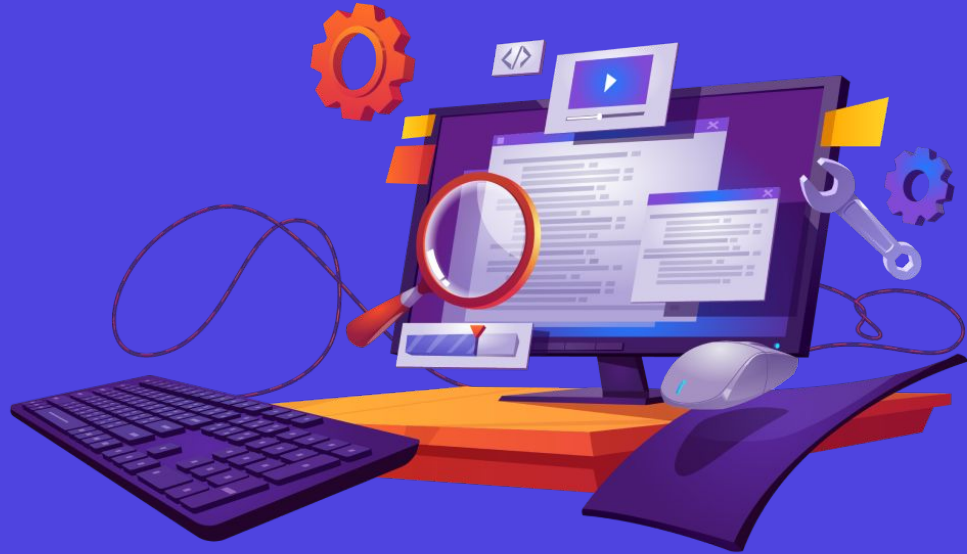


Data Retrieval

Relevel
by Unacademy



List of topic content:

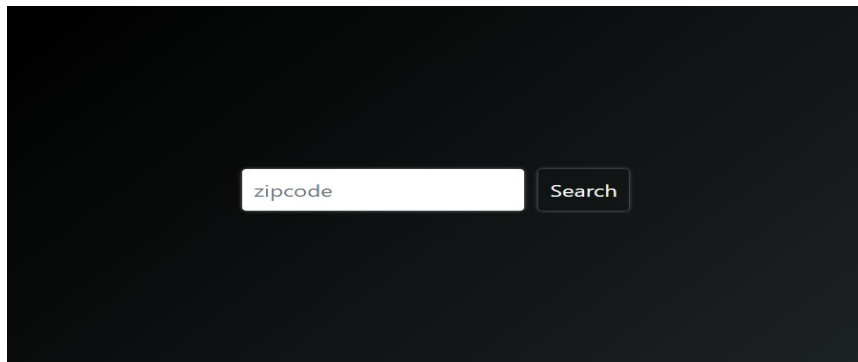
- What is Asynchronous JavaScript?
- How Callbacks Work in JavaScript
- How Promises Work in JavaScript
- How Async / Await Works in JavaScript

We are going to build and run an Ice Cream Shop and learn Asynchronous JavaScript at the same time and learn how to use Callbacks, Promises, Async/Await through this application

App Introduction

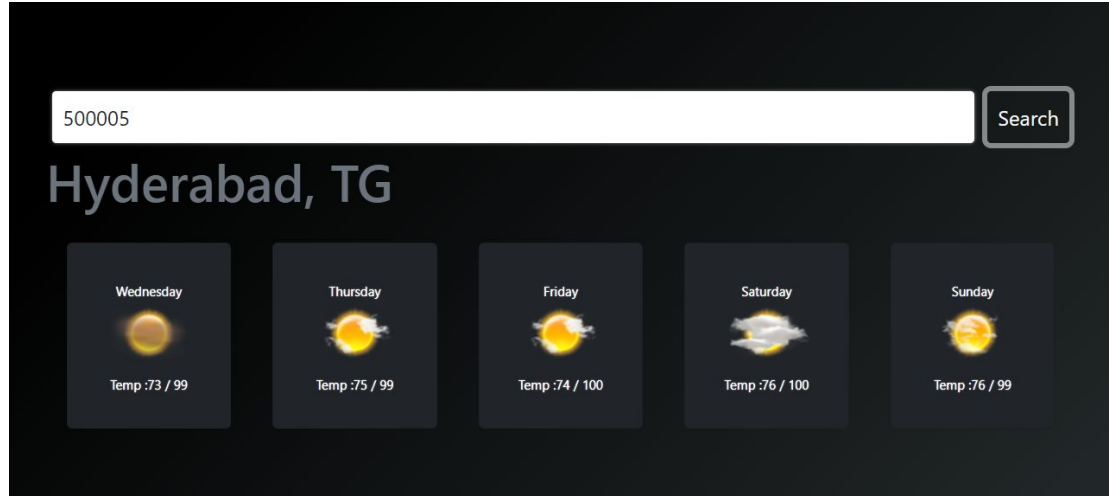
We will build a **Weather application**:

