Slide Note

Take 3 mins

Is waiting in a queue at a toll plaza a pleasing experience?

Slide Note

Can We Make the Experience At Toll Plaza Pleasing?

- Once you have entered a lane in a toll plaza, there is no option but to wait till your turn comes, you make the payment and move ahead.
- When there is a long queue, it is frustrating.
- However, the experience can be better if an official from the toll booth captures a photo of the registration number of the vehicle crossing the booth and notifies you about the payment later.
- The payment can be done within the given duration.

Slide Note

What Is Common in All the Scenarios?

- Poor user experience
- Irritation caused by delays.
- The next task cannot be initiated until the previous task is completed.
- This mode of execution of tasks is known as a **Synchronous** execution.
- The entities interacting with each other are heavily dependent on each other leading to tightly coupled architecture.

Slide Note

Let us see how restaurants function to ensure a better customer experience.

Slide Note

Restaurant – Loosely Coupled Architecture Model

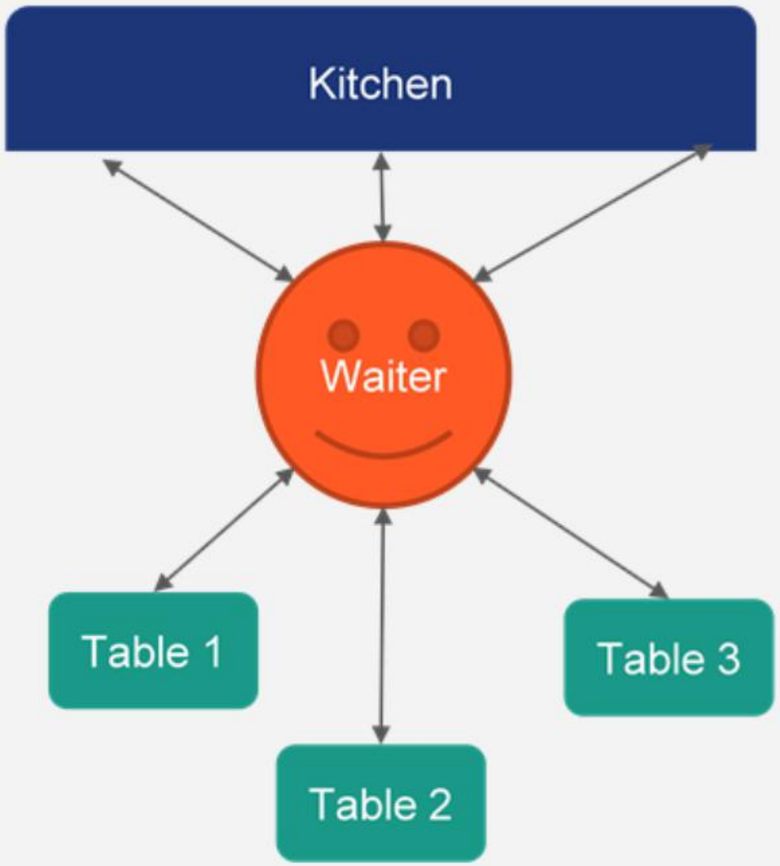
- Restaurant operations follow a loosely coupled architecture model.
- In this model, no two entities are heavily dependent on each other.
- The kitchen is loosely coupled with the waiter and the waiter is loosely coupled with the customer.
- The waiter connects with the customer only when he must take and serve orders.
- The waiter connects with the kitchen to forward customer's order and collect food when it is ready.
- Rest of the time, he can continue attending to other customers.
- Such flexibility brings out the best productivity.

Slide Note

Benefits of Loosely Coupled Architecture

- The key benefit of loosely coupled architecture is to provide an asynchronous execution environment.
- In this architectural model, a big process is broken down into chunks of small tasks that can process independently and can notify the user when the task completes.
- When a task is not completed, instead of waiting, another task can be initiated.
- As the previously initiated task completes, the response is captured.
- In case of the restaurant, while the order is prepared for one table, the waiter can attend to other customers.
- As the kitchen notifies the waiter, the waiter collects the order and serves it to the respective customer.

Waiter Handling Orders From Customers - Asynchronously



Slide Note

A Few More Examples of Asynchronous Execution

- Preparing breakfast:
  - While the coffee is getting ready, make toast.
  - While the toast is getting ready, cut the fruit.
  - Once the coffee is ready, pour it in mugs.
  - Once the toast is ready, apply butter and keep them in serving tray.
  - By now, all items are ready to serve and eat.
- Home renovation:
  - Masonry work can get started while the furniture is being made.
  - While masonry is done, electrical cables and fixtures can be installed.
  - As the furniture gets ready, whitewashing can be done.
  - Finally, the renovated house is ready.

