**Conditional Based Asynchronous**

These are tasks that depend on a condition. You want to perform different asynchronous actions based on whether the condition is true or false.

**Callback**

To pass a function in another function as a parameter. Sequence wise task

Functions that are dependent on each other or one another

Callbacks are a fundamental concept in JavaScript and TypeScript for handling asynchronous operations.

Callbacks are functions that you pass as arguments to other functions, and they are executed when a particular operation is completed. In TypeScript, callbacks can be used to work with asynchronous code in a clean and organized manner.

**// Step 1: Define a function for the asynchronous task with callbacks**

**function fetchData(condition: boolean, successCallback: (result: string) => void, errorCallback: (error: Error) => void) {**

**if (condition) {**

**// Step 2: If the condition is true, perform an asynchronous task (e.g., fetching data)**

**setTimeout(() => {**

**const data = "Data fetched successfully";**

**successCallback(data); // Step 3: Call the success callback with the result**

**}, 1000);**

**} else {**

**// Step 4: If the condition is false, simulate an error**

**setTimeout(() => {**

**errorCallback(new Error("Failed to fetch data")); // Step 5: Call the error callback with an error**

**}, 1000);**

**}**

**}**

**// Step 6: Define success and error callback functions**

**function handleSuccess(result: string) {**

**console.log("Success:", result);**

**}**

**function handleError(error: Error) {**

**console.error("Error:", error.message);**

**}**

**// Step 7: Set your condition**

**const condition = true; // Change this to true or false as needed**

**// Step 8: Call the function with callbacks based on the condition**

**fetchData(condition, handleSuccess, handleError);**

**Call back Hell**

Callback hell, also known as "pyramid of doom," is a situation that occurs when you have multiple nested callback functions in your code. It can make your code difficult to read and maintain. Can encounter callback hell when dealing with asynchronous operations. However, modern JavaScript and TypeScript provide solutions like Promises and async/await to mitigate callback hell.

Like

**function fetchData(callback1: (data1: string) => void) {**

**setTimeout(() => {**

**const data1 = "First data";**

**callback1(data1);**

**setTimeout(() => {**

**const data2 = "Second data";**

**callback1(data2);**

**setTimeout(() => {**

**const data3 = "Third data";**

**callback1(data3);**

**}, 1000);**

**}, 1000);**

**}, 1000);**

**}**

**fetchData((data) => {**

**console.log(data);**

**});**

**Above we have multiple nested setTimeout callbacks, creating a pyramid of callbacks. This can quickly become hard to manage as the number of nested operations grows.**

**Promises**

Promises provide a more structured way to handle asynchronous operations. Here's how the previous example can be rewritten using Promises:

**Like**

**function fetchData() {**

**return new Promise<string>((resolve) => {**

**setTimeout(() => {**

**const data = "Data fetched using Promises";**

**resolve(data);**

**}, 1000);**

**});**

**}**

**fetchData()**

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

**console.log(data);**

**})**

**.catch((error) => {**

**console.error(error);**

});

**Async/Await**: The **async** and **await** keywords make it even more concise and readable:

**async function fetchData() {**

**return new Promise<string>((resolve) => {**

**setTimeout(() => {**

**const data = "Data fetched using async/await";**

**resolve(data);**

**}, 1000);**

**});**

**}**

**(async () => {**

**try {**

**const data = await fetchData();**

**console.log(data);**

**} catch (error) {**

**console.error(error);**

**}**

**})();**