We will be building a simple Weather app that will show weather of a specific area from zipcode as shown below



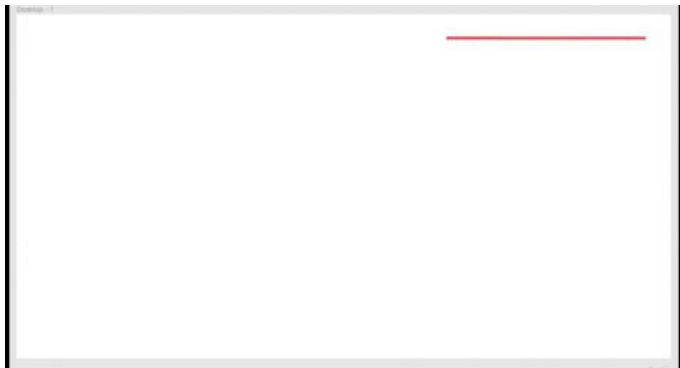
The image shows a dark-themed user interface for a weather application. In the center, there is a white rectangular input field with the placeholder text "zipcode". To the right of this field is a button labeled "Search".

Weather Application



What is an Asynchronous programming?

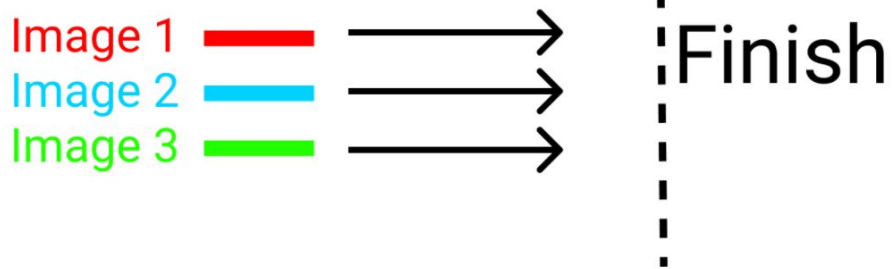
Here tasks are completed independently. Imagine that for 10 tasks, you have 10 hands. So, each hand can do each task independently and at the same time. Take a look at the GIF below – you can see that each image loads at the same time.



Again, all the images are loading at their own pace. None of them is waiting for any of the others.

What is an Asynchronous programming?

Asynchronous



What is an Asynchronous programming?

Let's see Asynchronous code example:

Let's say it takes two seconds to eat some ice cream. Now, let's test out an asynchronous system. Write the below code in JavaScript.

```
console.log("I");  
  
// This will be shown after 2 seconds  
  
setTimeout(()=>{  
  console.log("eat");  
},2000)  
  
console.log("Ice Cream")
```

What is an Asynchronous programming?

Here's the result in the console:

```
Console
"I"
"Ice Cream"
"eat"
```


What are callbacks in JavaScript?

When you nest a function inside another function as an argument, that's called a callback:

Callback illustrated

```
Function One (){  
    // Do something  
}
```

```
Function Two (call_One){  
    // Do something else  
    call_One()  
}
```

```
Two(One);
```



code is being executed

An example of a callback

Callbacks:

Why do we use callbacks?

When doing a complex task, we break that task down into smaller steps. To help us establish a relationship between these steps according to time (optional) and order, we use callbacks.