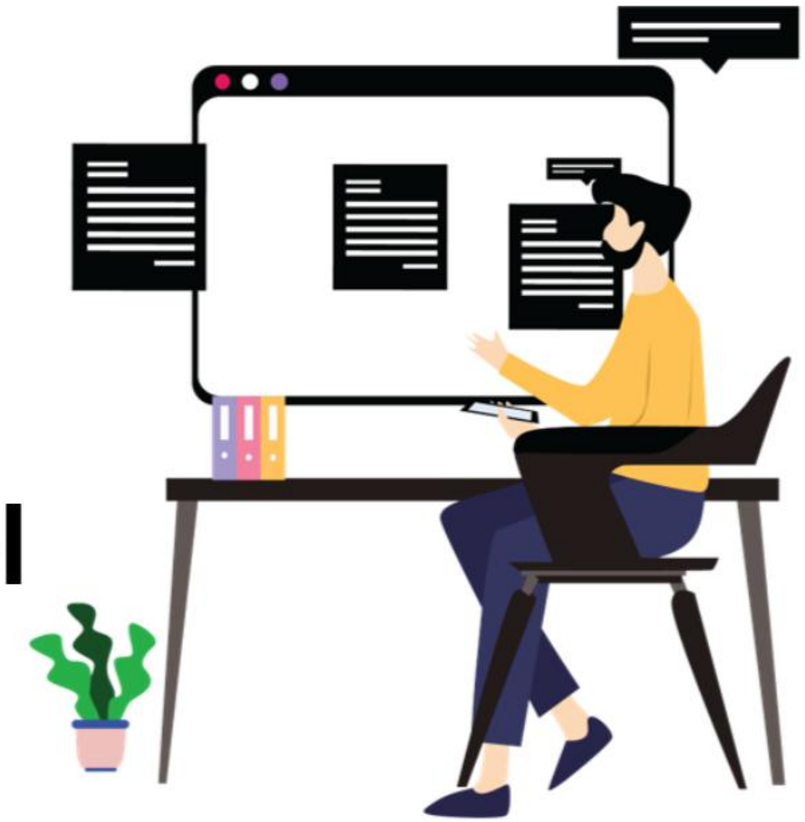
Slide Note

Convert sync to async for L.C.A

Would you want your websites to perform tasks synchronously and cause long delays or perform tasks asynchronously for a better user experience?

Slide Note

Interact with Application Servers Using HTTP Protocol



Slide Note

Learning Objectives

- Explain synchronous execution
- Explore the limitations of synchronous execution
- Explain asynchronous execution
- Implement asynchronous execution using Callbacks
- Implement asynchronous execution using Promise object
- Perform server operations using Axios API
- Create a promise chain

Slide Note

Withdrawing Money From an ATM – How Does ATM Operate?

- Read ATM card
- Request and read ATM PIN number
- Send the details to server for verification
- Wait until the response is received
- If successfully verified, read the amount to withdraw
- Check the balance in account
- If the account has sufficient balance, dispense cash

Slide Note

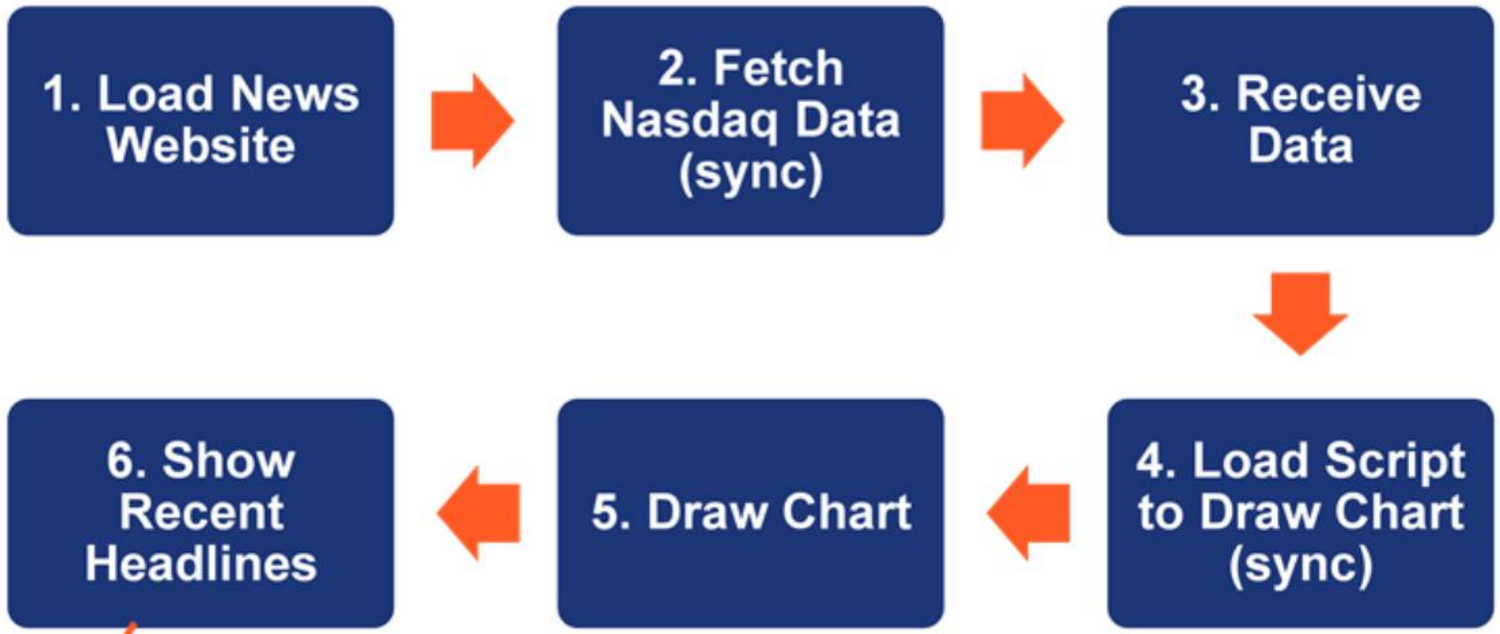
What was the mode of execution?

