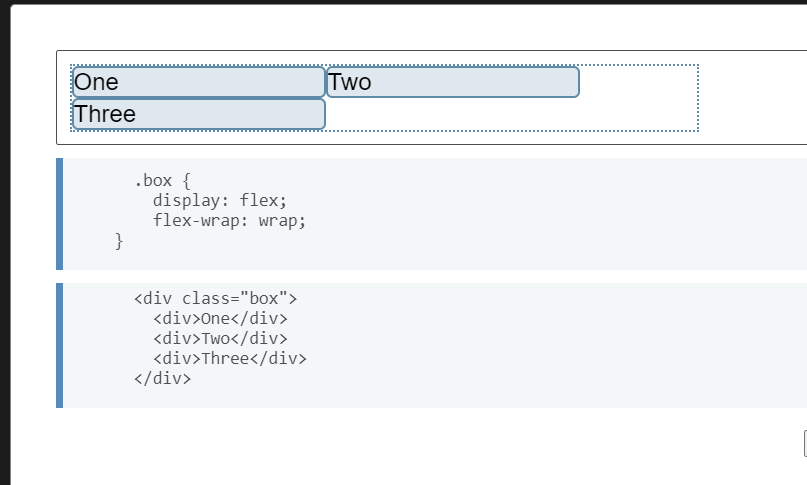
flex: 1 0 auto;

}

Reordering Flex Items:

Flex items can be reordered using the order property. By default, flex items have an order of 0, but you can assign positive or negative values to change their position.

**Example with output:**



**4.2.3 Hooks:**

"Hooks" typically refers to a feature or concept in a framework or library that allows developers to add custom logic or behavior to a component. Hooks are particularly prevalent in modern JavaScript frameworks like React and Vue.js. They provide a way to manage state, handle side effects, and enhance the functionality of components.

Hooks are used to break down the complexity of managing state and side effects in class-based components by allowing developers to use functional components with a more concise and intuitive syntax. They enable developers to reuse stateful logic across different components without the need for inheritance or higher-order components.

Let's explore two of popular hooks:

1=>useState() Hook (React):

The useState() hook allows you to add state management to functional components in React. It returns a pair of values: the current state and a function to update that state.

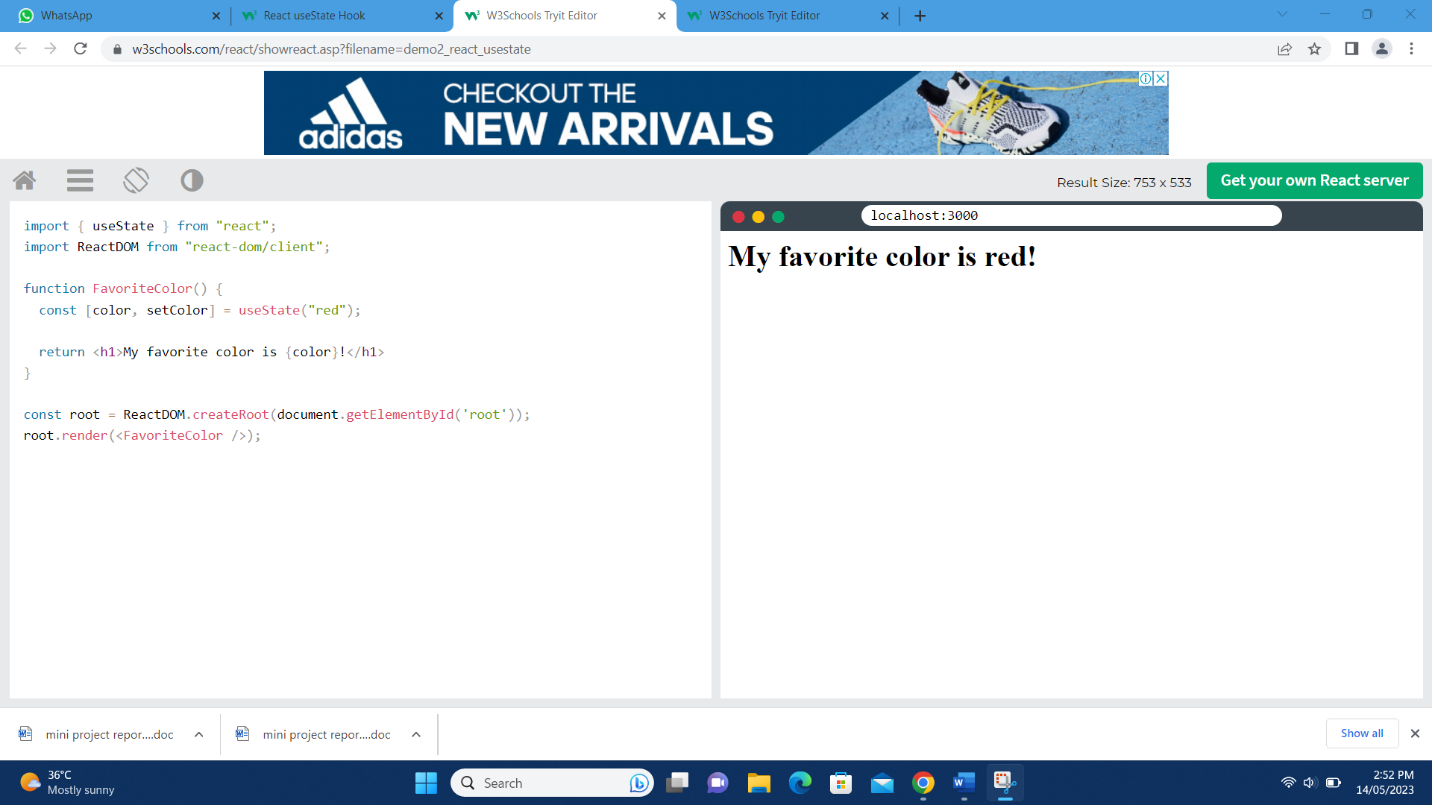
2=>useEffect() Hook (React):