Take a look at this example:

Steps to make Ice Cream

1. Place order
2. Cut the fruit
3. Add water and ice
4. Start the machine
5. Select container
6. Select toppings
7. Serve Ice Cream

These are the small steps you need to take to make ice cream. Also note that in this example, the order of the steps and timing are crucial. You can't just chop the fruit and serve ice cream.

At the same time, if a previous step is not completed, we can't move on to the next step.

To explain that in more detail, let's start our ice cream shop business.

The shop will have two parts:

- The storeroom will have all the ingredients [Our Backend]
- We'll produce ice cream in our kitchen [The frontend]

Let's store our data

Now, we're gonna store our ingredients (the fruits) inside an object. Let's start!

The Fruits



Strawberry



Grapes



Banana

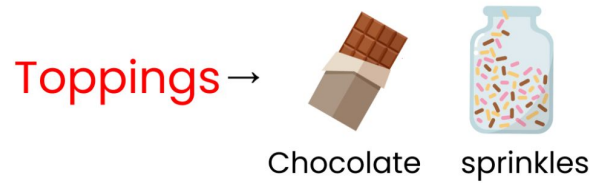


Apple

you can store the ingredients inside objects like this:

```
let stocks = {  
  Fruits : ["strawberry", "grapes",  
    "banana", "apple"]
```

Our other ingredients are here:



You can store these other ingredients in JavaScript objects like this:

```
let stocks = {  
  Fruits : ["strawberry", "grapes", "banana", "apple"],  
  liquid : ["water", "ice"],  
  holder : ["cone", "cup", "stick"],  
  toppings : ["chocolate", "peanuts"],  
};
```

The entire business depends on what a customer orders.
Once we have an order, we start production and then we
serve ice cream. So, we'll create two functions

- order
- production

This is how it all works: Order from customer -> fetch ingredients -> start production -> serve
Get order from customer, fetch ingredients, start production, then serve.
Let's make our functions. We'll use arrow functions here:

```
let order = () =>{};

let production = () =>{};
```

Now, let's establish a relationship between these two functions using a callback, like this:

```
let order = (call_production) =>{

  call_production();
};

let production = () =>{};
```

Let's do a small test : We'll use the `console.log()` function to conduct tests to clear up any doubts we might have regarding how we established the relationship between the two functions.

```
let order = (call_production) =>{  
  
  console.log("Order placed. Please call production")  
  
  // function 📌 is being called  
  call_production();  
};  
  
let production = () =>{  
  
  console.log("Production has started")  
  
};
```

To run the test, we'll call the order function. And we'll add the second function named production as its argument.


```
// name 📌 of our second function  
order(production);
```

Here's the result in our console

```
Console Clear ×  
"Order placed. Please call production"  
"Production has started"
```

Clear out the console.log: Keep this code and remove everything [don't delete our stocks variable]. On our first function, pass another argument so that we can receive the order [Fruit name]:

```
// Function 1

let order = (fruit_name, call_production) =>{

  call_production();
};

// Function 2

let production = () =>{};

// Trigger 📌

order("", production);
```

Here are our steps, and the time each step will take to execute.
Steps to make Ice Cream

Time(seconds)

- #1 Place Order → 2
- #2 Cut The Fruit → 2
- #3 Add water and ice → 1
- #4 Start the machine → 1
- #5 Select Container → 2
- #6 Select Toppings → 3
- #7 Serve Ice Cream → 2

Above, you can see that step 1 is to place the order, which takes 2 seconds. Then step 2 is cut the fruit (2 seconds), step 3 is add water and ice (1 second), step 4 is to start the machine (1 second), step 5 is to select the container (2 seconds), step 6 is to select the toppings (3 seconds) and step 7, the final step, is to serve the ice cream which takes 2 seconds.

To establish the timing, the function `setTimeout()` is excellent as it is also uses a callback by taking a function as an argument.

