**Namaste React**

**Ep06 Exploring the World!**

# Revision

## Config Driven UI

Config driven UI is very powerful way of building web app, config driven UI means configuration or JSON Object is sent from the backend most of the times.

It is also possible to hardcode the configuration on the frontend app also.

At the end of the day configuration is just a normal JS object, JSON Data we can say.

So any configuration can power the UI for example depending on the location the offers and the coupon code are different and will be changed depending on the configuration.

## What is useState?

useState is a hook and the hooks are just a normal JS functions.

This hook is coming from react we are importing the hooks from react, because someone has already wrote the code for that hook in react library and exported from there.

useState gives us functionality of creating local state variables, here local state variables are the variables which changes as per the changes in the UI.

If we want to change normal JS variables on some conditions in UI e.g. on click of some button some variable needs to change in this case the normal JS variables it is not possible. Why because whenever we are changing the variables the UI needs to be in sync with the data or the state.

As react keeps track of state variables it is easy to manage state by using useState variable rather than normal JS variables as react does not track the changes.

Whenever there is a small change in the state react will re-render the component with updated content.

# How to do API call in JS/React

We can do any API call in JS by using the fetch function given by the window object, it is a browser API.

## 🔎 Fetch

* The Fetch API provides an interface for fetching resources (including across the network) in an asynchronous manner.
* It returns a **Promise**
* It is an object which contains a single value either a Response or an Error that occurred.
* .then() tells the program what to do once Promise is completed.

### fetch("https://api.github.com/users/anuradha9712")

### .then((response) => {

### return response.json();

### })

### .then((data) => {

### console.log(data);

### });

We can call this function and pass the API address and it will call the that API.

## Where should we do our API Calls?

There are two cases of doing an API call.

Case 1 :

* User loads the page
* Make the API call (300ms)
* Render the page (500ms)

Case 2:

* User loads the page
* Show something on the page/render the initial page
* Call the API
* Update the UI

Here in both the cases case 2 is the best way because, the loading time is less, user does not have to wait for the page to reload. As we know the DOM manipulation is very fast in React case 2 becomes the best way of doing it.

To use this functionality React gives us access to 2nd most important Hook known as useEffect.

## How to use useEffect hook?

As we know at the end of the day hooks are just normal JS functions. We will import it as a named import from react library and we call functions as below,

### useEffect();

but we call useEffect by passing another function to it which is a call back function. This call back function will not be called immediately but whenever useEffect wants to call.

### useEffect(() => {

### console.log("rendered!!!");

### });

useEffect will be called after every re render of the component. To avoid the execution of useEffect after every re render we will give an dependency array to it.

### useEffect(() => {

### console.log("rendered!!!");

### }, []);

e.g. if we want our useEffect to run after state variable searchText changes then we will add the searchText inside the dependency array.

### useEffect(() => {

### console.log("rendered!!!");

### }, [**searchText**]);

whenever there a change in the dependency array useEffect call back function will be called

If useEffect is not dependant on anything i.e. dependency array is empty in this case useEffect will be called only once after the initial render.

So for making an API call we will use useEffect along with an empty dependency array, API call will happen in the end once page is render.

Now the case 2 will become something like below,

Page loads => render the initial page => useEffect API call => re render the page with the updated components.

We will do the API call as below,

### useEffect(() => {

### getAllRestaurants();

### }, []);

### async function getAllRestaurants() {

### const data = await fetch(

### "https://www.swiggy.com/mapi/homepage/getCards?lat=18.6320627&lng=73.84679779999999"

### );

### const AllRestaurants = await data.json;

### return AllRestaurants;

### }

❗So when we use useEffect React renders our component and then useEffect is called once and suppose dependency array has some state Variables in this case our component will render and useEffect will be call once and it will be called every time the state variable changes.

## We have used async and await keywords while calling an API, so what are they?

## Async/await

Asynchronous code can become difficult to follow when it has a lot of things going on. async and await are two keywords that can help make asynchronous read more like synchronous code. This can help code look cleaner while keeping the benefits of asynchronous code.

For example, the two code blocks below do the exact same thing. They both get information from a server, process it, and return a promise.

### function getPersonsInfo(name) {

### return server.getPeople().then(people => {

### return people.find(person => { return person.name === name });

### });

### }

### async function getPersonsInfo(name) {

### const people = await server.getPeople();

### const person = people.find(person => { return person.name === name });

### return person;

### }

The second example looks much more like the kind of functions you are used to writing. However, did you notice the async keyword before the function declaration? How about the await keyword before server.getPeople()?

## The async keyword

The async keyword is what lets the JavaScript engine know that you are declaring an asynchronous function. This is required to use await inside any function. When a function is declared with async, it automatically returns a promise; returning in an async function is the same as resolving a promise. Likewise, throwing an error will reject the promise.

An important thing to understand is async functions are just syntactical sugar for promises.

The async keyword can also be used with any of the ways a function can be created. Said differently: it is valid to use an async function anywhere you can use a normal function.

