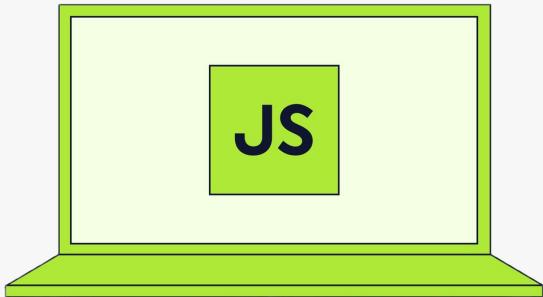




# The Complete Javascript Course



@newtonschool

## Lecture 11: Understanding APIs

-Narendra Kumar



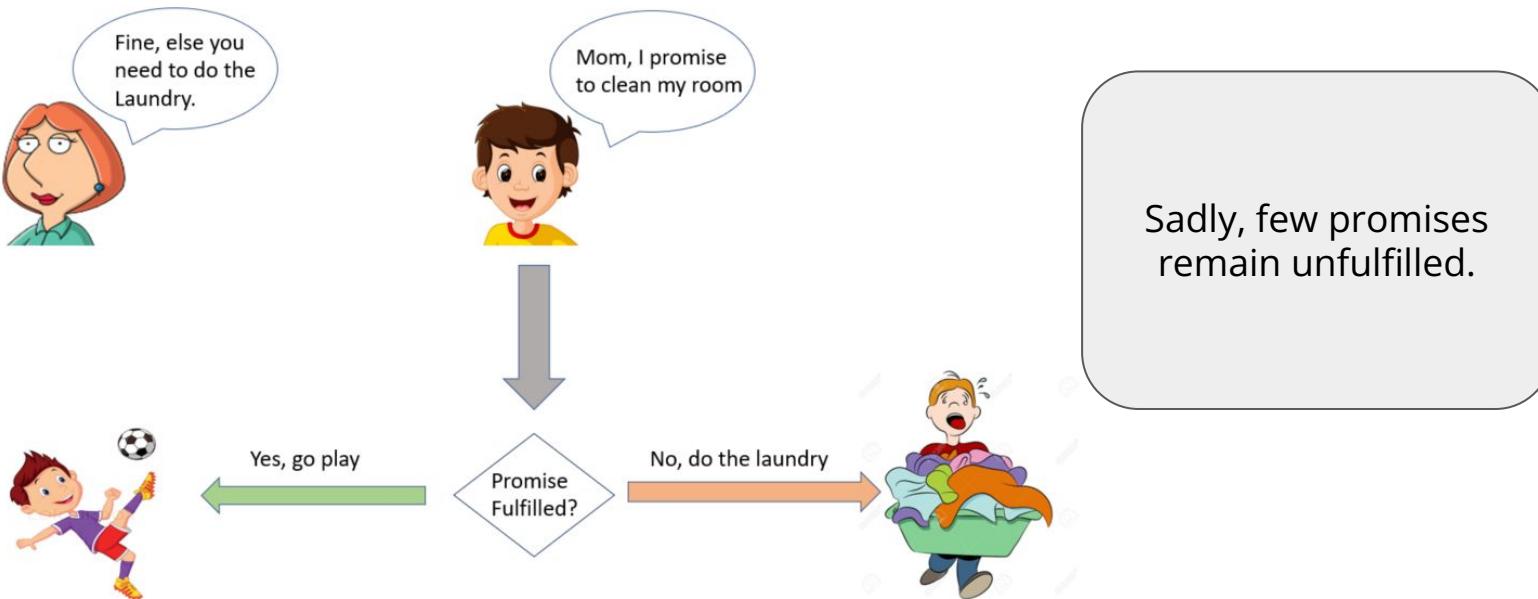
# Table of Contents

- Quick Recap
- Understanding APIs
- HTTPS REST API: Understanding with fetch()
- Error Handling
  - Basic Error Catching
  - Multiple Error Handlers
  - Recovery Patterns
- Promise: Static Methods
- Status Codes