The useEffect() hook is used to handle side effects in React, such as fetching

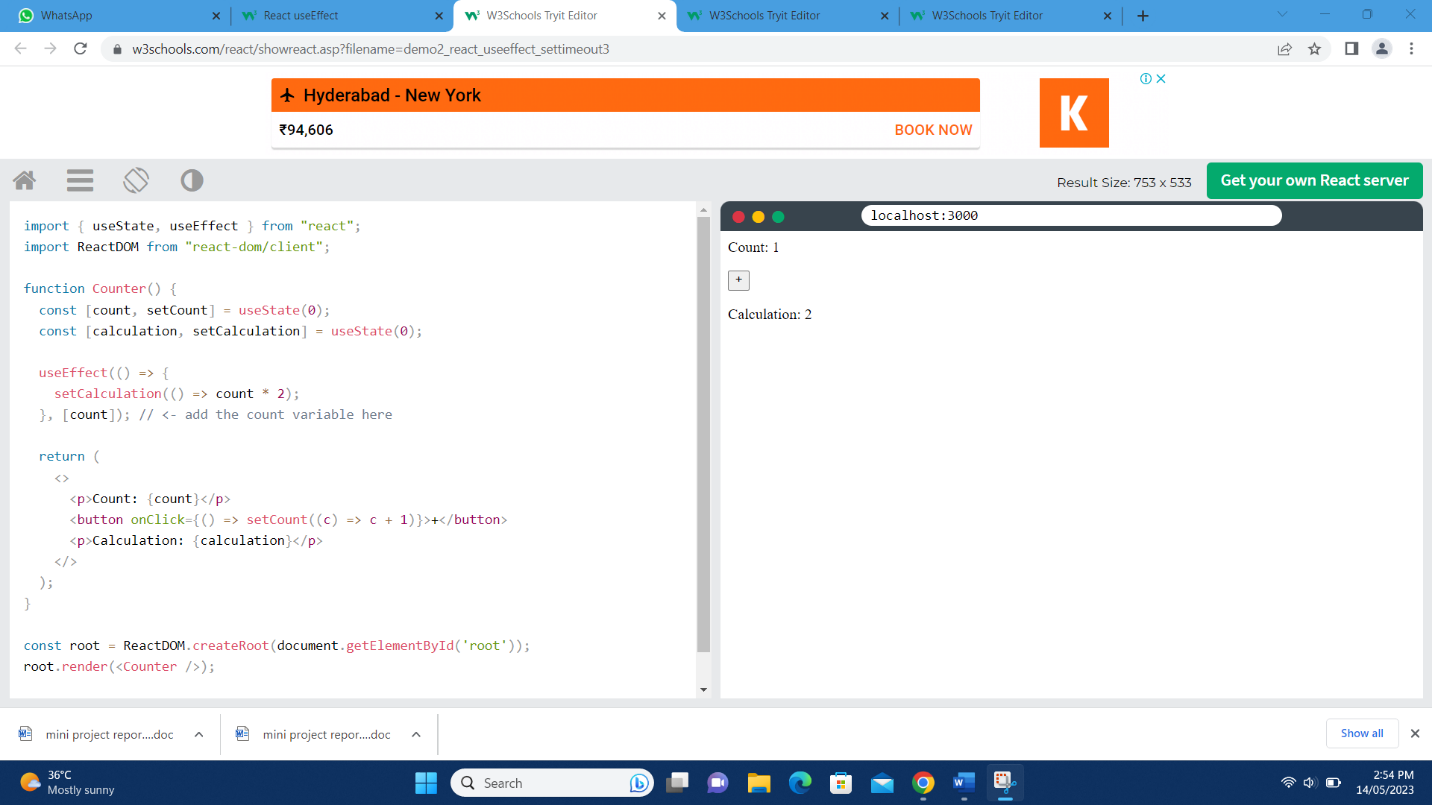
from an API, subscribing to events, or modifying the DOM. It allows you to perform these operations after the component has rendered or when certain dependencies change.

