React Workshop

By Hackerspace MMU

Installation of Node

Windows:

1. Run `npm -v` in CMD to see if it is installed.

If it isn't:

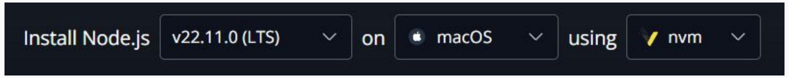
2. Navigate to this website
3. Download nvm-setup.exe in the latest one.
4. Run nvm-setup.exe and install
5. Run `nvm -v` in CMD, proceed if there is version
6. Run `nvm install latest` in CMD
7. Run `nvm use <version no.>` in CMD
8. Check by running `npm -v`

Mac:

1. Run `npm -v` in Terminal to see if it is installed.

If it isn't:

2. Navigate to this website
3. Select MacOS and using NVM to install.


Install Node.js v22.11.0 (LTS) on macOS using nvm

4. Follow the guide shown.

Installation of IDE

IDEs that you can install (Google Search or click on slide in PPT mode):

Visual Studio Code (recommended)
Developed by Microsoft, this is technically the most customisable yet easiest to learn IDE on the market right now. It is free to use.

JetBrains WebStorm
Developed by JetBrains, this is the most advanced IDE that will guide you on web development. You will need a student account for this.

Vim / NeoVim
Less popular options due to its difficulty of operation. However if you manage to master it, it will be better to use than normal options.

Installation of Git

Windows:

Before Proceeding:

Open CMD and type in `git` to see if there's any respond. If there isn't, proceed.

1. Navigate to this [website]
2. Download the latest version of Git
3. Perform the installation, no need to select "Open Git Bash" afterwards.
4. Check if installation is good by typing in `git` in a restarted CMD prompt.

Mac:

Before Proceeding:

Open Terminal and type in `git` to see if there's any respond. If there isn't, proceed.

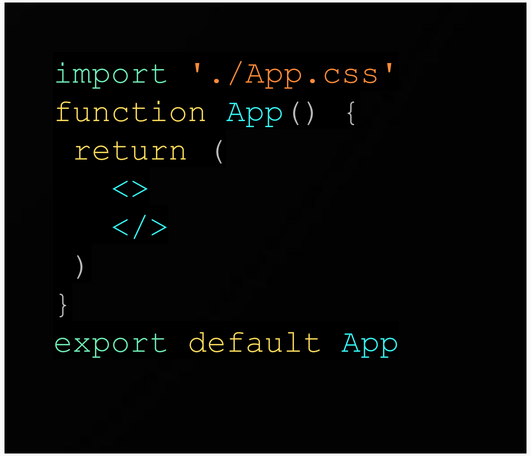
1. Navigate to this [Git repo] and download homebrew.
2. After installing homebrew, open Terminal to type in `brew install git`
3. Restart Terminal and type in `git` to see if the installation is successful.

Setting up Vite

1. Open where you want the code to be, copy the path of the folder, then open CMD/Terminal and run command `cd /d <Path Here>`
2. Do the following.
   - Run the command `npm create vite@latest timer-app`
   - Select React from the list of frameworks.
   - Select JavaScript from the list of variants
   - Run the command in order, `cd timer-app`, `npm install`, `npm run dev`
3. Open your browser and type `localhost:5173` and see something cool
4. To start coding, type `code` into the CMD/Terminal, and VSC should open!

Setting up the files

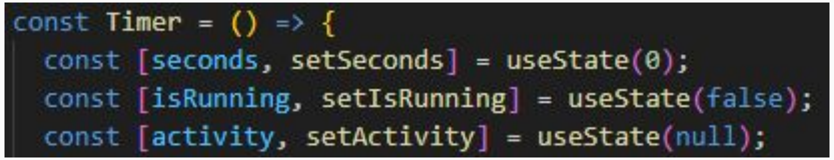
- Remove everything in App.css and replace it with the contents provided in the speaker's note.
- Delete some of the code in the App.jsx file. It should look like the code below. (Tip: you can just copy paste it if you're lazy)
- Make a new folder named components in the /src folder.



```
import './App.css'
function App() {
 return (
    <>
    </>
 )
}
export default App
```

Making the timer component

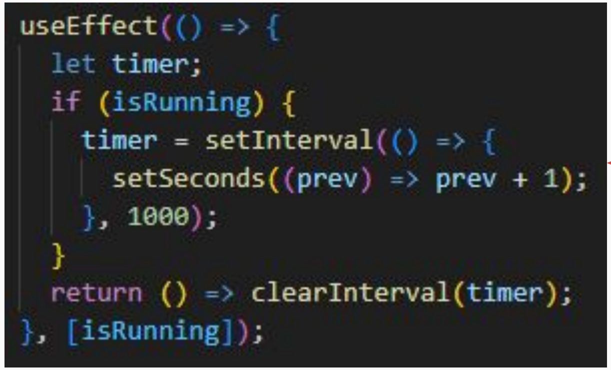
Make a new file called Timer.jsx in the /component folder and do the following:



```
const Timer = () => {
  const [seconds, setSeconds] = useState(0);
  const [isRunning, setIsRunning] = useState(false);
  const [activity, setActivity] = useState(null);
```

The name of our component.

States used to keep track of the time, whether the timer is running or not and what the current activity is.



```
useEffect(() => {
  let timer;
  if (isRunning) {
    timer = setInterval(() => {
      setSeconds((prev) => prev + 1);
    }, 1000);
  }
  return () => clearInterval(timer);
}, [isRunning]);
```

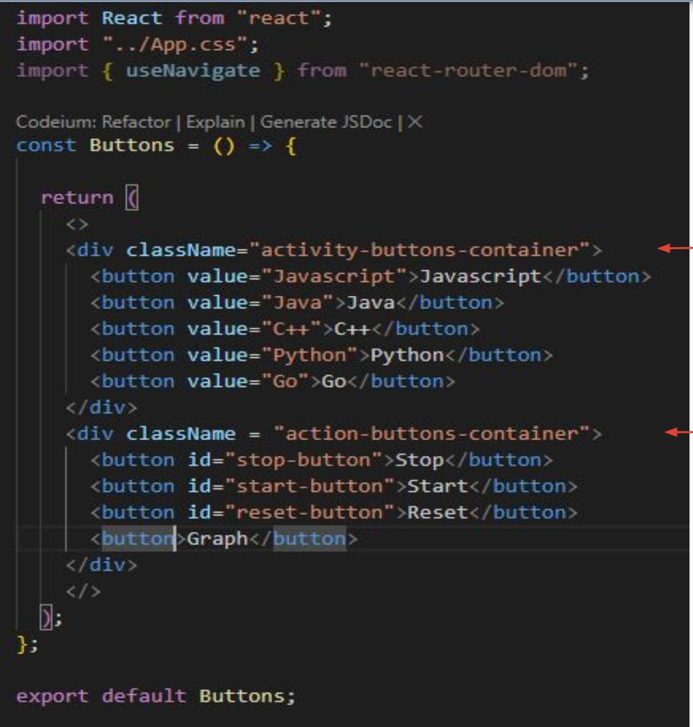
Dependency array

The logic for the timer.

