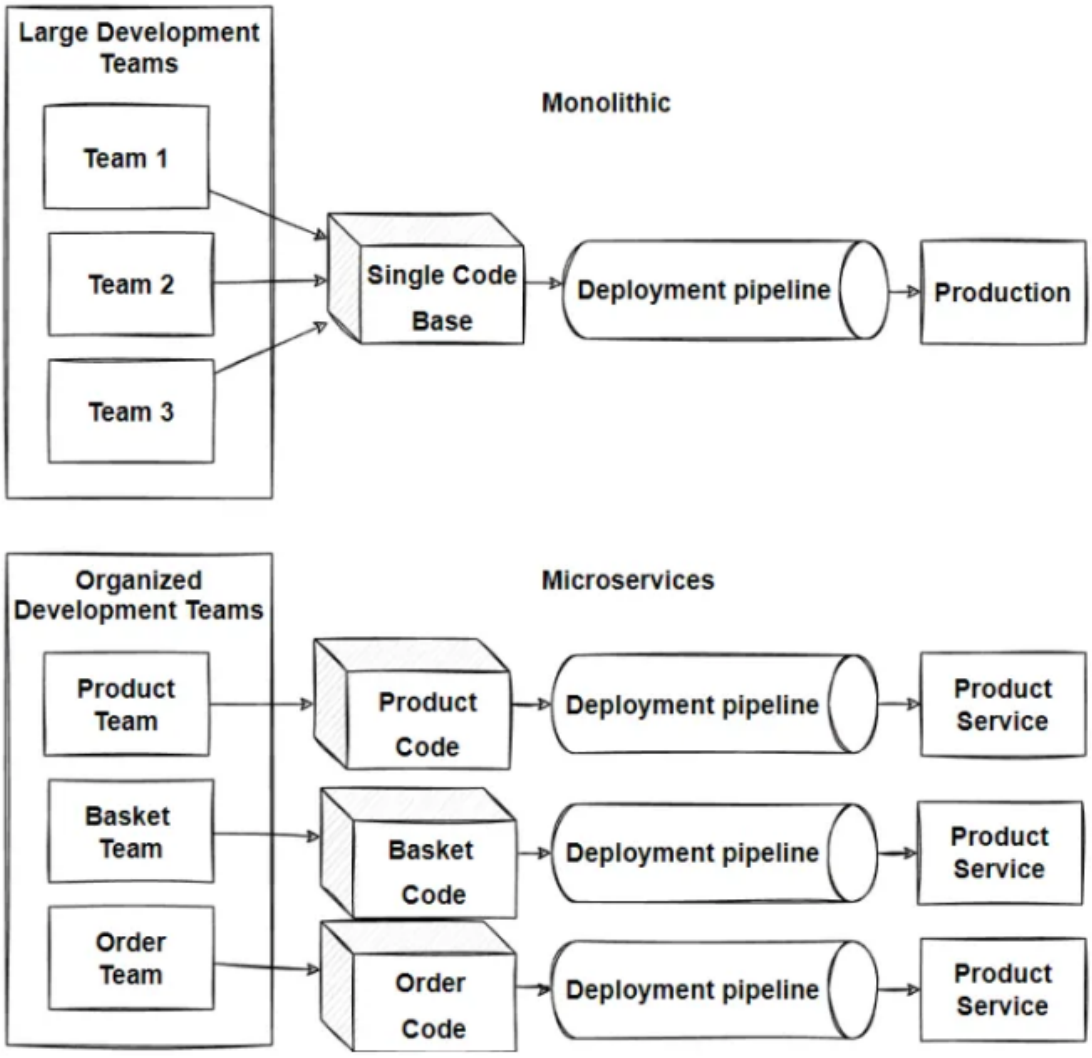
Exploring the World

> 📢 In today's episode, we discussed the trend towards lighter and more adaptable web architectures.

1. What are ‘Monolithic’ and ‘Microservices’ architectures exactly?
    - Understanding ‘Monolith’ and ‘Microservices’ architectures is a big deal in software development, but as developers, it's important to grasp the basics. So, in this episode, we'll break it down into simple terms.
    - 

Monolith Architecture

- In the past, we used to build large projects where everything was bundled together. Imagine building an entire application where all the code—APIs, user interface, database connections, authentication, even notification services—resides in one massive project with single code base.
    - **Size and Complexity Limitation:** Monolithic applications become too large and complex to understand.
    - **Slow Startup:** The application's size can slow down startup time.
    - **Full Deployment Required:** Every update requires redeploying the entire application.
    - **Limited Change Understanding:** It's hard to grasp the full impact of changes,leading to extensive manual testing.
    - **Difficult Continuous Deployment:** Implementing continuous deployment is challenging.