`setTimeout()` Syntax

`setTimeout (()=>{} , 1000)`

setting Time
1000 millisecond =
1 second

calling a function[`callback`]

Syntax of a `setTimeout()` function

Now, let's select our fruit and use this function:

```
// 1st Function

let order = (fruit_name, call_production) =>{

  setTimeout(function(){

    console.log(`${stocks.Fruits[fruit_name]} was selected`)

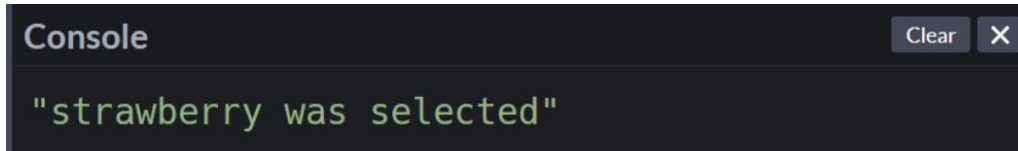
    // Order placed. Call production to start
    call_production();
  },2000)
};

// 2nd Function

let production = () =>{
  // blank for now
};

// Trigger 🖱️
order(0, production);
```

And here's the result in the console: Note that the result is displayed after 2 seconds.



```
Console
```

```
"strawberry was selected"
```

If you're wondering how we picked strawberry from our stock variable, here's the code with the format

Format -> `variable.object[array]`

Our code -> `stocks.Fruits[0]`

Don't delete anything. Now we'll start writing our production function with the following code. We'll use arrow functions:

```
let production = () =>{  
  
  setTimeout(()=>{  
    console.log("production has started")  
  },0000)  
  
};
```

And the result:

Console

"strawberry was selected"

"production has started"

We'll nest another setTimeout function in our existing setTimeout function to chop the fruit. Like this:

```
let production = () =>{  
  
  setTimeout(()=>{  
    console.log("production has started")  
  
    setTimeout(()=>{  
      console.log("The fruit has been chopped")  
    },2000)  
  
  },0000)  
};
```

And the result:

Console

"strawberry was selected"

"production has started"

"The fruit has been chopped"

If you remember, here are our steps:

Time(seconds)

- #1 Place Order → 2
- #2 Cut The Fruit → 2
- #3 Add water and ice → 1
- #4 Start the machine → 1
- #5 Select Container → 2
- #6 Select Toppings → 3
- #7 Serve Ice Cream → 2

Let's complete our ice cream production by nesting a function inside another function – this is also known as a callback, remember?

```

let production = () =>{

  setTimeout(()=>{
    console.log("production has started")
    setTimeout(()=>{
      console.log("The fruit has been chopped")
      setTimeout(()=>{
        console.log(`${stocks.liquid[0]} and ${stocks.liquid[1]} Added`)
        setTimeout(()=>{
          console.log("start the machine")
          setTimeout(()=>{
            console.log(`Ice cream placed on ${stocks.holder[1]}`)
            setTimeout(()=>{
              console.log(`${stocks.toppings[0]} as toppings`)
              setTimeout(()=>{
                console.log("serve Ice cream")
              },2000)
            },3000)
          },2000)
        },1000)
      },1000)
    },2000)
  },0000)
};

```

And here's the result in the console

Console

"strawberry was selected"

"production has started"

"The fruit has been chopped"

"water and ice Added"

"start the machine"

"Ice cream placed on cup"

"chocolate as toppings"

"serve Ice cream"

Code link: <https://jsfiddle.net/u9naf3hk/9/>

This is called callback hell. It looks something like this (remember that code just above?):

Callback Hell



Illustration of Callback hell

What's the solution to this?

How to Use Promises to Escape Callback Hell

Promises were invented to solve the problem of callback hell and to better handle our tasks. This is how promise looks:

Promises **Format**



illustration of a promise format

- **Pending:** This is the initial stage. Nothing happens here. Think of it like this, your customer is taking their time giving you an order. But they haven't ordered anything yet.
- **Resolved:** This means that your customer has received their food and is happy.
- **Rejected:** This means that your customer didn't receive their order and left the restaurant.

Let's adopt promises to our ice cream production case study.

We need to understand four more things first ->

- Relationship between time and work
- Promise chaining
- Error handling
- The .finally handler

Let's start our ice cream shop and understand each of these concepts one by one by taking baby steps.