If **isRunning** is set to true, set interval will run the function inside it at the specified interval in milliseconds(1000 milliseconds is 1 second). It will set our latest seconds state to increase by 1 every second.

The useEffect hook allows us to perform side effects.
In this case the side effect is the starting of the timer whenever the **isRunning** is toggled between true and false

Making the buttons component



```
import React from "react";
import "../App.css";
import { useNavigate } from "react-router-dom";

Codeium: Refactor | Explain | Generate JSDoc | X
const Buttons = () => {

  return (
    <>
    <div className="activity-buttons-container">
      <button value="Javascript">Javascript</button>
      <button value="Java">Java</button>
      <button value="C++">C++</button>
      <button value="Python">Python</button>
      <button value="Go">Go</button>
    </div>
    <div className = "action-buttons-container">
      <button id="stop-button">Stop</button>
      <button id="start-button">Start</button>
      <button id="reset-button">Reset</button>
      <button>Graph</button>
    </div>
    </>
  );
};

export default Buttons;
```

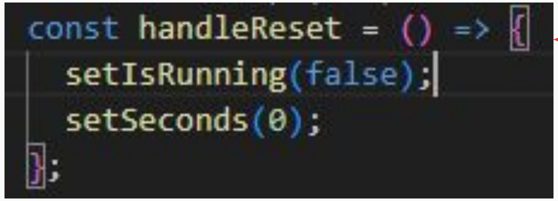
Make a new file called **Buttons.jsx** in **/components** folder

Container button for our Activities. Used to change activities.

Container button for our action buttons.

Logic for handling reset timer

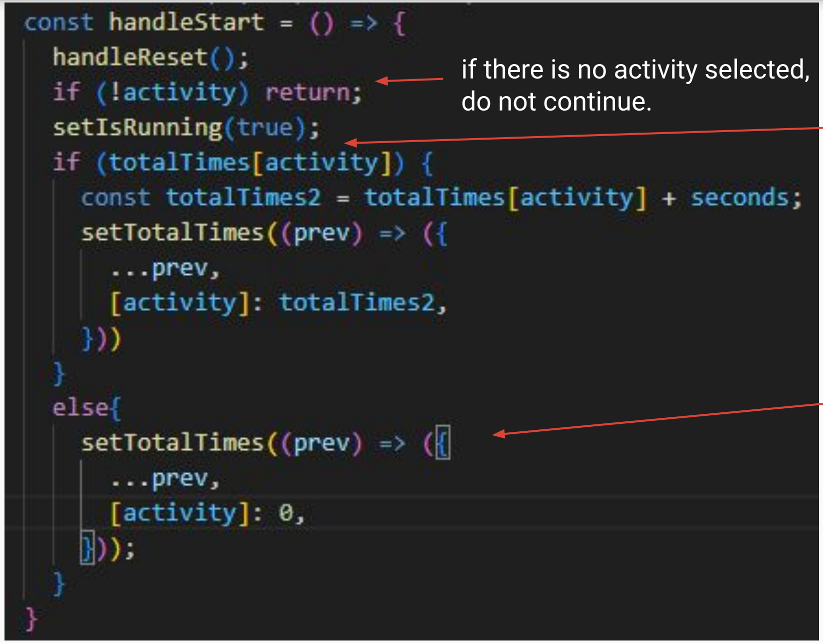
Go back to our **Timer.jsx** component. We will be creating the logics for the buttons



```
const handleReset = () => {
  setIsRunning(false);
  setSeconds(0);
};
```

Set **isRunning** to **false** using the setIsRunning() function and reset seconds back to 0 using the setSeconds() function.

Logic for starting the timer



```
const handleStart = () => {
  handleReset();
  if (!activity) return;
  setIsRunning(true);
  if (totalTimes[activity]) {
    const totalTimes2 = totalTimes[activity] + seconds;
    setTotalTimes((prev) => ({
      ...prev,
      [activity]: totalTimes2,
    }))
  }
  else{
    setTotalTimes((prev) => ({
      ...prev,
      [activity]: 0,
    }));
  }
}
```

if there is no activity selected, do not continue.

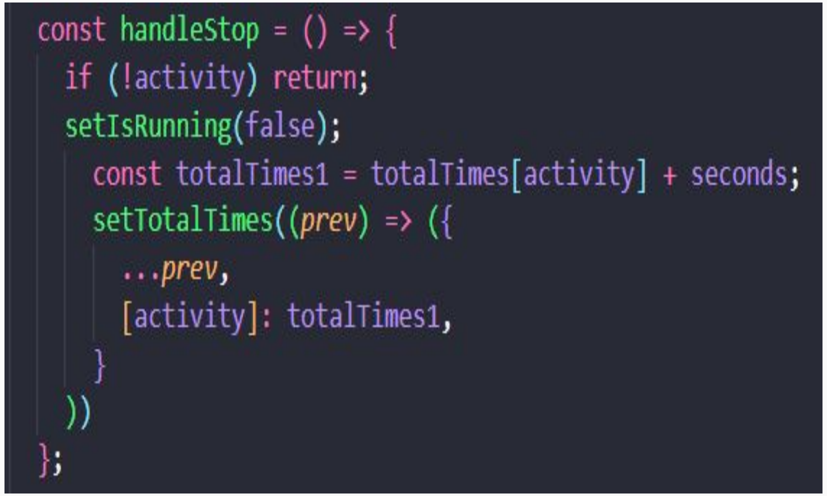
Set **isRunning** state to true.

if there is already a same activity recorded in **totalTimes**. Then set the time of the activity to the recorded time + the time on the timer.

Otherwise, add the current activity to **totalTimes** with a value of 0.

What ...prev does here is it "spreads" the properties of the prev object into the new object, effectively copying all existing key-value pairs. After spreading prev, you can add or overwrite specific properties (e.g., [activity]: totalTimes1 in this case.