        - **Scaling Challenges:** Different modules may have conflicting resource needs, making scaling difficulty.
        - **Reliability Concerns:** Bugs in any module can crash the whole application, affecting availability.
        - **Adoption of New Technologies:** Making changes in frameworks or languages is expensive and time-consuming since it affects the entire application.

    Microservices Architecture

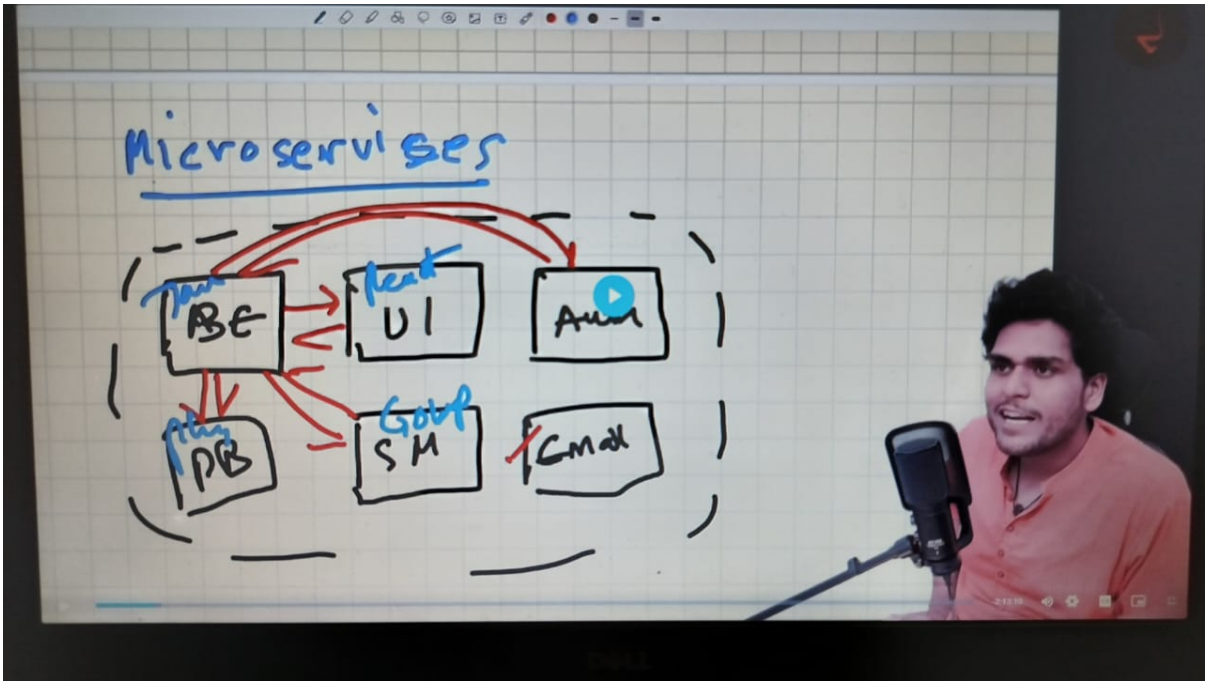
    - The idea is to split your application into a set of smaller, interconnected services instead of building a single monolithic application. Each service handles a specific job, like handling user accounts or managing payments. Inside each service, there's a mini-world of its own, with its own set of rules (business logic) and tools (adapters). Some services talk to each other in different ways, like using REST or messaging. Others might even have their own website!
        - **Simpler Development:** Microservices break down complex applications into smaller, easier-to-handle services. This makes development faster and maintenance easier.
        - **Independent Teams:** Each service can be developed independently by a team focused on that specific task.
        - **Flexibility in Technology:** Developers have the freedom to choose the best technologies for each service, without being tied to choices made at the project's start.
        - **Continuous Deployment:** Microservices allow for independent deployment, enabling continuous deployment for complex applications.
        - **Scalability:** Each service can be scaled independently, ensuring efficient resource usage.
        - **Separation of Concerns:** With each task having its own project, the architecture stays organized and manageable.
        - **Single Responsibility:** Every service has its own job, following the principle of single responsibility. This ensures focused and efficient development.

2. **Why Microservices?**

    - Breaking things down into microservices helps us work faster and smarter. We can update or replace each piece without causing a fuss. It's like having a well-oiled machine where each part does its job perfectly.

3. **How do these services interact with each other?**

- In our setup, the UI microservice is written in React, which handles the user interface.