**Example and output:**

**Use state Example:**

****

**Use effect Example:**

****

**4.2.4 Promises:**

Promises are a programming construct used to handle asynchronous operations, such as making API requests, querying databases, or reading files. Promises are part of JavaScript's built-in support for asynchronous programming and are commonly used in both frontend and backend development.

A promise represents the eventual completion or failure of an asynchronous operation and allows you to write code that can handle the result of that operation without blocking the execution of other code. It provides a way to handle asynchronous operations in a more readable and manageable manner, especially when dealing with multiple asynchronous tasks.

Here's a breakdown of how promises work:

Creating a Promise:

You can create a new promise using the Promise constructor, which takes a function as an argument. This function, often referred to as an executor, is responsible for initiating the asynchronous operation and defining how the promise will be resolved or rejected.

Resolving a Promise:

When the asynchronous operation is successfully completed, you can call the resolve function provided in the executor function, passing the result as an argument. This marks the promise as fulfilled.

Rejecting a Promise:

If an error occurs during the asynchronous operation, you can call the reject function provided in the executor function, passing an error object as an argument. This marks the promise as rejected.

Handling Promise Results:

To handle the result of a promise, you can chain .then() and .catch() methods to it. The .then() method is called when the promise is successfully resolved, and the .catch() method is called when the promise is rejected.

Ex:

myPromise

.then((result) => {

console.log(result);

})

.catch((error) =>

});

The .then() method can also return another promise, allowing you to chain multiple asynchronous operations together.

Handling Multiple Promises:

When dealing with multiple promises, you can use Promise.all() to wait for all promises to resolve or reject. The resulting promise is fulfilled with an array of the resolved values from each promise, in the same order as the input promises.

**Example and output:**

****

**5 .HOW CREATION IS TO BE DONE**

**5.1 Codes:**

**Index.html:**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document</title>

<link rel="stylesheet" href="index.css">

</head>

<body>

<div id="outer">

<div id="head">USER DETAILS</div>

<div id="users"></div>

<button id="new\_user">NEW USER</button>

</div>

**<**script src="index.js" ></script>

</body>

</html>

**Index.CSS:**

body{

margin: 0px;

background-color:#ffbbb4;

}

#outer{

border-style: solid;

border-width: 7px;

border-radius: 20px;

border-color: #ffbbb4;

padding: 20px;

background-color: #f35588;

height: 100%;

}

#users{

width: 100%;

}

#users,.singleUserDiv{

display: flex;

flex-direction: column;

justify-content: center;

align-items: center;

margin-top: 10px;

margin-bottom: 15px;

}

.singleUserDiv{

padding: 20px;

margin: 20px;

width: 90%;

border-radius: 20px;

background-color: #ffbbb4;

display: flex;

justify-content: left;

align-items: start;

padding-left: 40px;

}

#head,#new\_user{

background-color: #007944;

color: #ffbbb4;

border-radius: 25px;

font-weight: 900;

font-family: fantasy;

font-size: 30px;

width: 200px;

height: 40px;

display: block;

margin-right: auto;

margin-left: auto;

}

#head{

display: flex;

align-items: center;

justify-content: center;