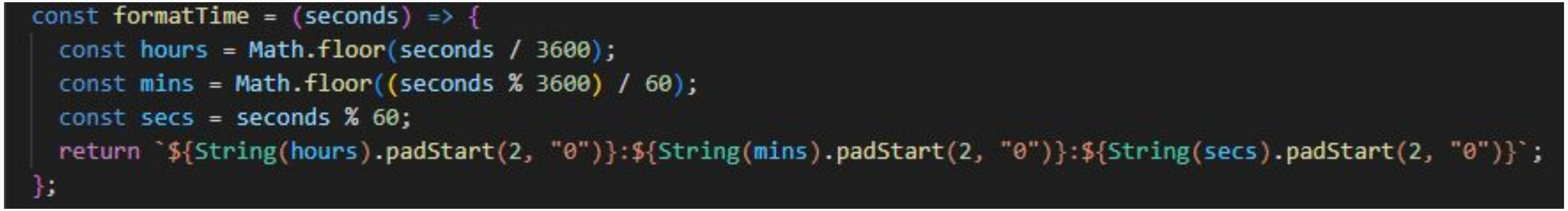
Logic for stopping the timer



```
const handleStop = () => {
  if (!activity) return;
  setIsRunning(false);
    const totalTimes1 = totalTimes[activity] + seconds;
    setTotalTimes((prev) => ({
      ...prev,
      [activity]: totalTimes1,
    }
  ))
};
```

Almost the same as starting the timer, but we **setIsRunning** to false instead. We can also remove the else condition as it will be handled by start timer.

Logic for formatting time



```
const formatTime = (seconds) => {
  const hours = Math.floor(seconds / 3600);
  const mins = Math.floor((seconds % 3600) / 60);
  const secs = seconds % 60;
  return `${String(hours).padStart(2, "0")}:${String(mins).padStart(2, "0")}:${String(secs).padStart(2, "0")}`;
};
```

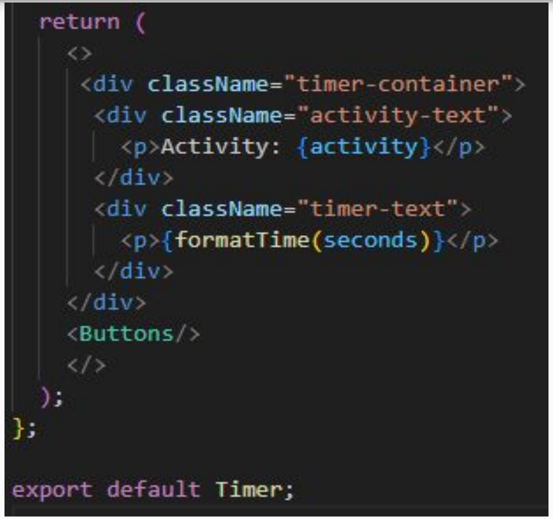
This is for format the time from raw seconds to a readable format.

First we convert the seconds to hours by dividing seconds with 3600. Floor is for rounding down to the nearest integer as we do not want decimals.

For minutes we take the remainder of seconds and 3600 and divide by 60. And for seconds we take the remainder of seconds and 60. (it's just math)

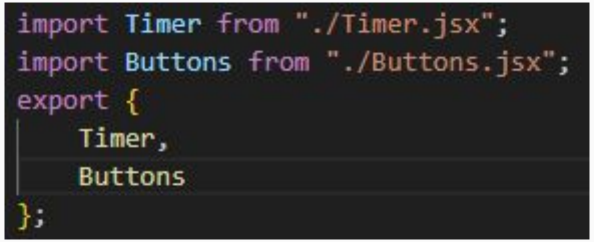
Padstart here means that it will show a 0 before the time, or 00 is there is no time on the timer if it is less than 2 digits.

Displaying it on the browser



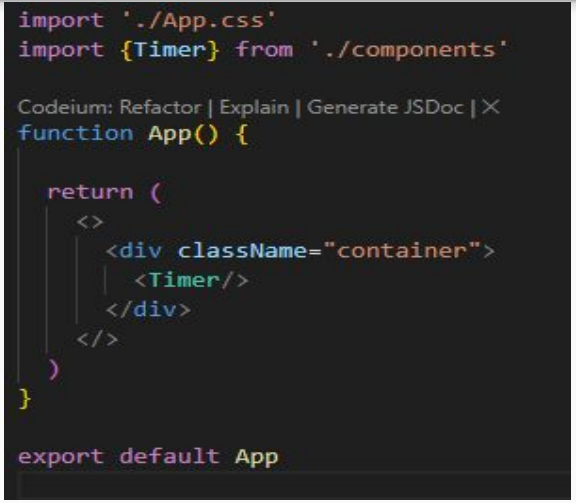
```
  return (
    <>
     <div className="timer-container">
      <div className="activity-text">
        <p>Activity: {activity}</p>
      </div>
      <div className="timer-text">
        <p>{formatTime(seconds)}</p>
      </div>
    </div>
    <Buttons/>
    </>
  );
};

export default Timer;
```

1. Return html to display the UI. and export it as Timer.



```
import Timer from "./Timer.jsx";
import Buttons from "./Buttons.jsx";
export {
    Timer,
    Buttons
};
```

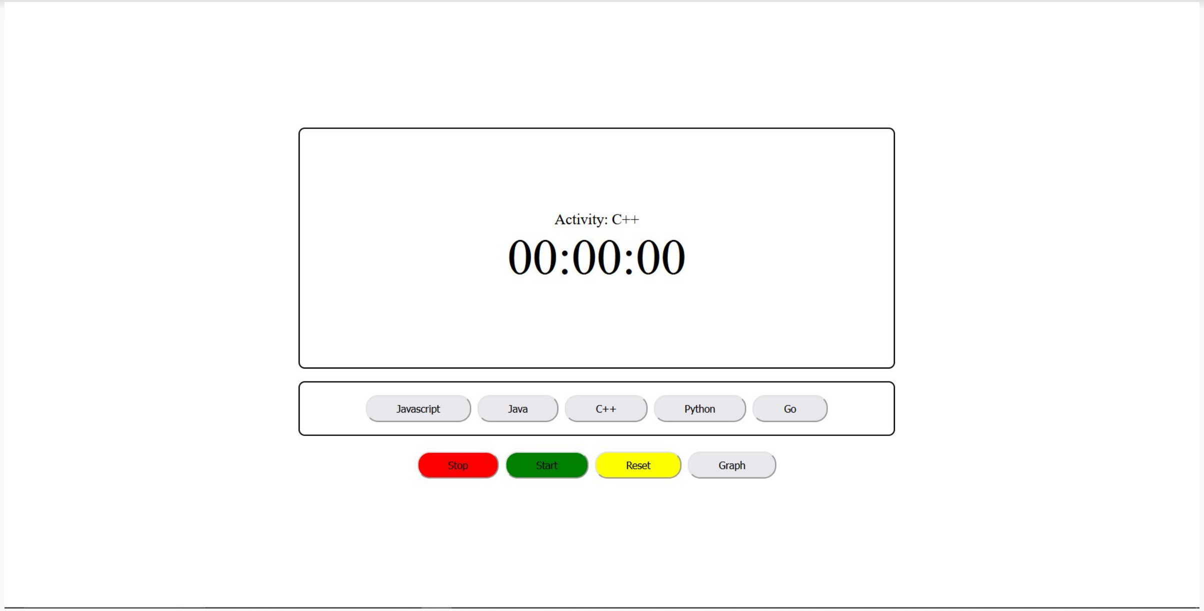
2. Make a new file called **index.js** in **/components** folder. This will act as our “hub file or barrel file” for all our imports to make importing components easier.



```
import './App.css'
import {Timer} from './components'

Codeium: Refactor | Explain | Generate JSDoc | ×
function App() {

  return (
    <>
      <div className="container">
        <Timer/>
      </div>
    </>
  )
}

export default App
```