- Advantage of Microservices that you can have different tech stack for different things.
    - suppose in `Monolith` you had a one big project a java application then you have to do everything in java.
    - But in this `Microservices Architecture`,
        - you can have `UI` written in `React`.
        - you can have `Backend` written in `Java`.
        - you can have `DataBase` written in `Python`.
        - you can have `SMS` service written in `go-lang`.
    - you can write your micro services in any architecture you want.

Communication Channels

- These services interact with each other through various communication channels. For instance, the UI microservice might need data from the backend microservice, which in turn might need to access the database

Ports and Domain Mapping

- Each microservice runs on its specific port. This means that different services can be deployed independently, with each one assigned to a different port. All these ports are then mapped to a domain name, providing a unified access point for the entire application.

Example :

Suppose the `backend` is mapped to `slash api`

```
namasteDev.com/api
```

- All these services API are deployed onto the same URL.

Example :

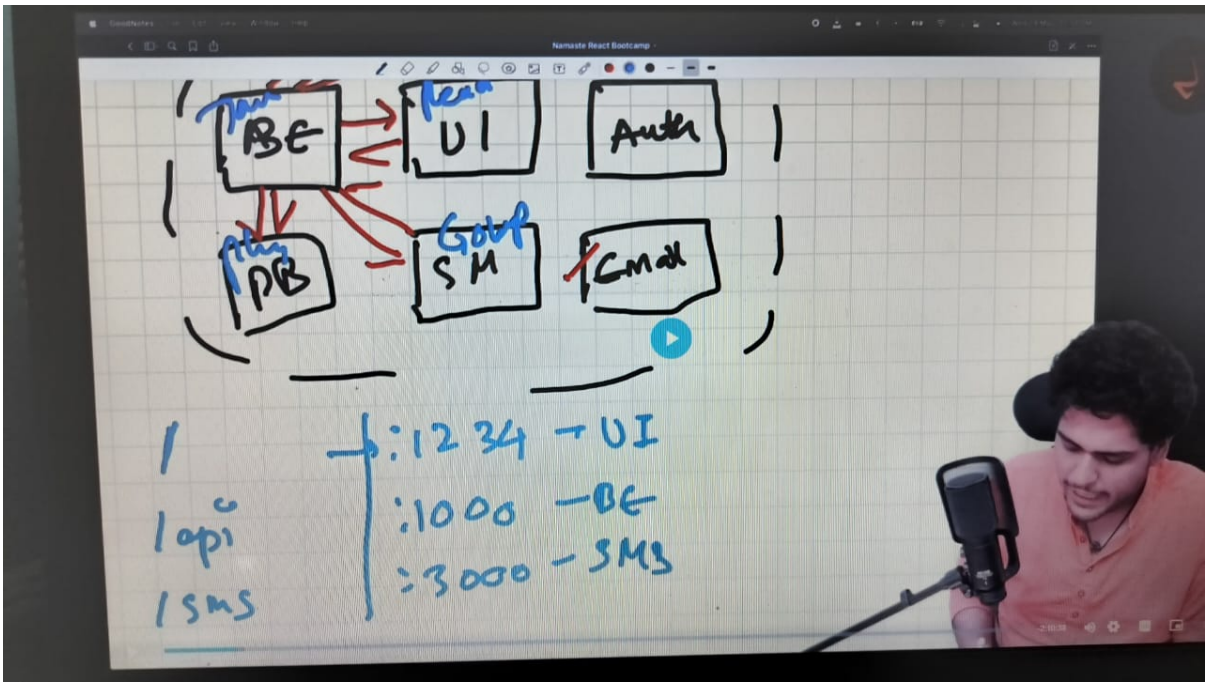
Suppose, we can have `SMS` on `slash sms`

```
namasteDev.com/sms
```

- We can deploy the `UI` on `just slash` and so on.

```
namasteDev.com/
```

- As soon as it hits the main domain, it just redirects to this port 1234



- These 1234, 1000, 3000 are the ports number.

> IMPORTANT

- They make a call to different URL's. Suppose if `UI` wants to connect to `backend`, they will make a call to `slash api`. So that's how these services are connected. and that's how they interact each other.