Slide Note

Synchronous and Sequential

Slide Note

What If the News on a Website Were Fetched Synchronously?

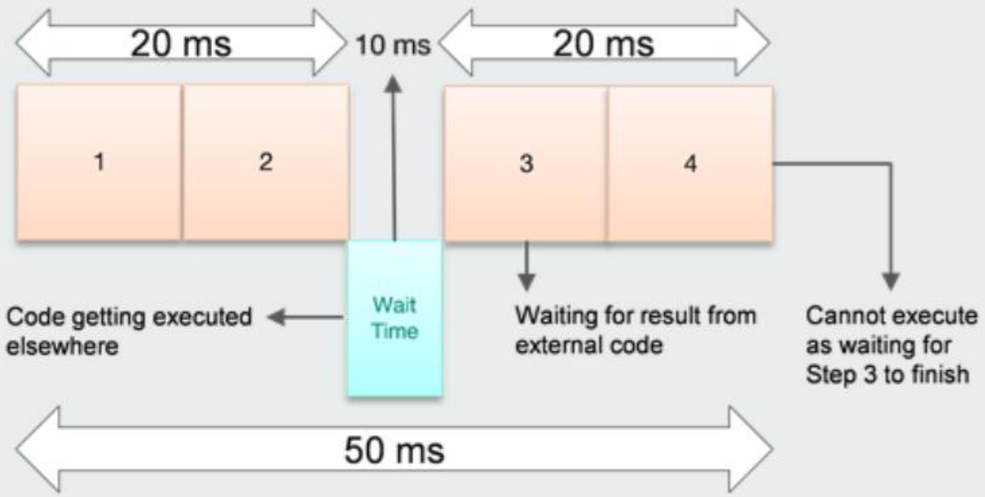


Must wait till the Nasdaq data is not fetched and chart is not drawn.

Fetching news should be of higher priority, it interests the viewers more, but is delayed due to fetching data for Sensex chart.

You can check the live demo of Nasdaq site **here**, that shows the trending market news as well as displays the Nasdaq index chart.

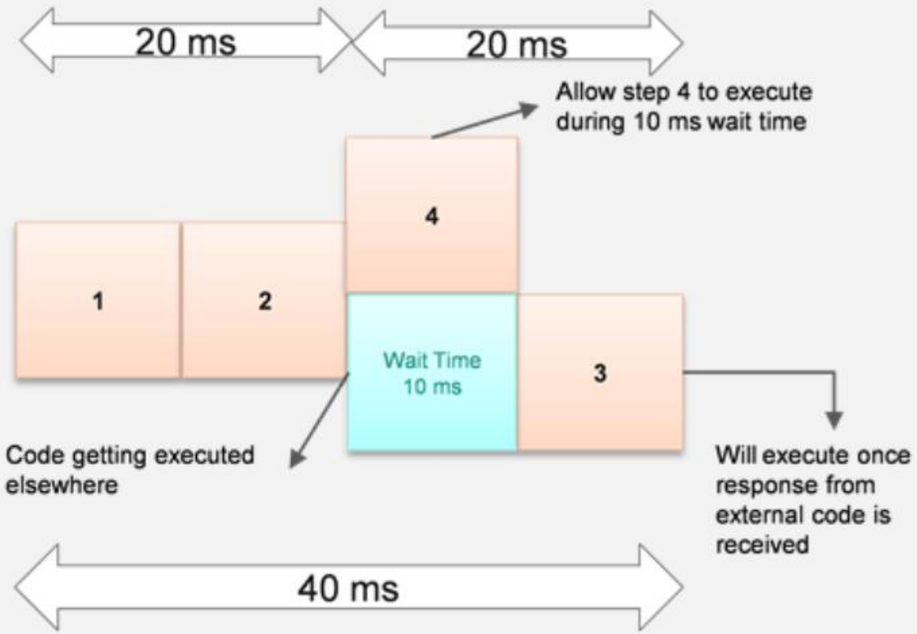
Slide Note



When Programs Execute Synchronously

- Calls to an external code can result in a delayed response.
- This can block subsequent execution.
- It can cause ‘freezing’ on the screen.
- This can lead to an unresponsive web page and result in poor user engagement.