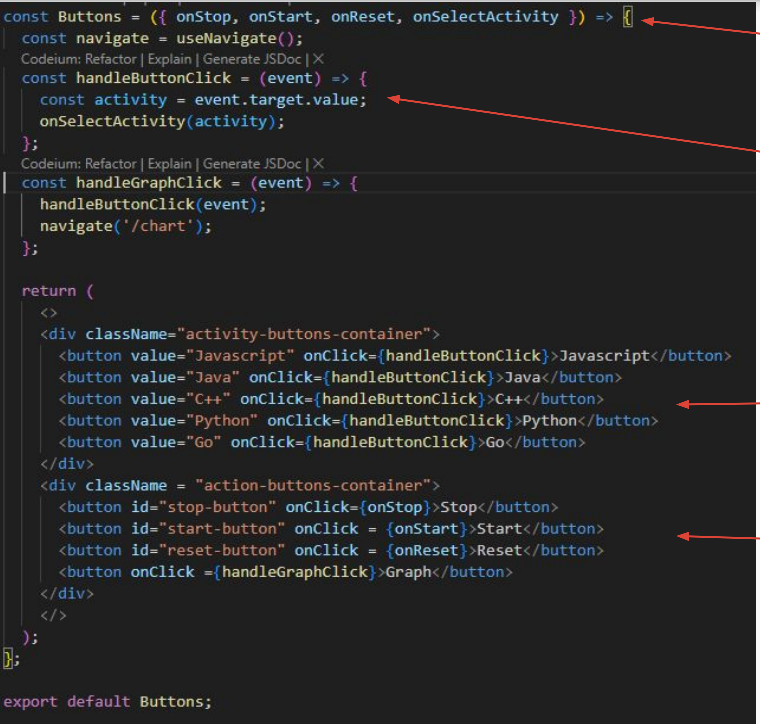
3. Go back to **App.jsx** file and import the Timer component and insert into our App component.

Displaying in on the browser



Now save and go back to your browser. Cool beans.

Adding functions to the button



```
const Buttons = ({ onStop, onStart, onReset, onSelectActivity }) => {
  const navigate = useNavigate();
  Codeium: Refactor | Explain | Generate JSDoc | X
  const handleButtonClick = (event) => {
    const activity = event.target.value;
    onSelectActivity(activity);
  };
  Codeium: Refactor | Explain | Generate JSDoc | X
  const handleGraphClick = (event) => {
    handleButtonClick(event);
    navigate('/chart');
  };

  return (
    <>
    <div className="activity-buttons-container">
      <button value="Javascript" onClick={handleButtonClick}>Javascript</button>
      <button value="Java" onClick={handleButtonClick}>Java</button>
      <button value="C++" onClick={handleButtonClick}>C++</button>
      <button value="Python" onClick={handleButtonClick}>Python</button>
      <button value="Go" onClick={handleButtonClick}>Go</button>
    </div>
    <div className = "action-buttons-container">
      <button id="stop-button" onClick={onStop}>Stop</button>
      <button id="start-button" onClick = {onStart}>Start</button>
      <button id="reset-button" onClick = {onReset}>Reset</button>
      <button onClick ={handleGraphClick}>Graph</button>
    </div>
    </>
  );
};

export default Buttons;
```

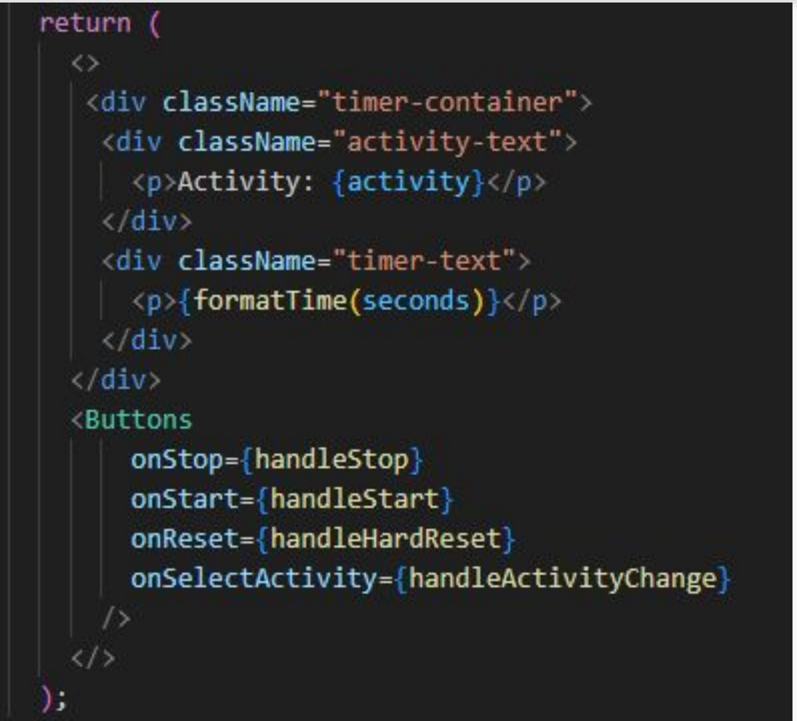
The Buttons components receive **onStop**, **onStart**, **onReset** and **onSelectActivity** as “props”.

event.target.value captures the value of the button. It Passes the selected activity to the **onSelectActivity** function for further processing.

Go back to the Buttons component. Set the value of each button to their respective name and set the onClick action to **handleButtonClick**

Set the OnClick action to **onStop**, **onStart**, **onReset.** This will trigger their respective functions in the **Timer** component.

Adding props to the buttons

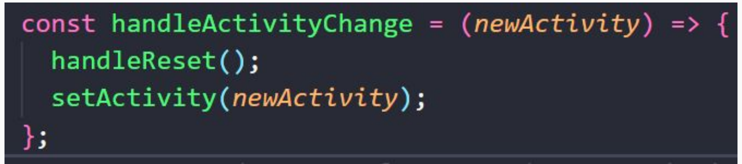


```
return (
  <>
   <div className="timer-container">
    <div className="activity-text">
      <p>Activity: {activity}</p>
    </div>
    <div className="timer-text">
      <p>{formatTime(seconds)}</p>
    </div>
  </div>
  <Buttons
      onStop={handleStop}
      onStart={handleStart}
      onReset={handleHardReset}
      onSelectActivity={handleActivityChange}
    />
  </>
);
```

Add the props to the Buttons component in Timer.jsx.