Connecting to the External World

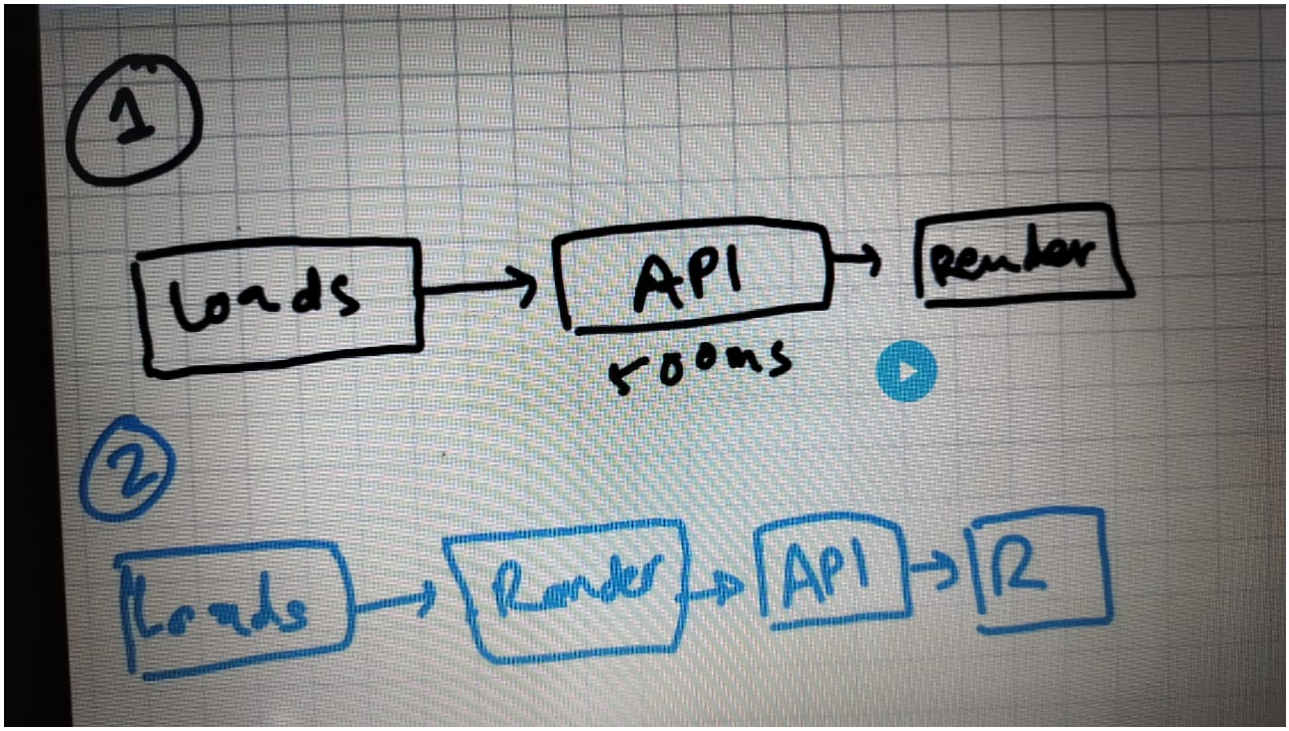
- In this episode, we're going to explore how our React application communicates with the outside world. We'll dive into how our application fetches data and seamlessly integrates it into the user interface. It's all about understanding data exchange that makes our app come alive.
- In our Body component, we're displaying a list of restaurants. Initially, we used mock data inside the `useState()` hook to create a state variable. However, in this episode, we're stepping up our game by fetching real-time data from Swiggy's API and displaying it dynamically on the screen. How cool is that? 😎

> **Very Important topic for Fast Loading**

Before diving in, let's understand two approaches to fetch and render the data :

1. **Load and Render:**
    - We can make the API call as soon as the app loads, fetch the data, and render it.
2. **Render First Fetch Later:**
    - Alternatively, we can quickly render the UI when the page loads we could show the structure of the web page, and then make the API call. Once we get the data, we re-render the application to display the updated information.
    - In React, we're opting for the second approach. This approach enhances user experience by rendering the UI swiftly and then seamlessly updating it once we receive the data from the API call.

Understand the concept from below image :



useEffect()

Today, we're diving into another important topic ‘ `useEffect()` ’ We've mentioned it before in a previous episode.

Essentially, ‘ `useEffect()` ’ is a Hook, React provides us, it is a regular JavaScript function, to help manage our components.

To start exploring its purpose, let's first import it from React.

```
import { useEffect } from "react";
```

`useEffect()` takes two arguments .

- Callback function.
- Dependency Array.

Syntax of useEffect()

```
// We passed Arrow function as callback function.

useEffect(() => {}, []);
```

> `Parameters` are the variables defined in a function's declaration, while `Arguments` are the actual values passed to a function when it is called.