Slide Note

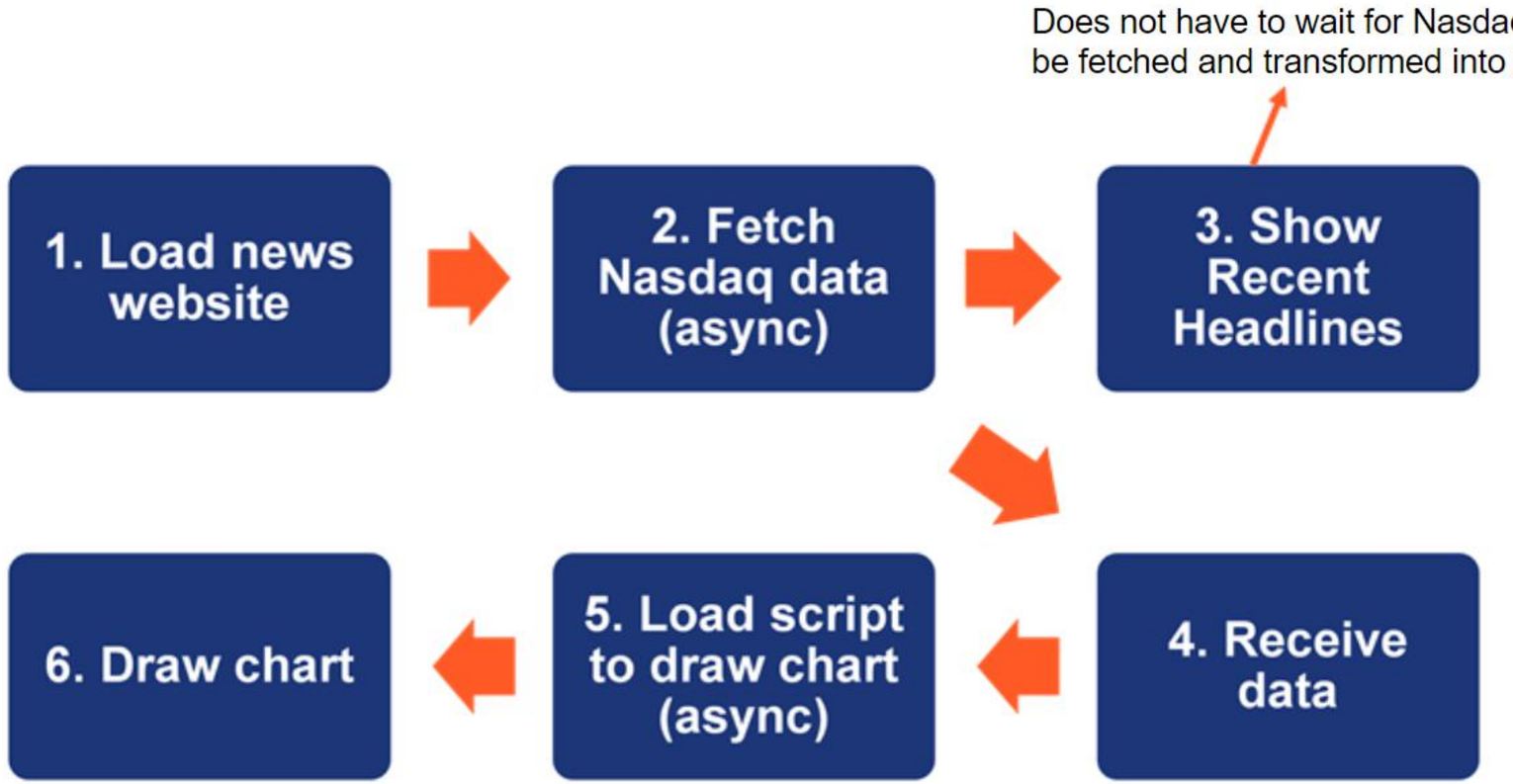


What Would Be a Better Approach?