Relationship between time and work

If you remember, these are our steps and the time each takes to make ice cream"

Time(seconds)

- #1 Place Order → 2
- #2 Cut The Fruit → 2
- #3 Add water and ice → 1
- #4 Start the machine → 1
- #5 Select Container → 2
- #6 Select Toppings → 3
- #7 Serve Ice Cream → 2

let's create a variable in JavaScript

```
let is_shop_open = true;
```

Now create a function named order and pass two arguments named time, work:

```
let order = ( time, work ) =>{  
  
}
```

Now, we're gonna make a promise to our customer, "We will serve you ice-cream" Like this ->


```
let order = ( time, work ) =>{  
  
  return new Promise( ( resolve, reject )=>{ } )  
  
}
```

Our promise has 2 parts:

- Resolved [ice cream delivered]
- Rejected [customer didn't get ice cream]

```
let order = ( time, work ) => {  
  
  return new Promise( ( resolve, reject )=>{  
  
    if( is_shop_open ){  
  
      resolve( )  
  
    }  
  
    else{  
  
      reject( console.log("Our shop is closed") )  
  
    }  
  
  })  
}
```

Let's add the time and work factors inside our promise using a `setTimeout()` function inside our if statement.

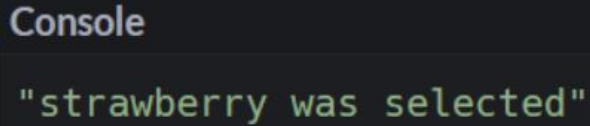
Note: In real life, you can avoid the time factor as well. This is completely dependent on the nature of your work.

```
let order = ( time, work ) => {  
  
  return new Promise( ( resolve, reject )=>{  
  
    if( is_shop_open ){  
  
      setTimeout(()=>{  
  
        // work is 🕒 getting done here  
        resolve( work() )  
  
        // Setting 🕒 time here for 1 work  
      }, time)  
  
    }  
  
    else{  
      reject( console.log("Our shop is closed") )  
    }  
  
  })  
}
```

Now, we're gonna use our newly created function to start ice-cream production.

```
// Set 🕒 time here  
order( 2000, ()=>console.log(`${stocks.Fruits[0]} was selected`))  
//   pass a 🙌 function here to start working
```

The result after 2 seconds looks like this:



Console

```
"strawberry was selected"
```

Promise chaining

In this method, we defining what we need to do when the first task is complete using the `.then` handler. It looks something like this

Promise

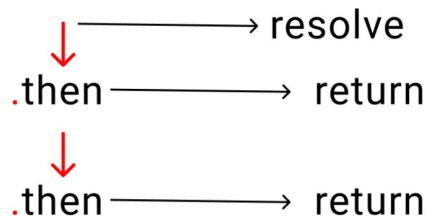


Illustration of promise chaining using `.then` handler

The `.then` handler returns a promise when our original promise is resolved.

Let me make it simpler: it's similar to giving instructions to someone. You tell someone to " First do this, then do that, then this other thing, then..., then..., then..." and so on.

- The first task is our original promise.
- The rest of the tasks return our promise once one small bit of work is completed

Let's implement this on our project. At the bottom of your code write the following lines.

Note: don't forget to write the return word inside your .then handler. Otherwise, it won't work properly. If you're curious, try removing the return once we finish the steps:

```
order(2000,()=>console.log(`${stocks.Fruits[0]} was selected`))

.then(()=>{
  return order(0000,()=>console.log('production has started'))
})
```

And the result:

Console

"strawberry was selected"

"production has started"

Using the same system, let's finish our project:

```
// step 1
order(2000, ()=>console.log(` ${stocks.Fruits[0]} was selected`))

// step 2
.then(()=>{
  return order(0000, ()=>console.log('production has started'))
})

// step 3
.then(()=>{
  return order(2000, ()=>console.log("Fruit has been chopped"))
})

// step 4
.then(()=>{
  return order(1000, ()=>console.log(` ${stocks.liquid[0]} and ${stocks.liquid[1]} added`))
})

// step 5
.then(()=>{
  return order(1000, ()=>console.log("start the machine"))
})

// step 6
.then(()=>{
  return order(2000, ()=>console.log(`ice cream placed on ${stocks.holder[1]}`))
})

// step 7
.then(()=>{
  return order(3000, ()=>console.log(` ${stocks.toppings[0]} as toppings`))
})

// Step 8
.then(()=>{
  return order(2000, ()=>console.log("Serve Ice Cream"))
})
```