1. **When will the callback function get called inside the useEffect()?**
    - Callback function is getting called after the whole component get rendered.
    - In our app we are using `useEffect()` inside Body component. So it will get called once Body component complete its render cycle.
    - If we have to do something after the rendercycle complets we can pass it inside the `useEffect()`. This is the actual use case of useEffect. It is really helpful to render data which we will get after the `fetch()` operation and we are going to follow second approach which we have discussed already.
2. **Where we fetch the data?**
    - Inside the `useEffect()`, we use `fetchData()` function to fetch data from the external world. don't worry we will see each and every steps in detail.
    - logic of fetching the data is exactly the same that we used to do in javascript.
    - here we are fetching the swiggy's API by using `fetch()` method.

> 📢 IMPORTANT: If getting difficulty to understand `fetch()`, Don't worry please read about how `fetch()` works.

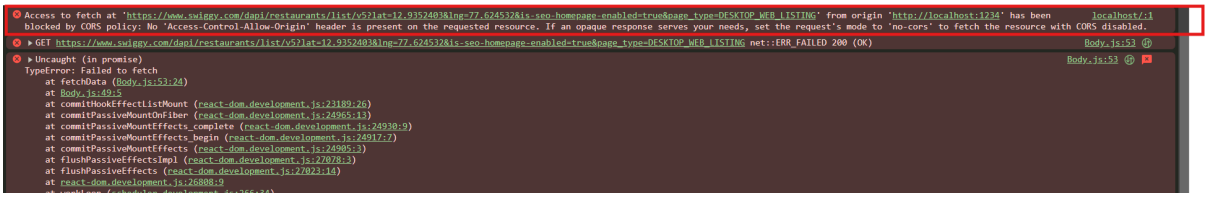
- `fetch()` is given to us by browsers which JS engine has.

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Basic_concepts

3. **How can we use Swiggys API in our App?**
    - We know that `fetch()` always returns a promise to us. we can handle response using `.then()` method.
    - but here we are using newer approach using `'async/await'` to handle the promise.
    - we convert this data to javascript object by using `.json`.

```
// here once the body component would have been rendered , we will fetch the
data

useEffect(() => {
fetchData();
```

```
}, []);

const fetchData = async () => {
const data = await fetch(
    "https://www.swiggy.com/dapi/restaurants/list/v5?
lat=19.9615398&lng=79.2961468&is-seo-homepage-
enabled=true&page_type=DESKTOP_WEB_LISTING"
);
const json = await data.json();
console.log(json);
};
```

4. By using above code let's see can we able to call swiggy's api sucessfully or not?
    - We got an error 😅
    - 

5. **What is the reasone we got that error?**
    - Basically calling swiggy's API from local host has been blocked due to CORS policy.
6. **What exactly the CORS policy is?**
    - (Cross-Origin Resource Sharing) is a system, consisting of transmitting HTTP headers, that determines whether browsers block frontend JavaScript code from accessing responses for cross-origin requests.
    - In simpler terms, CORS (Cross-Origin Resource Sharing) is a security feature implemented by browsers that restricts web pages from making requests to a different origin (domain) than the one from which it was served. Therefore, when trying to call Swiggy's API from localhost, the browser blocks the request due to CORS restrictions.

      > IMPORTANT: If getting difficulty to understand CORS, Don't worry please read the below document.

      https://developer.mozilla.org/en-US/docs/Glossary/CORS

      > IMPORTANT! To prevent CORS errors when using APIs, utilize a CORS extension and activate it.

      > 📢 IMPORTANT! In future swiggy definately change their API data so always remember go to swiggy's website and copy the updated URL of API to fetch data.

    - To show the new data on our page, we just need to update the `listOfRestaurant` with the fresh info. React will then refresh the page to display the updated data.
7. **How do we Update the data ?**

- We're updating the `listOfRestaurant` using a state variable we've already defined. We simply use the `setlistOfRestaurant()` function to replace the old data with the new.