When a Button is Clicked:

1. The button triggers its associated onClick event handler in Buttons.
2. The corresponding prop (e.g., onStart, onReset) is invoked.
3. The parent component's function (e.g., handleStart) runs the logic for that action.



```
const handleActivityChange = (newActivity) => {
  handleReset();
  setActivity(newActivity);
};
```

Storing data to localstorage

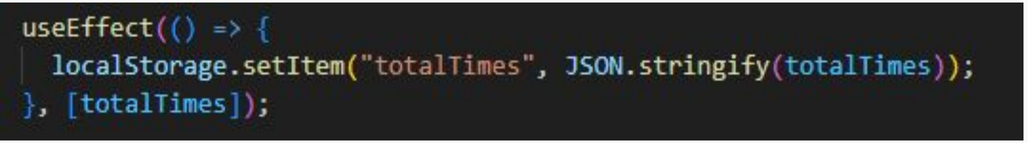
Now your app works! But what happens if you refresh the page or switch to a different activity? You lose all your elapsed time on your activity!

We can save our activity times by saving it to localstorage instead. Even if you close the tab, when you come back, all your progress will be recorded.

Cool beans.

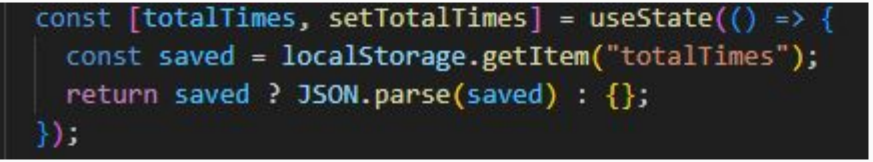
Storing data to localstorage

Open Timer.jsx and do the following



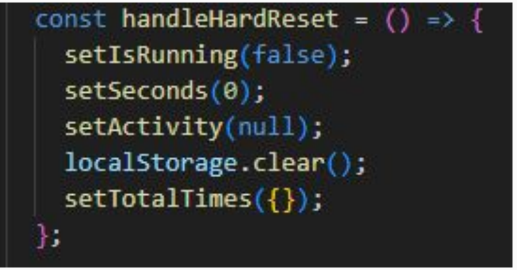
```
useEffect(() => {
  localStorage.setItem("totalTimes", JSON.stringify(totalTimes));
}, [totalTimes]);
```

Whenever **totalTimes** state is updated, we store it into localStorage.



```
const [totalTimes, setTotalTimes] = useState(() => {
  const saved = localStorage.getItem("totalTimes");
  return saved ? JSON.parse(saved) : {};
});
```

If there exists a **totalTimes** item in localStorage we parse the item into our **totalTimes** state otherwise we set our state as an empty javaScript object.



```
const handleHardReset = () => {
  setIsRunning(false);
  setSeconds(0);
  setActivity(null);
  localStorage.clear();
  setTotalTimes({});
};
```

Hard reset to clear localStorage if you ever feel like resetting your progress.

Setting up React Router

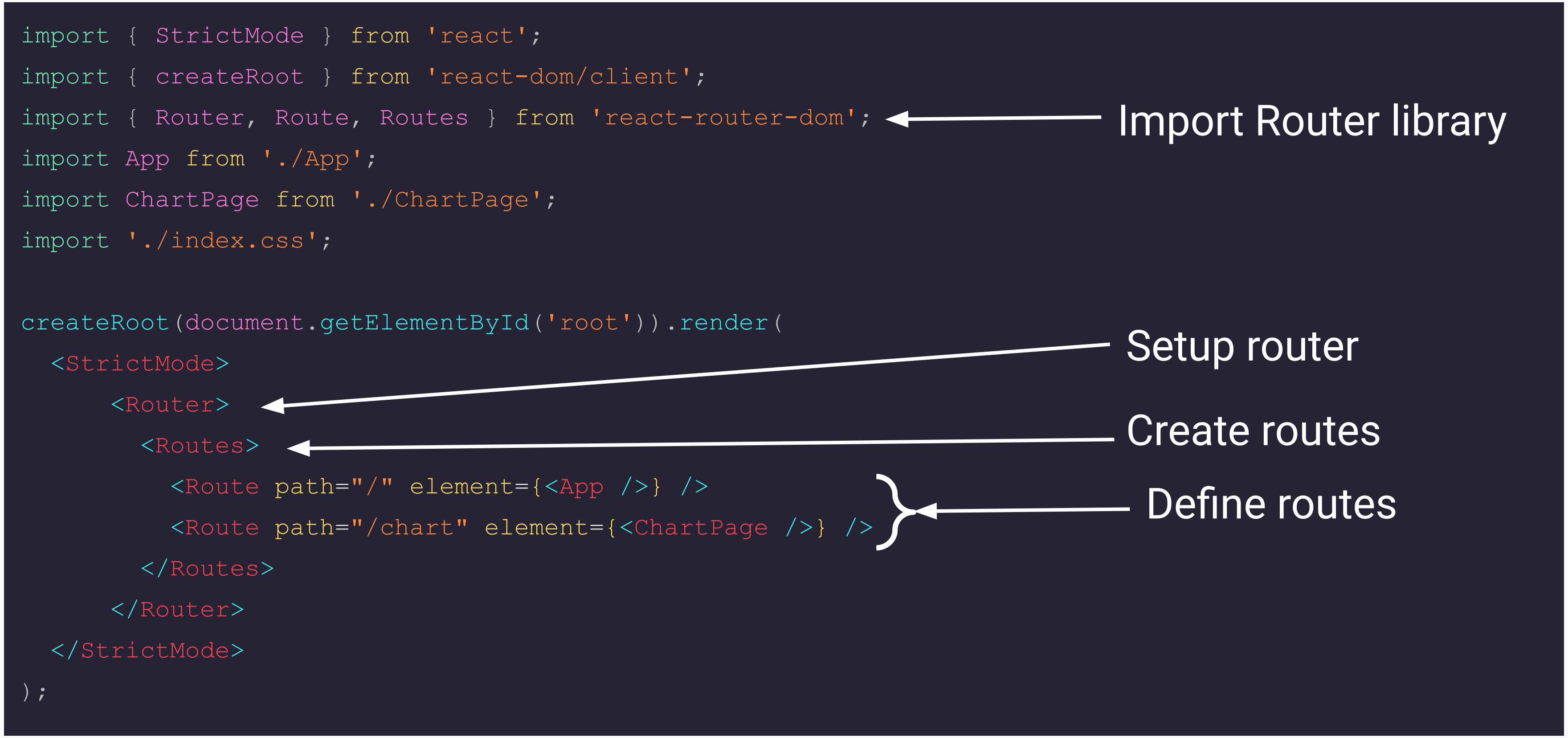
Now that we have the Timer/Main page setup, we need to create another page to show the chart. In order to show multiple pages, we would need to create a React Router, to route users to different pages.