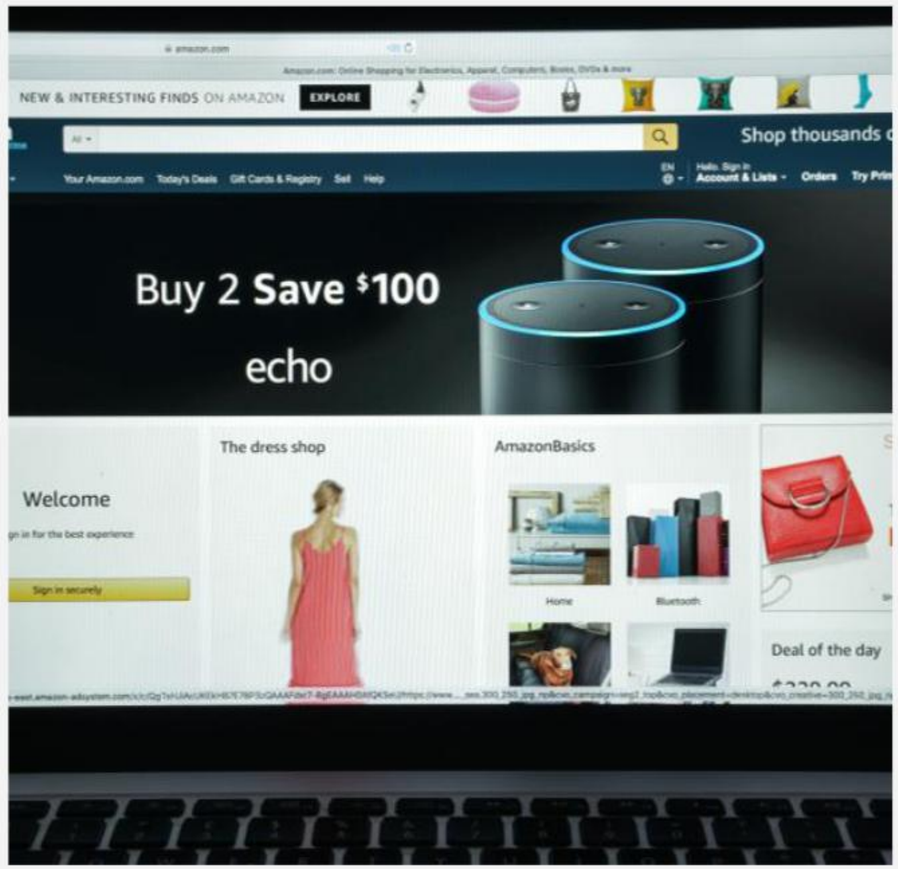
- JavaScript allows asynchronous programming.
- With this approach, wait time is utilized to proceed with subsequent instructions to execute.
- It helps provide non-blocking execution, reducing latency.
- This helps improve user engagement.

Slide Note

Fetching News Asynchronously



Slide Note



Fetching Cart Items Count Asynchronously

- In an online shopping app, an asynchronous mode of execution helps provide good user experience.
- Once the users log in, they would be:
  - Browsing products to shop.
  - Seeing count of current products displayed in cart.
- The cart data may reside on a different server, requiring the main server to hop on to another server to fetch the cart details.
- Hence, while the cart details are being fetched, the user should not be held up, preventing him from browsing products.
- The users can browse the products, asynchronously cart details are fetched, and the count is displayed.

Slide Note

Mailer App Reads Mail Asynchronously

- Imagine if, while your inbox is populated with mails from server, you must wait.
- The app does not allow you to compose a new email until all the emails are received.
- This will lead to a bad user experience.
- Mail from the server should be read asynchronously without blocking the user from composing new emails.
- The inbox keeps notifying the user as and when a new email has arrived.

Slide Note

Timer Events Execute Asynchronously

- Let’s say, in an online quiz app, the web page should display the number of seconds that has elapsed.
- In the background, the timer is set up with a defined interval of 1 second.
- After 1 second, the page should be displayed with a time-elapsed message.
- However, while the timer runs in the background, the user should be able to attempt the quiz questions parallelly.

Slide Note

```
function process(task) {
    let result = task(4, 10);
    console.log(result);
}


process((x, y) => x + y);

process((x, y) => x * y);
```

Callback function

Implement Asynchronous Programming – Approach 1

- Using callbacks:
  - Callbacks are functions passed as parameters to another function.
  - When the function with callback parameter is called, it calls the callback function.
  - This helps the developer to make a function execute a different task every time it is called.

Slide Note

Asynchronous Scenario – Timing Events

- In JavaScript, two popular functions help to set up timers.
- These timers work in the background asynchronously, without blocking other tasks that the user would like to perform.
- These timer functions take a callback function as the parameter that executes when the timer times out or the timer interval expires.
- These functions are:
  - `setTimeout(callback, interval)`
    - Executes a function after waiting for a specified interval (in milliseconds).
  - `setInterval(callback, interval)`
    - Same as setTimeout() but repeats the execution of the function continuously.

Slide Note

Display Time Elapsed Every 1 Second

Using callbacks and `setTimeout()` function makes an asynchronous request to display the time elapsed every 1 second.

The time elapsed can be displayed on the browser.

Click here for the demo solution.

DEMO

Slide Note

For a Series of Timeouts…

- Call the `setTimeout()` that expires after 1 second and displays 1 sec lapsed.
- After 1 second, again call `setTimeout()` that expires after 1 second and displays 1 more sec lapsed.
- Repeat the step one more time.

```
setTimeout(() => {
    console.log("1 sec lapsed")
}, interval)
```

```
setTimeout(()=>{
    console.log(`1 sec lapsed`);
    setTimeout(()=>{
        console.log(`2 sec lapsed`);
    },1000)
},1000);
```

```
setTimeout(()=>{
    console.log(`1 sec lapsed`);
    setTimeout(()=>{
        console.log(`2 sec lapsed`);
        setTimeout(()=>{
            console.log(`3 sec lapsed`);
        },1000)
    },1000)
},1000);
```

Slide Note

What Would the Code Be After 5 Such Time-Out Calls?