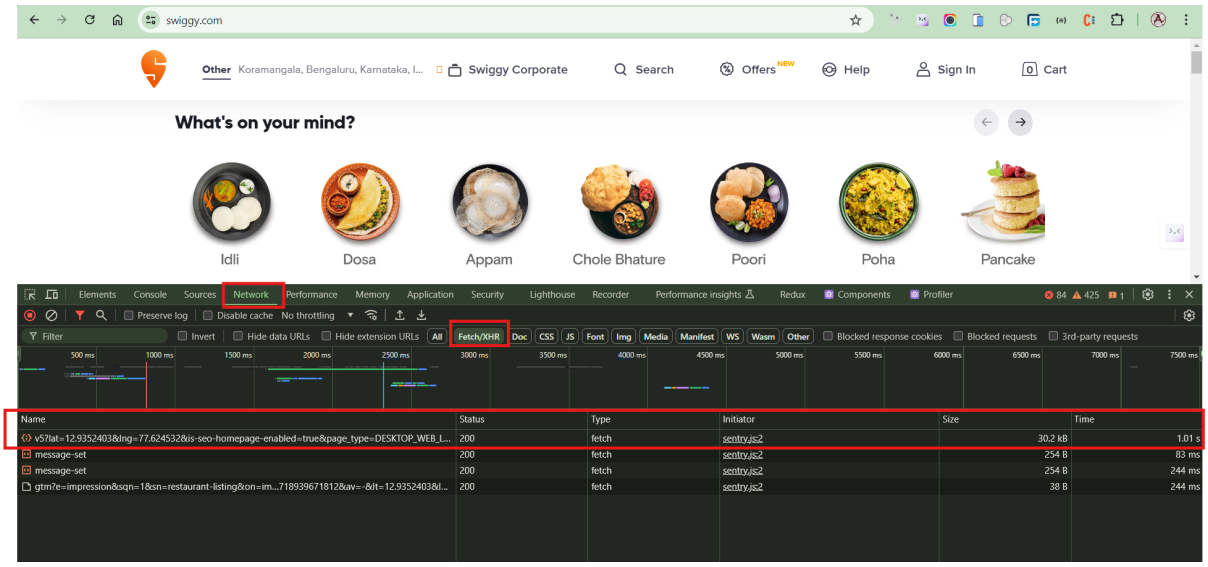
```
const fetchData = async () => {
const data = await fetch(
    "https://www.swiggy.com/dapi/restaurants/list/v5?
lat=19.9615398&lng=79.2961468&is-seo-homepage-
enabled=true&page_type=DESKTOP_WEB_LISTING"
);

const json = await data.json();
setListOfRestaurant(
    json.data.cards[4].card.card.gridElements.infoWithStyle.restaurants
);
};
```

```
 > IMPORTANT:
 As we delve into the JSON data, it's essential to note its complexity. Our
focus lies solely on extracting cards which have restaurant information for
our project.

 Attempting to directly implement the code snippet provided here definitely
results in errors due to potential changes in Swiggy's API structure. Your
understanding is greatly appreciated during this phase. Focus on the concept
of whatever Akshay taught us in this Episode.

 NOTE: In the upcoming episode, ‘Akshay’ addresses all API-related issues,
ensuring a smoother experience. So, don't stress—everything will be resolved
in the upcoming episode.

 Happy coding!
```

8. **How to get an `API`?**

- Open the website of swiggy
- Go to `Network Tab`
- select the `Fetch/XHR`
- Reload the website
- Cool - you will see the link under the `name` coloumn
  - confirm by checking these things
    - `type` : fetch
    - `Size` : should be in `KB`
    - `Name` : start with `object symbol` & name `v5` or something else
- Just double click on the `Name` , it will open in new tab
- Congrats, got the data.

> Below images are just for reference.

- 

- 

Part - 2

After fetching the data, there's a noticeable one-second delay before it appears on the screen. This delay occurs because the APIs take some time to load.

Improving this can enhance the user experience.

1. **How could we improve it?**
   - To enhance the user experience, we could add a `spinning loader` that appears while we wait for the data to load from the APIs. This provides visual feedback to the user and indicates that the application is working to retrieve the information.
   - We could implement a condition to display a `spinning loader` if our list of restaurants hasn't received any data yet.

```
if (listOfRestaurant.length === 0) {
    return <h1>loading. . .</h1>;
}
```

- Refreshing the page to see the result,but this isn't an ideal approach. Instead, we can enhance the user experience by implementing a `‘Shimmer UI’`.

Shimmer UI

- Shimmer UI is a technique that shows placeholder content while data is loading, reducing wait time and keeping users engaged.
- Also known as `Fake Cards`.
- Instead of displaying a generic `"loading"` message, we'll integrate a `<shimmer/>` component within our app to provide visual feedback while data is loading. this concept is known as `conditional rendering`.

```
// conditional rendering
if (listOfRestaurant.length === 0) {
    return <Shimmer/>;
}
```

Conditional rendering

- Rendering on the basis of specific conditions is known as `Conditional rendering`.
- In React, conditional rendering can be achieved using JavaScript expressions within JSX. Here's a simple example:

```
import React from 'react';

function App() {
const isLoggedIn = true;

return (
    <div>
    {isLoggedIn ? <p>Welcome, User!</p> : <p>Please log in to continue.</p>}
    </div>
);
}

export default App;
```