In CMD/powershell/terminal, run
npm install react-router-dom

Here's what we'll do:

1. Import `import { BrowserRouter as Router, Route, Routes } from 'react-router-dom';` in main.jsx
2. Import `import ChartPage from './ChartPage';` (this is the component of the page)
3. Set up router with routes

Code for main.jsx



```jsx
import { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';
import { Router, Route, Routes } from 'react-router-dom';
import App from './App';
import ChartPage from './ChartPage';
import './index.css';

createRoot(document.getElementById('root')).render(
  <StrictMode>
      <Router>
        <Routes>
          <Route path="/" element={<App />} />
          <Route path="/chart" element={<ChartPage />} />
        </Routes>
      </Router>
  </StrictMode>
);
```

Looking Closer

```
<Route path="/" element={<App />} />


<Route path="/chart" element={<ChartPage />} />
```

The page/component to render,
(We imported this at the top of main.jsx)

Define the path for this route,for example this would be:
http://localhost:5173**/chart**

Setting up Chart.js

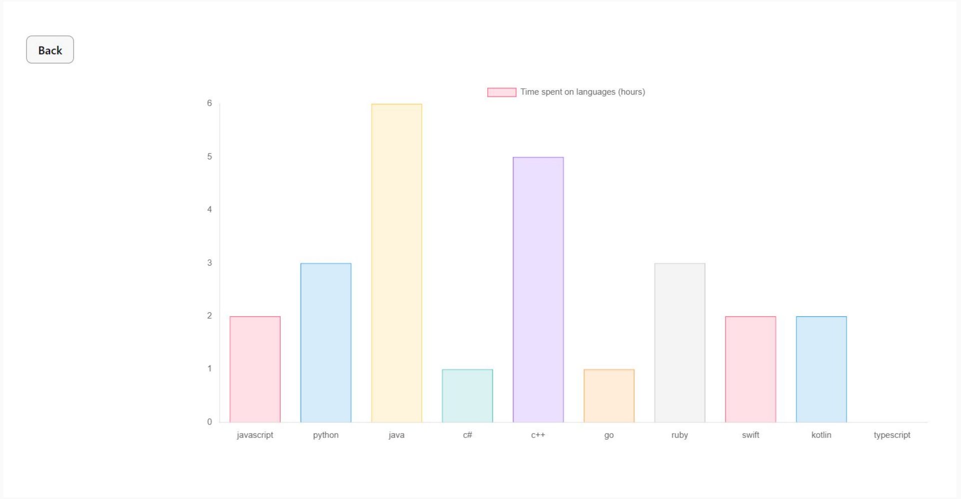
For this demo, we will be using Chart.js to display our tracked languages, along with their respective times.

In order to install Chart.js, following the documentation:

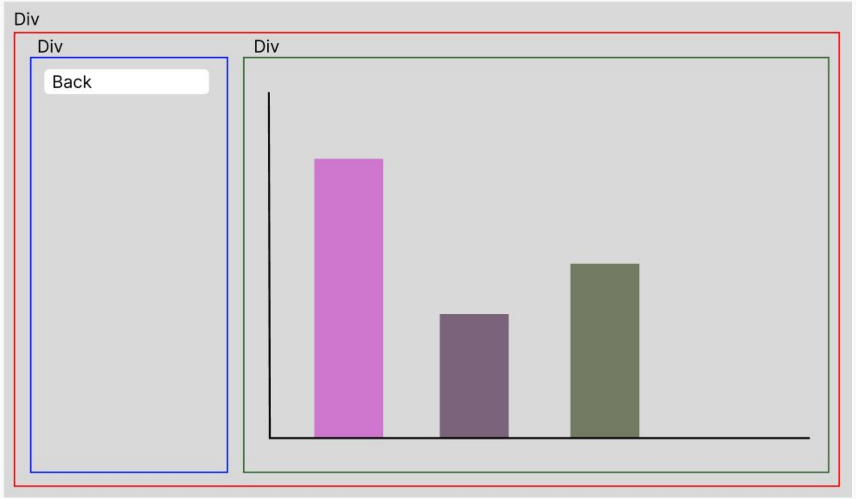
1. Open/create package.json
2. Paste the following code
3. Navigate to the folder with package.json
4. Run `npm install`

```
{
  "name": "chartjs-example",
  "version": "1.0.0",
  "license": "MIT",
  "scripts": {
    "dev": "parcel
src/index.html",
    "build": "parcel build
src/index.html"
  },
  "devDependencies": {
    "parcel": "^2.6.2"
  },
  "dependencies": {
    "@cubejs-client/core":
"^0.31.0",
    "chart.js": "^4.0.0"
  }
}
```