- See the code shown here.
- It has created a callback hell.
- Callback hell is caused by coding with complex nested callbacks.
- Every callback takes an argument that is a result of the previous callbacks.
- This is also known as ‘Pyramid of Doom’ since the code structure looks like a pyramid, making it difficult to read and maintain.

```
setTimeout(()=>{
    console.log(`1 sec lapsed`);
    setTimeout(()=>{
        console.log(`2 sec lapsed`);
        setTimeout(()=>{
            console.log(`3 sec lapsed`);
            setTimeout(()=>{
                console.log(`4 sec lapsed`);
                setTimeout(()=>{
                    console.log(`5 sec lapsed`);
                    setTimeout(()=>{
                        console.log(`6 sec lapsed`);
                    },1000);
                },1000);
            },1000);
        },1000);
    },1000);
},1000);
```

Callback Hell

Slide Note

Take an example from daily life.

A kid tells his father I promise to get good results.

Father says if you get good results then you get a bicycle else your pocket money will be deducted.

So in this case there is no action that is happening in present.

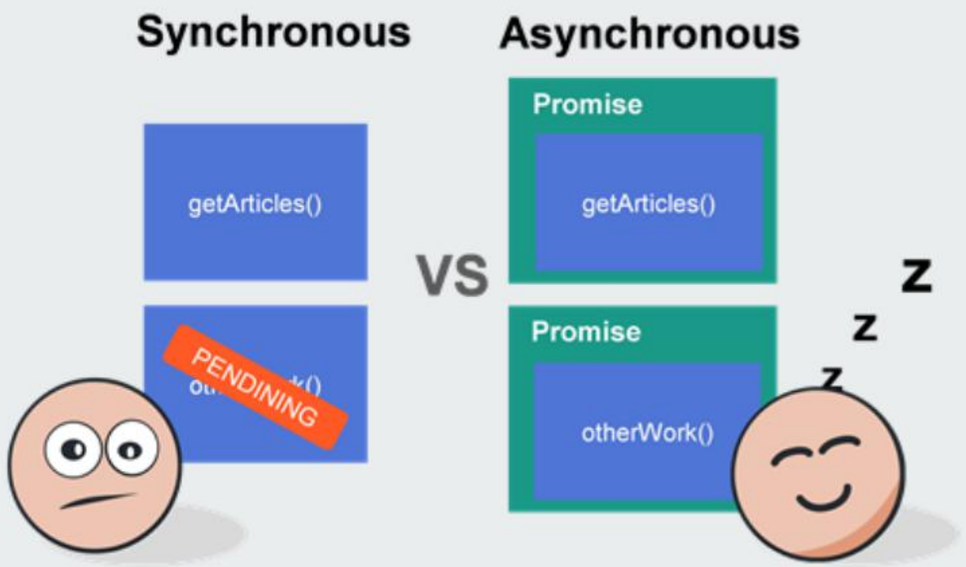
The discussion is all about some future event and what would be eventually done based on the outcome of that event.

In JavaScript, Promise is used with similar intent.

Definition of Promise says, when the action happens I shall either make success call or else failure call.

So it talks about some action that will happen in future.

This execution is also asynchronous in nature, as the execution of action may be delayed and hence there is a need to have non-blocking execution.



Implement Asynchronous Programming – Approach 2

- Promise, in JavaScript, is an object that represents an operation that will happen sometime in the future.
- A promise is an asynchronous action that may complete at some point and produce a value.

Slide Note

Display Time Elapsed Every One Second

Using Promise object and `setTimeout()` function, make an asynchronous request to display the time elapsed every second.

The time elapsed can be displayed on the browser.

Click here for the demo solution.

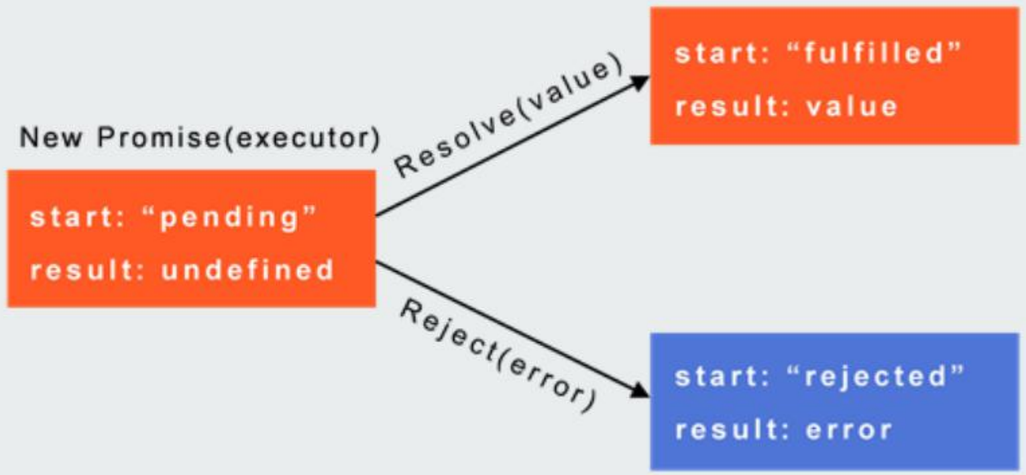
DEMO

Slide Note

So, What is Promise?