- In this example, the `<p>` element will only be rendered if the `isLoggedIn` variable is `true`. If `isLoggedIn` is `false`, a different message will be displayed.
- Conditional rendering can also be used to render different components based on conditions, handle loading states, or display error messages. By using conditional rendering effectively, you can create dynamic and interactive web applications that respond appropriately to different scenarios.
- **It has 4 different approaches :**

- `If /else:`
- `Element variables:` In this, we use Js variables to store elements. It will also help to conditionally render the entire component or only a part of the component as well.

```
let message;
    if (this.state.isLoggedIn) {
        message = <div>Welcome Aditya to conditional rendering through
element variable</div>
    } else {
        message = <div>Welcome Ranjan to conditional rendering through
element variable</div>
    }
    return (
    <div>
        {message}
    </div>
    )
```

- `Ternary conditional operator:` we can use it inside the JSX. It is simpler.

```
return (
    <div>
        {this.state.isLoggedIn ? "Welcome Aditya to ternary conditional" :
"Welcome Ranjan to ternary conditional"}
    </div>
    )
```

- `Short circuit operator:` when you want to render either something or nothing. If the left hand side is true, then it also evaluates the right hand side. However, if the left hand side is false, then the right hand side will never be evaluated. As it doesn't affect the final value of the whole expression. Make sure to add the return keyword at the beginning of the statement.

```
return this.state.isLoggedIn && <div>Welcome Aditya to short circuit
operator.</div>
```

2. **Why do we need State variable? why we can't use normal Javascript variable?**
   - Many developers have this confusion today we will see that Why with the help of following example:
   - To understand this we will introduce on feature in our app is a `'login/logout'` button
   - Inside Header component we are adding the button look at the code given below. also we want to make that login keyword dynamic it should change to logout after clicking.
   - `step1 —->`

        - We create `btnName` variable with login string stored in it and we are going use that btnName as a button text look at the code below

- `step2 —>`

    - Upon clicking this button, it changes to `'logout'`.

```
const btnName = "Login";

return (
    <div className="container header">
        <a>logo</a>
        {navItems}

        <button
        className="login"
        onclick={() => {
            btnName = "Logout";
            }}
        >
        {btnName}
        </button>
    </div>
    );
```

- But it will not change 😒.
- It's frustrating that despite updating the `btnName` value and seeing the change reflected in the `console`, but the UI remains unchanged.
- This happens because we're treating `btnName` as a regular variable.
- To address this issue, we need a mechanism that triggers a `UI refresh` whenever `btnName` is updated.
- To ensure UI updates reflect changes in `btnName`, we may need to use state management that automatically refreshes the UI when data changes.
- That's the reasone we need state variable `useState()`.

Let's utilize `reactBtn` as a state variable using `useState()` instead of *`btnName`*

Here's the code:

```
const [reactBtn, setReactBtn] = useState("login");
```

To update the default value of `reactBtn`, we use `setReactBtn` function.

> 📢 NOTE: In React, we can't directly update a state variable like we would use a normal JavaScript variable. Instead, we must use the function provided by the `useState()` hook. This function

> allows us to update the state and triggers a re-render of the component, ensuring our UI is always up-to-date with the latest state.

With the code provided below, we've enhanced the functionality of our app. Now, we can seamlessly toggle between `"login"` and `"logout"` states using a ternary operator. This addition greatly improves the user experience.

```
const [loginBtn, setLoginBtn] = useState("Login");

<button
    className="nav-link"
    onClick={() => {
    loginBtn === "Login"
        ? setLoginBtn("Logout")
        : setLoginBtn("Login");
    }}
>
    {loginBtn}
</button>
```

> NOTE : The interesting aspect of the above example is how we manage to modify a const variable like `reactBtn`, which traditionally isn't possible. However, because React rerenders the entire component when a state variable changes, it essentially creates a new instance of `reactBtn` with the updated value. So, in essence, we're not updating `reactBtn` instead, React creates a new one with the modified value each time the state changes. This is the beauty of React.

> Important question

- **Once login name changed then it only rendered the updated `value` or the whole `Header` component?**
    - It re-rendered the whole `Header` component.

3. **When to useState()?**

    - If you want to make your component dynamic
    - if you want someting should change in your component

    We use local state variables. Here, `useState()` comes into the picture.

Search Functionality

When you input text into the search field, it provides suggestions based on the data related to restaurants that we already have.