Displaying Data with Chart

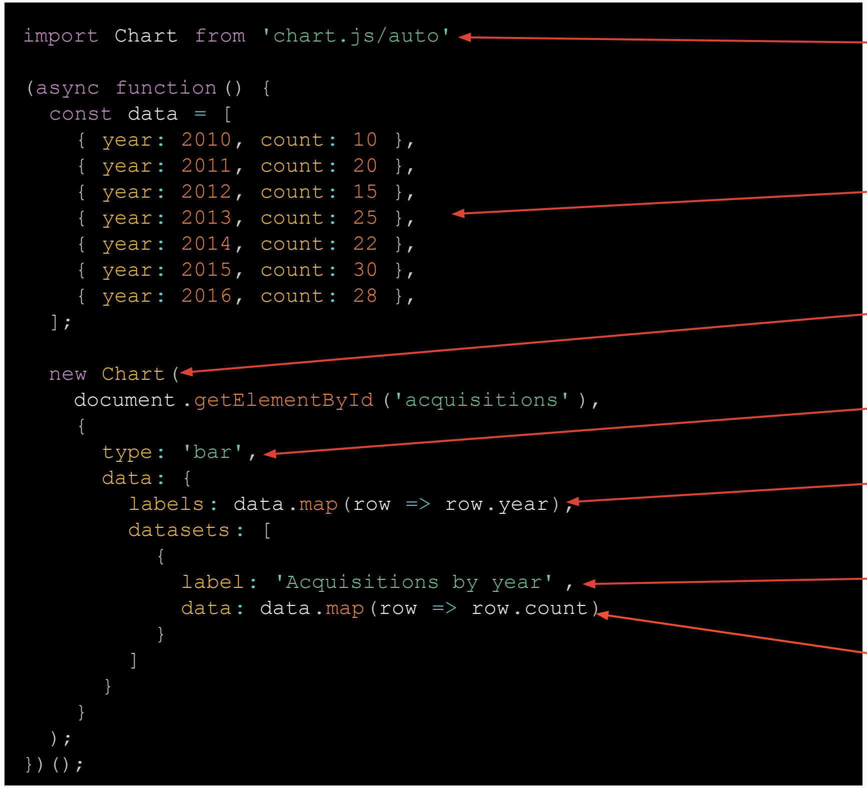


This is what we want to achieve



The basic layout/components of the page

How Chart.js works



```
import Chart from 'chart.js/auto'

(async function() {
  const data = [
    { year: 2010, count: 10 },
    { year: 2011, count: 20 },
    { year: 2012, count: 15 },
    { year: 2013, count: 25 },
    { year: 2014, count: 22 },
    { year: 2015, count: 30 },
    { year: 2016, count: 28 },
  ];

  new Chart(
    document.getElementById('acquisitions'),
    {
      type: 'bar',
      data: {
        labels: data.map(row => row.year),
        datasets: [
          {
            label: 'Acquisitions by year' ,
            data: data.map(row => row.count)
          }
        ]
      }
    }
  );
})();
```

Here, we are importing the Chart library from the package that we just installed.

Data is provided in a List, where each entry of the list is a key-value pair (dictionary).

A new Chart instance is created, and the component with a certain 'id' is passed to the instance.

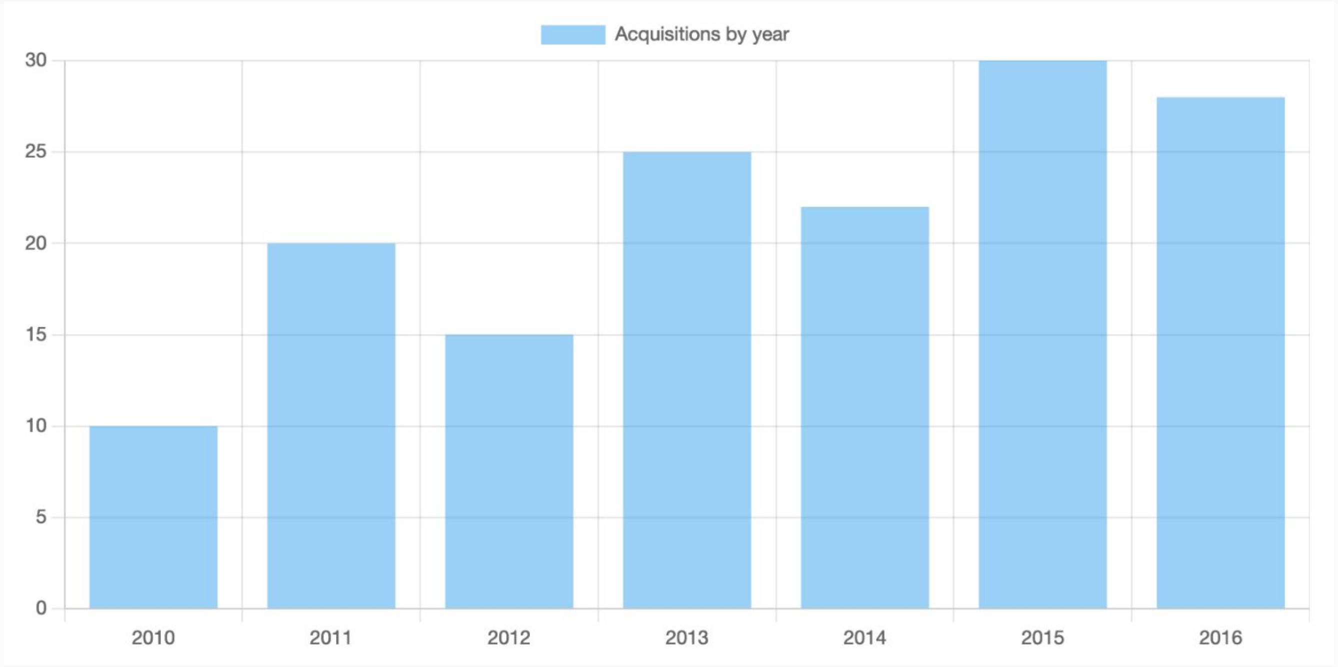
Type of chart

Label for each data entry (in this case each bar)

Label at top of chart (like a title)

Data that defines the height/value of each bar

What the sample would look like



Changes for our use case

Usually, Chart.js is used as a function that targets a ‘div’ or a component with a certain ‘id’, and this function inserts the chart to be displayed in that ‘div’.

However, since we’re using React, we need to instead return a component containing the Chart, rather than targeting a ‘div’ to render the Chart in.

Creating new file

To keep our App.jsx tidy, we create a separate .jsx file for our Chart component.

1. Create a ChartPage.jsx (for the page that contains the back button and the Chart Component)
2. Create ChartComponent.jsx (for the Chart Component itself)

** note that both these files should be in the same directory

We will provide the ChartPage.jsx to save you the effort of designing the page, and instead just focus on making the Chart.

ChartPage.jsx

```jsx
import React from 'react';
import { useNavigate } from 'react-router-dom';
import ChartComponent from './chartComponent';
import './App.css';

function ChartPage() {
  const navigate = useNavigate();
  function handleBack() {
    navigate('/');
  }
  return (
    <>
    <div style={{display:'flex',width:'100%',height:'100%'}}>
      <div>
        <button className='backButton' onClick={handleBack}>
          Back
        </button>
      </div>
      <div style={{width:'60%',justifyContent:'center',alignContent:'center',display:'flex',margin:'100px auto'}}>
        <ChartComponent />
      </div>
    </div>

    </>
  );
}
export default ChartPage;
```

Going Forward

Now that we have ChartPage.jsx set up, we just need to focus on editing the ChartComponent.jsx file.

Stage 1: Getting data from localStorage

In the Timer section, we saved the timings in localStorage, and now we have to retrieve the data, in order to render the bar chart.