### const yourAsyncFunction = async () => {

### // do something asynchronously and return a promise

### return result;

### }

### anArray.forEach(async item => {

### // do something asynchronously for each item in 'anArray'

### // one could also use .map here to return an array of promises to use with 'Promise.all()'

### });

### server.getPeople().then(async people => {

### people.forEach(person => {

### // do something asynchronously for each person

### });

### });

## The await keyword

await is pretty simple: it tells JavaScript to wait for an asynchronous action to finish before continuing the function. It’s like a ‘pause until done’ keyword. The await keyword is used to get a value from a function where you would normally use .then(). Instead of calling .then() after the asynchronous function, you would simply assign a variable to the result using await. Then you can use the result in your code as you would in your synchronous code.

Note: You can use await only inside of async functions.

**Error Handling**

Handling errors in async functions is very easy. Promises have the .catch() method for handling rejected promises, and since async functions just return a promise, you can simply call the function, and append a .catch() method to the end.

### asyncFunctionCall().catch(err => {

### console.error(err)

### });

But there is another way: the mighty try/catch block! If you want to handle the error directly inside the async function, you can use try/catch just like you would inside synchronous code.

### async function getPersonsInfo(name) {

### try {

### const people = await server.getPeople();

### const person = people.find(person => { return person.name === name });

### return person;

### } catch (error) {

### // Handle the error any way you'd like

### }

### }

Doing this can look messy, but it is a very easy way to handle errors without appending .catch() after your function calls. How you handle the errors is up to you, and which method you use should be determined by how your code was written. You will get a feel for what needs to be done over time. The assignments will also help you understand how to handle your errors.

async/await works well with Promise.all

When we need to wait for multiple promises, we can wrap them in Promise.all and then await:

### // wait for the array of results

### let results = await Promise.all([

### fetch(url1),

### fetch(url2),

### ...

### ]);

In the case of an error, it propagates as usual, from the failed promise to Promise.all, and then becomes an exception that we can catch using try..catch around the call.

Summary

The async keyword before a function has two effects:

1. Makes it always return a promise.
2. Allows await to be used in it.

The await keyword before a promise makes JavaScript wait until that promise settles, and then:

1. If it’s an error, an exception is generated — same as if throw error were called at that very place.
2. Otherwise, it returns the result.

Together they provide a great framework to write asynchronous code that is easy to both read and write.

With async/await we rarely need to write promise.then/catch, but we still shouldn’t forget that they are based on promises, because sometimes (e.g. in the outermost scope) we have to use these methods. Also Promise.all is nice when we are waiting for many tasks simultaneously.

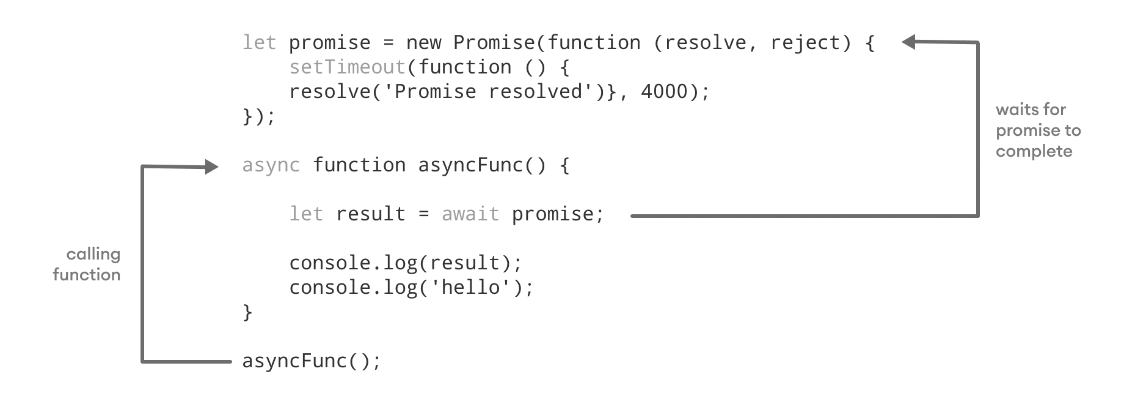


Fig. Working of async/await function

Whenever our page loads there will be API calls which will take some time. In olden days some loading screens were showed and suddenly page loads but after some psychological studies they found that it is a bad user experience.

Instead of the loading screen we can have a skeleton of how our content on the page will look like. This is something known as Shimmer.

## What is shimmer?

A shimmer screen is a version of the UI that doesn’t contain actual content. Instead, it mimics the page’s layout by showing its elements in a shape similar to the actual content as it is loading and becoming available (i.e. when network latency allows).

A shimmer screen is essentially a wireframe of the page, with placeholder boxes for text and images.

A shimmer UI resembles the page’s actual UI, so users will understand how quickly the web or mobile app will load even before the content has shown up. It gives people an idea of what’s about to come and what’s happening (it's currently loading) when a page full of content/data takes more than 3 - 5 seconds to load.