- In JavaScript **Promise** is an object that :
  - Takes executor function as a parameter to its constructor:
    - This function can be used to make asynchronous requests.
  - Allows handlers to be associated with promise object:
    - Handlers are functions that can execute once the request has been processed.
- Benefits of using Promise object:
  - No more Callback Hell
  - Better code readability
  - Efficient error handling

Slide Note



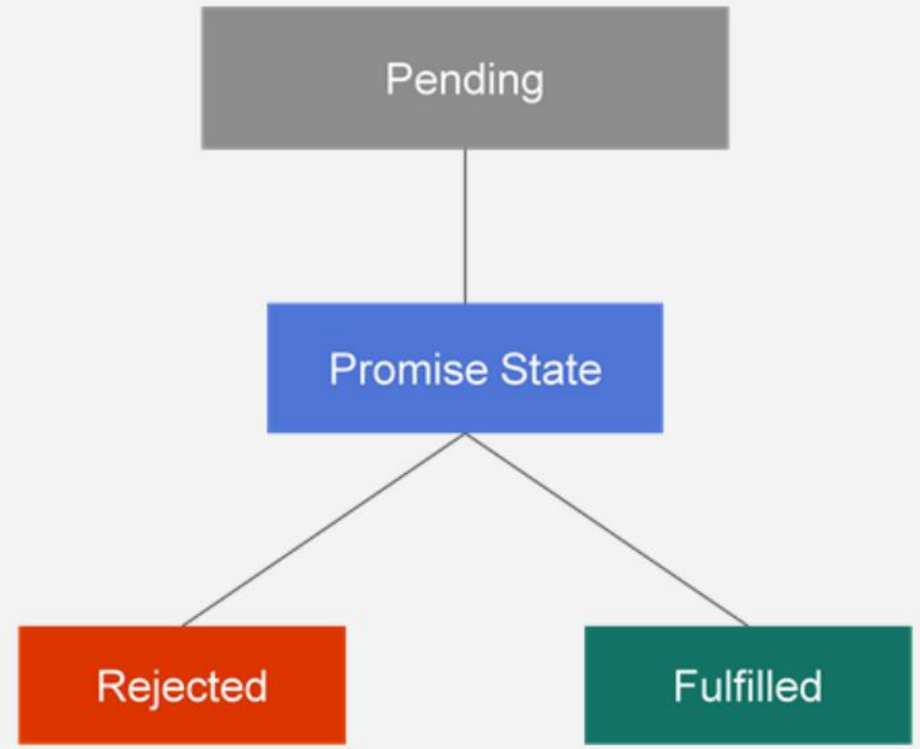
Creating Promise

- When created, the Promise object takes the `executor` function as a parameter for its constructor.
- The executor function takes two parameters, which are callbacks, popularly named `resolve` and `reject`.
- Resolve is called for successful fulfillment of promise.
- Reject is called for rejection of promise.
- The executor function can be used for making asynchronous requests.

Slide Note

Promise States

- The executor function of the Promise immediately executes.
- However, the asynchronous requests made from the executor function may not immediately execute.
- Until that time, Promise is in **Pending** state.
- If the request responds with success, Promise can call the resolve function (first parameter) of the executor function. This changes the state of the Promise to **Fulfilled**.
- However, if the request responds with error, Promise can call the reject function (second parameter) of the executor function. This changes the state of Promise to **Rejected**.
- The Promise once gets fulfilled (resolved) or rejected does not re-execute.



Slide Note

then()

Used to attach resolve handlers

Called when the action is successful.

Helps to retrieve result of Promise

catch()

Used to attach reject handlers

Called when the action has failed.

Helps to retrieve reason of rejection of Promise

Consuming Promise

- Consuming Promise means using the Promise object and attaching handler functions.
- Handlers are functions that get executed when the Promise is either fulfilled or rejected.
- To execute the handler that gets called when Promise is fulfilled, attach it by with the `then()` method of Promise.
- To execute the handler that gets called when Promise is rejected, attach it by with the `catch()` method of Promise.

Slide Note

Asynchronous Scenario – Client Server Communication

- In case of client server communication in a web application, the client makes HTTP requests to the server through REST API, and the server returns HTTP response to the client through the API.
- JavaScript provides an `XMLHttpRequest` object that

Provides methods to:

- create request (GET, POST, PUT, DELETE)
- send request

Fires event when

- request processing state changes
- request completes
- request completes with error

Provides properties to

- know the state of request
- get the response

Slide Note

Think and Tell

- **Too much work !!!**
- Would it not be helpful if there exists an API that encapsulates the tasks done by `XMLHttpRequest` object?
- That would allow the developers only to:
  - Make a request
  - Handle the response

Slide Note

Asynchronous HTTP Request Execution With Promises

- The `Axios` API is a Promise-based HTTP client for the browser and Node.js.
- It allows you to make HTTP requests and provides the property to capture responses.
- **`Axios`** helps to make API calls and returns promises.
- Promises help to asynchronously process external interactions.
- Axios provides methods that include `get()`, `post()`, `put()` and `delete()`.
- Successful call returns Promise with data.
- The failed call returns Promise with an error.
- To use `Axios`, in index.html or any .html file created for designing web page. include:

```
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
```

Slide Note

Get and Post Blog Posts

Locally, `json-server` is running to serve blog posts data.