1. useEffect() hook to access external functions
2. Use localStorage.getItem('totalTimes') to fetch saved data
3. JSON.parse() the fetched data to get JSON object
4. Loop over each JSON entry and add them into a list of key-value pairs
5. Pass Data to Chart instance

Code example

```
useEffect(() => {
  // Retrieve and parse the data from localStorage
  // get totalTimes from localstorage
  let totalTimes = JSON.parse(localStorage.getItem('totalTimes'));
  // go through totalTimes and convert it to an array of dicts
  let storedData = Object.entries(totalTimes).map(([language, time]) => (
     {
      language,
      time,
     }
  ));
}, []);
```

Fetching data from localStorage
And parsing to JSON

Going through each entry, keeping track of the current language (used as key) and time (used as value)

Creating a key-value pair of the current language and time

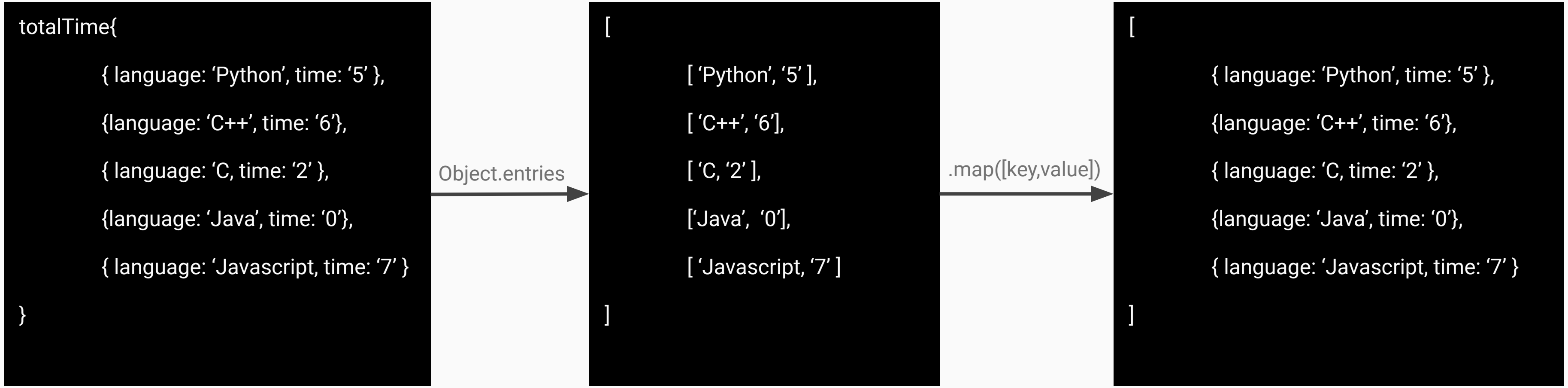
Make a list of each entry in the totalTimes JSON object

Stage 2: Setting up Chart

Since we are going to be returning a component, the chart still needs something to target/mount to, so we are going to use the ‘useRef()’ hook.

1. Set up ‘chartRef’ to reference the <canvas> to render chart
2. Set up ‘chartInstance’ to reference the instance of chart rendered, in order to check if there is already an instance, and clear it if there is.
3. Use `chartRef.current.getContext('2d')` to get 2D context to draw on <canvas>
4. Create new chart instance
5. Pass data into Chart and map through each entry

How our data is transformed



```
totalTime{
        { language: ‘Python’, time: ‘5’ },
        {language: ‘C++’, time: ‘6’},
        { language: ‘C, time: ‘2’ },
        {language: ‘Java’, time: ‘0’},
        { language: ‘Javascript, time: ‘7’ }
}
```

```
[
        [ ‘Python’, ‘5’ ],
        [ ‘C++’, ‘6’],
        [ ‘C, ‘2’ ],
        [‘Java’,  ‘0’],
        [ ‘Javascript, ‘7’ ]
]
```

```
[
        { language: ‘Python’, time: ‘5’ },
        {language: ‘C++’, time: ‘6’},
        { language: ‘C, time: ‘2’ },
        {language: ‘Java’, time: ‘0’},
        { language: ‘Javascript, time: ‘7’ }
]
```

JSON is turned into a nested list of values

Loop through each entry, with key (language) and value (time), and create a key-value pair (dictionary).

```
const chartRef = useRef(null);
 const chartInstanceRef = useRef(null);
 useEffect(() => {
   // {previous slide code will go here(but still same useEffect hook)}
   const ctx = chartRef.current.getContext('2d');
   // Destroy the previous chart instance if it exists
   if (chartInstanceRef.current) {
     chartInstanceRef.current.destroy();
   }
   // Create a new chart instance
   chartInstanceRef.current = new Chart(ctx, {
     type: 'bar',
     data: {
       labels: storedData.map(row => row.language),
       datasets: [
         {
           label: 'Time spent on languages (seconds)',
           data: storedData.map(row => row.time),
         },
       ],
     }
   });

   // Cleanup function to destroy the chart instance when the component unmounts
   return () => {
     if (chartInstanceRef.current) {
       chartInstanceRef.current.destroy();
     }
   };
 }, []);
```

Getting references for <canvas> and Chart instance

Getting 2D context to draw on <canvas>

Check for existing chart instance and destroy it

Setting the labels and data for the Chart

Stage 3: Return the <canvas> component

Now that we set up the functions to render the Chart, we just need to pass the rendered Chart as a <canvas> component.

1. Return a <canvas> component
2. Give the component a ‘ref={chartRef}’ attribute for the functions to target this <canvas>
3. Done!

Final code structure

```
const ChartComponent = () => {
  const chartRef = useRef(null);
  const chartInstanceRef = useRef(null);
  useEffect(() => {
  // all the slides code goes here
  }, []);

  return <canvas id="acquisitions" ref={chartRef}></canvas>;
};
```

Main function of this file

Function returns this component

'Ref' attribute for our functions to target

Final Code (ChartComponent.jsx)

```jsx
import React, { useEffect, useRef } from 'react';
import Chart from 'chart.js/auto';


const ChartComponent = () => {
  const chartRef = useRef(null);
  const chartInstanceRef = useRef(null);
  useEffect(() => {
    // Retrieve and parse the data from localStorage
    // get totalTimes from localstorage
    let totalTimes = JSON.parse(localStorage.getItem('totalTimes'));
    // go through totalTimes and convert it to an array of dicts
    let storedData = Object.entries(totalTimes).map(([language, time]) => ({
        language,
        time,
    }));

    const data = storedData;

    const ctx = chartRef.current.getContext('2d');

    // Destroy the previous chart instance if it exists
    if (chartInstanceRef.current) {
      chartInstanceRef.current.destroy();
    }

    // Create a new chart instance
    chartInstanceRef.current = new Chart(ctx, {
      type: 'bar',
      data: {
        labels: data.map(row => row.language),
        datasets: [
          {
            label: 'Time spent on languages (seconds)',
            data: data.map(row => row.time),
          },
        ],
      }
    });

    // Cleanup function to destroy the chart instance when the component unmounts
    return () => {
      if (chartInstanceRef.current) {
        chartInstanceRef.current.destroy();
      }
    };
  }, []);

  return <canvas id="acquisitions" ref={chartRef}></canvas>;
};

export default ChartComponent;
```