`step 1 —->`

Let's create a search bar within a `<div>` element and assign any class name of your choice to it. Additionally, we'll give class names to the input field and button inside the search bar.

`step 2 —->`

Upon clicking the button, filter the restaurant cards and update the UI to retrieve data from the input box. To link our input to the button, we'll use the `value` attribute within the input field and bind that value to a local state variable. We'll create a local state variable named `searchText` along with a function named `setSearchText` to update the value. lets see below code will work or not by simple puting the call back function.

```
const [searchText, setsearchText] = useState("");
<div className="search">
  <input type="text" className="search-box" value={searchText} />
  <button className="searchBtn" onClick={() => {
  console.log(searchText);
  }}>
    Search
  </button>
</div>;
```

we could see that our input not taking value. we unable to type any thing.

- we knew already, we have bind this searchText to the input field. what ever is inside the searchText variable will inside the value attribute of the input field.
- when we will change the value of input field by typing on it still it will tied to the searchText but searchText is not Updating . because default value of search text is empty string .this is most important point to understand whole concept. this input box not changed unless we change the search text.

1. **How could we solve this problem ?**

To solve this, we have to add `onChange` eventHandler inside the input field, so as soon as input changes the onchange call back function should also be changed the input text.

Inside the onchange event handler we have event `'e'` inside the call back . so access that typed input by using `event 'e'` see the code

```
  <input type="text" className="search-box" value={searchText}
  onchange={(e)=>{setSearchText(e.target.value)}/>
```

based on the `onChange` we have make in the code, now we can type inside the search box and see the output inside the console.

> 📢 NOTE: when ever search text is change on the every key press state variable re render the component. its find the difference between every updated V-DOM with new text added inside the input field with older one.

> `Important :` + Whenever state variables update, react triggers a reconciliation cycle(re-renders the component). + React is re-rendering the whole body component, but it is only updating input box value inside the DOM.

`Step 3 —->`

We're currently filtering the list of restaurants to update the UI. When we type a word in the input field, it filters out the restaurant cards based on whether the typed word matches any restaurant names. However, we're facing a challenge with the input field being case-sensitive. We want the suggestions to be based solely on the word typed, without considering whether it's in uppercase or lowercase.

2. **How could we solve this problem ?**

To fix the problem, we just need to use the code provided. It uses `toLowerCase()` to make our search bar insensitive to capitalization.

```
<button
  className="searchBtn"
  onClick={() => {
    // filter the Restaurant and update the UI
    const filtertheRestaurant = listOfRestaurant.filter((res) => {
      return res.info.name.toLowerCase().includes(searchText.toLowerCase());
    });
    setListOfRestaurant(filtertheRestaurant);
  }}
>
  Search
</button>;
```

`Step 4 —->`

We've encountered another issue in our app: after searching for a restaurant, the UI doesn't render anything when we search again. Instead, we only see the Shimmer UI.

3. **How could we solve this problem ?**

here problem is when we search 1st time we are updating `listOfRestaurents`. If we try to search it again it is searching from previous updated list thats the problem. simple solution for this instead of filtering the original data we simple make a copy to of that original data in our case it is nothing but a `listOfRestaurant` and stored the copy with new variable `filteredRestaurant`.

```
//original
const [listOfRestaurant, setListOfRestaurant] = useState([]);

//copy
const [filteredRestaurant, setFilteredRestaurant] = useState([]);
```

In our code, when we fetch data using the `fetchData` function, it's important to update the rendering to display the new data. We achieve this by updating the state variables `listOfRestaurant` and `filteredListOfRestaurant` using functions provided by the `useState()` hook. Initially, both arrays are empty, but after fetching data, we fill them with the retrieved information. otherwise we won’t see any thing on the page.

```
const fetchData = async () => {
const data = await fetch(
"https://www.swiggy.com/dapi/restaurants/list/v5?lat=19.9615398&ln
g=79.2961468&is-seo-homepage-enabled=true&page_type=DESKTOP_WEB_LISTING"
);
const json = await data.json();
// here we are filling both the variable with new data with the help of their
functions.
setListOfRestaurant(
json.data.cards[4].card.card.gridElements.infoWithStyle.restaurant
s
);
setFilteredRestaurant(
json.data.cards[4].card.card.gridElements.infoWithStyle.restaurant
s
);
};
```

📢IMPORTANT:
here there is two important points to remember

1) When we need to modify the list Of Restaurants based on
certain conditions, we're essentially using the original
data we fetched and stored within the `listOfRestaurant` variable. (original)

2) To display data on the UI, we use a copy of
`listOfRestaurant` called `filteredRestaurant`. (copy)