Using axios fetch the blog posts and display the same.

Also, add new post and send it to the server for storing it.

Click here for the demo solution.

DEMO

Slide Note

Concluding thoughts

According to you, Where async helps?

Quick Check

Asynchronous request is made in the below code to fetch data of users?

What would be the state of Promise?

```
let p = axios.get('http://localhost:3000/contacts')

console.log(p);
```

Slide Note

Concluding thoughts

According to you, Where async helps?

Quick Check: Solution

Asynchronous request is made in the below code to fetch data of users?

What would be the state of Promise?

```
let p = axios.get('http://localhost:3000/contacts');

console.log(p);
```

Correct Answer: Promise is not consumed and hence the state of Promise would be *pending*.

Slide Note

Online Shopping – Fetching Order With Customer Details

```
{
  "orders":[{
    "id": "9876867986",
    "orderDate":"22-Jul-2022",
    "customerId": "316058a",
    "products":[
      {
        "productName":"mouse",
        "quantity":2,
        "price":400
      }
    ]
  },
  {
    "id": "98768553386",
    "orderDate":"22-Jul-2022",
    "customerId": "316058a",
    "products":[
      {
        "productName":"ssd",
        "quantity":2,
        "price":400
      }
    ]
  }]
}
```

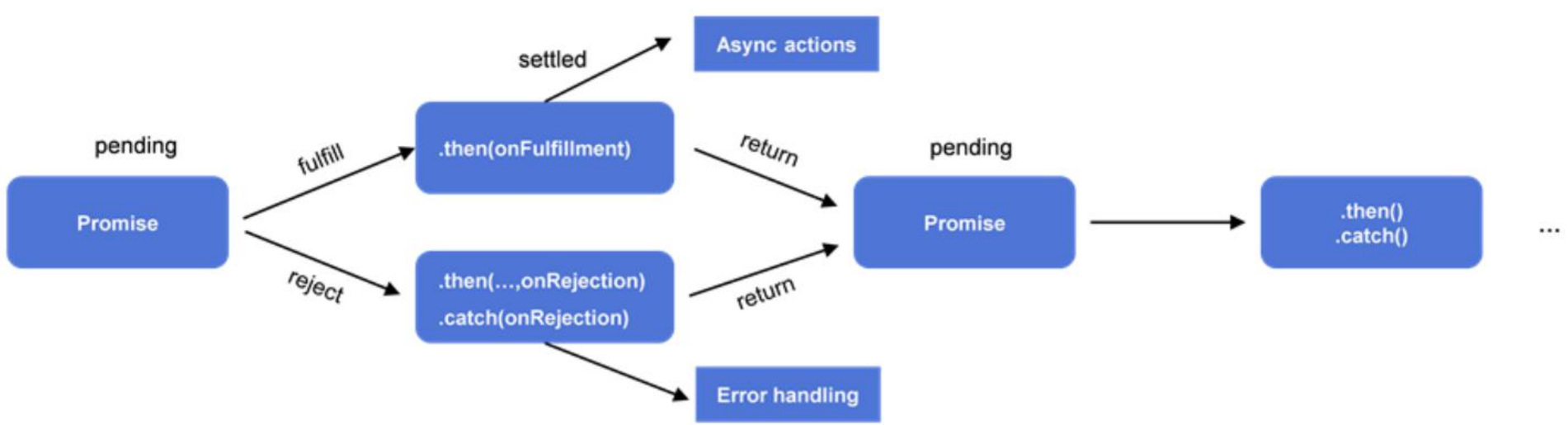
```
"customers":[
    {
      "id": "316058a",
      "name": "Brenda Watkins",
      "gender": "female",
      "email":
             "brendawatkins@visualix.com"
,
      "phone": "+1 (985) 506-2849",
      "address": "466 Bergen Street,
             Bagtown, Kentucky, 6339"
    },
    {
      "id": "316058b",
      "name": "Finch Marsh",
      "gender": "male",
      "email": "finchmarsh@visualix.com",
      "phone": "+1 (868) 532-2477",
      "address": "451 Wythe Place,
             Hollymead, North Carolina,
             1179"
    }
]
```

- Let's say, we have two JSON data – Orders and Customers
- The Orders JSON stores only customer id with other order details.
- The complete customer details are available in Customers JSON,
- **How can we fetch customer details along with the order requested?**

Slide Note

Create a Promise Chain

- `Promise.prototype.then()` and `Promise.prototype.catch()` methods return Promises.
- Thus, they can be chained.
- This means the promise returned by them can further be consumed to attach next set of handlers.
- Chaining is useful, when resultant from Promise is further used to make next asynchronous request.



Slide Note

Fetch Order with Customer Details

Let's say we have two JSON data – Orders and Customers.

The Order JSON stores only customer id with other order details.

The complete customer details are available in Customers JSON.

By chaining promises, fetch customer details along with the order requested.

Click here for the demo solution.

DEMO