Here's the result

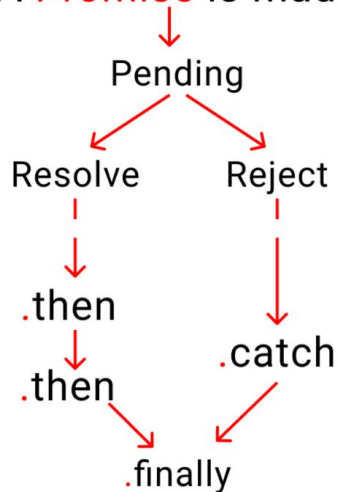
```
Console
"strawberry was selected"
"production has started"
"Fruit has been chopped"
"water and ice added"
"start the machine"
"ice cream placed on cup"
"chocolate as toppings"
"Serve Ice Cream"
```

Code link: <https://jsfiddle.net/am0yekuL/3/> :

Error Handling:

We need a way to handle errors when something goes wrong. But first, we need to understand the promise cycle:

A **Promise** is made



An illustration of the life of a promise

To catch our errors, let's change our variable to false

```
let is_shop_open = false;
```

Which means our shop is closed. We're not selling ice cream to our customers anymore.

To handle this, we use the `.catch` handler. Just like `.then`, it also returns a promise, but only when our original promise is rejected.

A small reminder here:

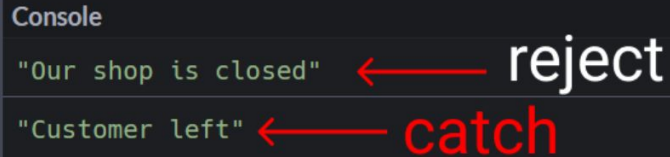
- `.then` works when a promise is resolved
- `.catch` works when a promise is rejected

Go down to the very bottom and write the following code:

Just remember that there should be nothing between your previous .then handler and the .catch handler.

```
.catch(()=>{  
  console.log("Customer left")  
})
```

Here's the result:



Console

"Our shop is closed" ← reject

"Customer left" ← catch

A couple things to note about this code:

- The 1st message is coming from the `reject()` part of our promise
- The 2nd message is coming from the `.catch` handler

How to use the `.finally()` handler

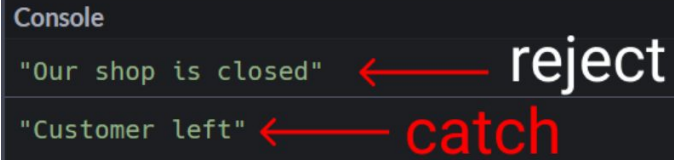
There's something called the finally handler which works regardless of whether our promise was resolved or rejected.

For example: whether we serve no customers or 100 customers, our shop will close at the end of the day

To test this,

```
.catch(()=>{  
  console.log("Customer left")  
}))
```

Result:



Console

"Our shop is closed" ← reject

"Customer left" ← catch

Code link: <https://jsfiddle.net/aghq96jv/2/>

Everyone, please welcome Async / Await~

How Does Async / Await Work in JavaScript?

This is supposed to be the better way to write promises and it helps us keep our code simple and clean.

All you have to do is write the word `async` before any regular function and it becomes a promise.

Promises vs Async/Await in JavaScript

Before `async/await`, to make a promise we wrote this:

```
function order(){  
  return new Promise( (resolve, reject) =>{  
  
    // Write code here  
  } )  
}
```

Now using async/await, we write one like this

```
//👉 the magical keyword
async function order() {
  // Write code here
}
```

You need to understand ->

- How to use the try and catch keywords
- How to use the await keyword

How to use the Try and Catch keywords

We use the try keyword to run our code while we use catch to catch our errors. It's the same concept we saw when we looked at promises.:

Let's see a comparison. We'll see a small demo of the format, then we'll start coding.

Promises in JS -> resolve or reject

We used resolve and reject in promises like this:

```
function kitchen(){

  return new Promise ((resolve, reject)=>{
    if(true){
      resolve("promise is fulfilled")
    }

    else{
      reject("error caught here")
    }
  })
}

kitchen() // run the code
.then()   // next step
.then()   // next step
.catch()  // error caught here
.finally() // end of the promise [optional]
```

Async / Await in JS -> try, catch

When we're using async/await, we use this format:

```
//👉 Magical keyword
async function kitchen(){

  try{
    // Let's create a fake problem
    await abc;
  }

  catch(error){
    console.log("abc does not exist", error)
  }

  finally{
    console.log("Runs code anyways")
  }
}

kitchen() // run the code
```

Now hopefully you understand the difference between promises and Async / Await.