}

.photodiv{

align-self: center;

border-color: #71A95A;

border-width: 10px;

border-style: solid;

padding: 4px;

background-color: black;

}

.info{

font-size: 25px;

color: #007944;

margin: 8px;

}

.photo{

transition-duration: 1s;

width: 250px;

}

#new\_user{

transition-duration: 0.5s;

}

#new\_user:hover{

width: 300px;

height: 60px;

background-color: white;

color: black;

}

.photo:hover{

width: 400px;

}

.info:hover{

color: white;

}

@media (min-width: 850px){

.singleUserDiv{

display: flex;

flex-direction: row;

}

.photo{

width: 350px;

}

#textInfo{

display: flex;

width: 100%;

align-items: center;

flex-direction: column;

}

}

**Index.js:**

console.log("start")

let users\_details=[]

// let details\_name=['uname','gender','dob','age','email','phono','uid','user','pass']

class datacls{

constructor(name,photo,gender,dob,age,email,phono,uid,user,pass){

this.photo=photo

this.uname=name

this.gender=gender

this.dob=dob

this.age=age

this.email=email

this.phono=phono

this.id=uid

this.username=user

this.password=pass

}

}

let out=document.getElementById('outer')

let usersDiv=document.getElementById('users')

let but = document.getElementById('new\_user')

but.addEventListener('click',function(){

getdata()

})

let getdata=()=>{

fetch('https://randomuser.me/api/')

.then(object => object.json())

.then(items => {

let details\_name=[]

uname=('NAME : '+items.results[0].name.title+'.'+items.results[0].name.first+' '+items.results[0].name.last)

details\_name.push(uname)

photo=items.results[0].picture.large

gender=('GENDER : '+items.results[0].gender)

details\_name.push(gender)

dob=('DOB : '+items.results[0].dob.date)

details\_name.push(dob)

age=('AGE : '+items.results[0].dob.age)

details\_name.push(age)

email=('EMAIL : '+items.results[0].email)

details\_name.push(email)

phono=('PHONE NO : '+items.results[0].phone)

details\_name.push(phono)

uid=('USER ID : '+items.results[0].id.value)

details\_name.push(uid)

user=('USERNAME : '+items.results[0].login.username)

details\_name.push(user)

pass=('PASWORD : '+items.results[0].login.password)

details\_name.push(pass)

let singleUserDiv = document.createElement('div')

singleUserDiv.className='singleUserDiv'

usersDiv.appendChild(singleUserDiv)

let imgdiv=document.createElement('img')

imgdiv.className='photo'

imgdiv.src=photo

let photodiv=document.createElement('div')

photodiv.className='photodiv'

photodiv.appendChild(imgdiv)

singleUserDiv.appendChild(photodiv)

let textInfoDiv=document.createElement('div')

textInfoDiv.id='textInfo'

for(let i=0; i<details\_name.length; i++){

let infodiv=document.createElement('div')

infodiv.className='info'

infodiv.innerHTML=details\_name[i]

textInfoDiv.appendChild(infodiv)

}

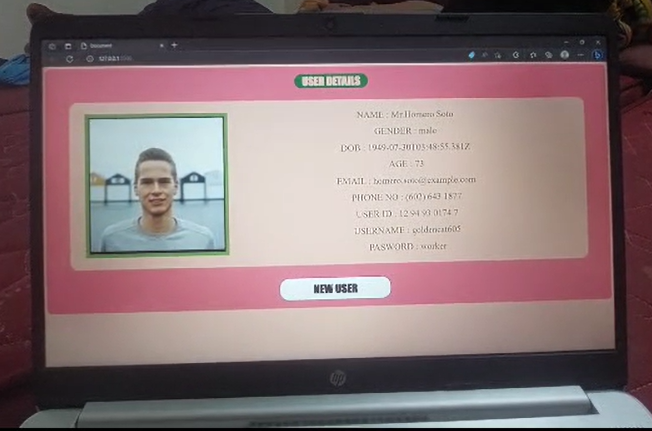
singleUserDiv.appendChild(textInfoDiv)

})

}

console.log('end')

**6.OUTPUT OF THE PROJECT:**

****

**7.CONCLUSION**

We conclude that we have created an random user with more than two users and it was implemented with concepts we have learned.

Step 1: When we run it.It shows the one user detail

Step 2: At the end of the page it shows new user navigation button if we click it, next user detail will be shown.

In this way it can be achieved.