How to Use JavaScript's Await Keyword

The keyword await makes JavaScript wait until a promise settles and returns its result.

How to use the await keyword in JavaScript

Let's go back to our ice cream shop. We don't know which topping a customer might prefer, chocolate or peanuts. So we need to stop our machine and go and ask our customer what they'd like on their ice cream.

Notice here that only our kitchen is stopped, but our staff outside the kitchen will still do things like:

- doing the dishes
- cleaning the tables
- taking orders, and so on.

An Await Keyword Code Example

Let's create a small promise to ask which topping to use. The process takes three seconds.

```
function toppings_choice (){  
  return new Promise((resolve, reject)=>{  
    setTimeout(()=>{  
  
      resolve( console.log("which topping would you love?" ) )  
  
    }, 3000)  
  })  
}
```

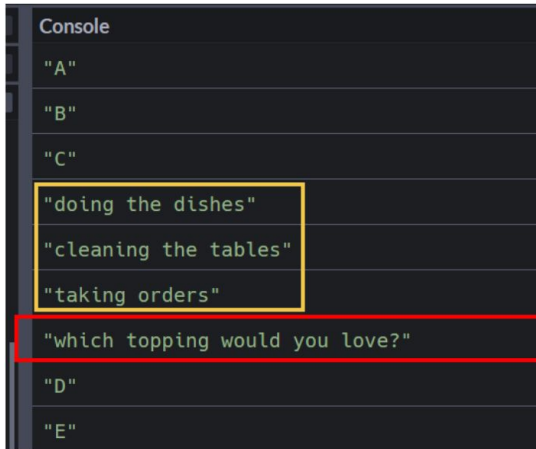
Now, let's create our kitchen function with the async keyword first.

```
async function kitchen(){  
  
  console.log("A")  
  console.log("B")  
  console.log("C")  
  
  await toppings_choice()  
  
  console.log("D")  
  console.log("E")  
  
}  
  
// Trigger the function  
  
kitchen();
```

Let's add other tasks below the kitchen() call.

```
console.log("doing the dishes")  
console.log("cleaning the tables")  
console.log("taking orders")
```

And here's the result:



We are literally going outside our kitchen to ask our customer, "what is your topping choice?" In the mean time, other things still get done.

Once, we get their topping choice, we enter the kitchen and finish the job.

Small note

When using Async/ Await, you can also use the .then, .catch, and .finally handlers as well which are a core part of promises.

Let's open our Ice cream shop again

We're gonna create two functions ->

- kitchen: to make ice cream
- time: to assign the amount of time each small task will take.

Let's start! First, create the time function:

```
let is_shop_open = true;

function time(ms) {

  return new Promise( (resolve, reject) => {

    if(is_shop_open){
      setTimeout(resolve,ms);
    }

    else{
      reject(console.log("Shop is closed"))
    }
  });
}
```


Now, let's create our kitchen:

```
async function kitchen(){  
  try{  
  
    // instruction here  
  }  
  
  catch(error){  
    // error management here  
  }  
}  
  
// Trigger  
kitchen();
```

Let's give small instructions and test if our kitchen function is working or not:

```
async function kitchen(){
  try{

    // time taken to perform this 1 task
    await time(2000)
    console.log(`${stocks.Fruits[0]} was selected`)
  }

  catch(error){
    console.log("Customer left", error)
  }

  finally{
    console.log("Day ended, shop closed")
  }
}

// Trigger
kitchen();
```

The result looks like this when the shop is open:

```
Console
"strawberry was selected"
"Day ended, shop closed"
```

The result looks like this when the shop is closed:

```
Console
"Shop is closed" ← reject
"customer left" ← catch
"Day ended, shop closed" ← finally
```

First, open our shop

```
let is_shop_open = true;
```

Now write the steps inside our kitchen() function:

```
async function kitchen(){
  try{
    await time(2000)
    console.log(`${stocks.Fruits[0]} was selected`)

    await time(0000)
    console.log("production has started")

    await time(2000)
    console.log("fruit has been chopped")

    await time(1000)
    console.log(`${stocks.liquid[0]} and ${stocks.liquid[1]} added`)

    await time(1000)
    console.log("start the machine")

    await time(2000)
    console.log(`ice cream placed on ${stocks.holder[1]}`)

    await time(3000)
    console.log(`${stocks.toppings[0]} as toppings`)

    await time(2000)
    console.log("Serve Ice Cream")
  }

  catch(error){
    console.log("customer left")
  }
}
```

And here's the result:

Console

"strawberry was selected"

"production has started"

"fruit has been chopped"

"water and ice added"

"start the machine"

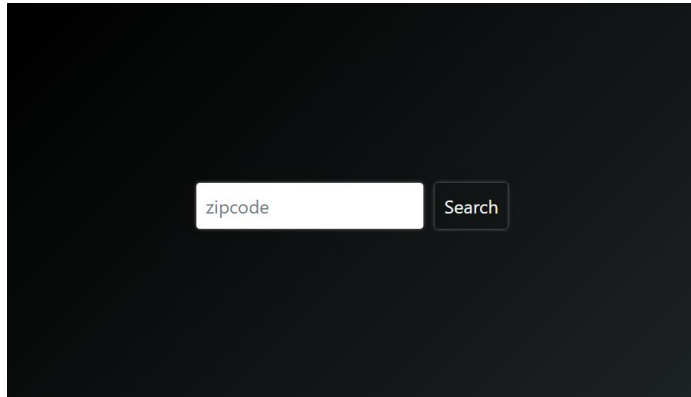
"ice cream placed on cup"

"chocolate as toppings"

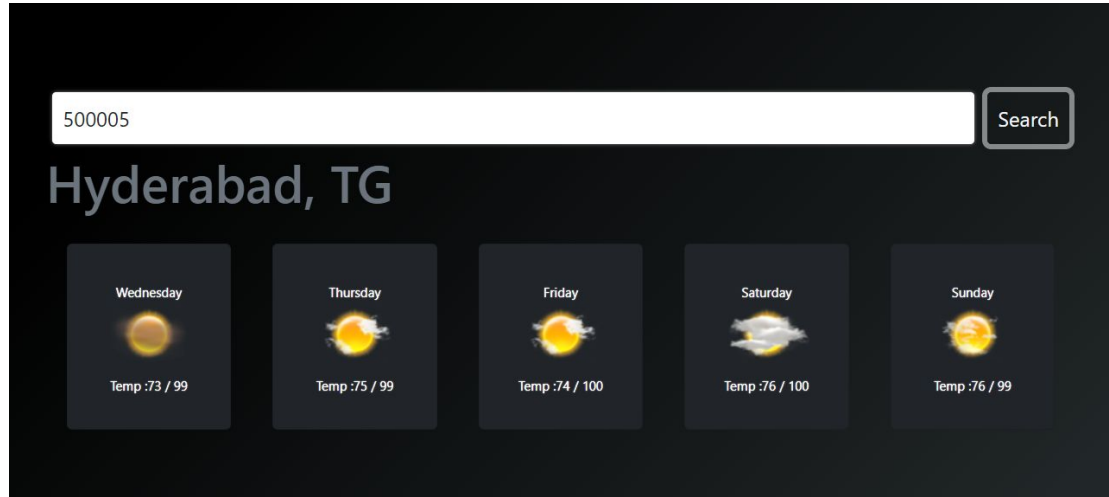
"Serve Ice Cream"

Weather Application:

This application give the weather information for the respective zipcode for five days. Below are the screenshots of the application



Weather Application:



Code link: <https://github.com/Utkarshini12/Weather>

Practice Question:

Try to implement a promise-based alarm API i.e `alarm()` pass the name of the person to wake up and delay in milliseconds to wait before waking the person up as arguments. After the delay, the function should send a “Wake up!” message, including the name of the person we need to wake up.

Upcoming class teaser

We will be going to how to implement the e-commerce app in javascript

E-commerce app in javascript

Thank You!