# Quick Recap

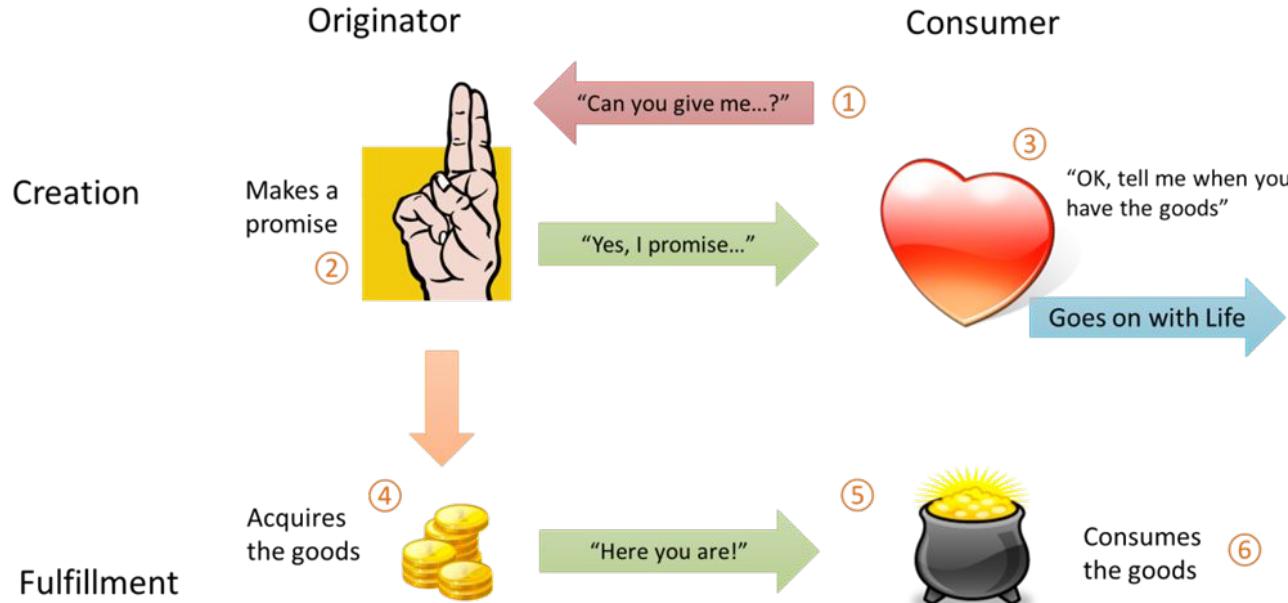
# Pre-discussed: Making a Promise

If you did, then you must know how valuable promises are. A promise in real life is a commitment or guarantee made by someone.



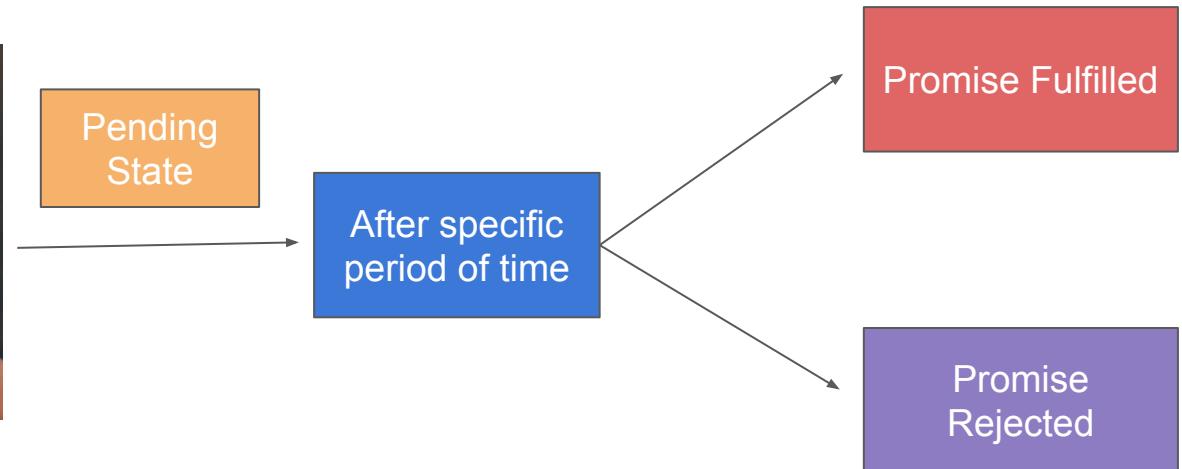
# Pre-discussed: Stages of promise

If you did then you must know how valuable promises are. A promise in real life is a commitment or guarantee made by someone.



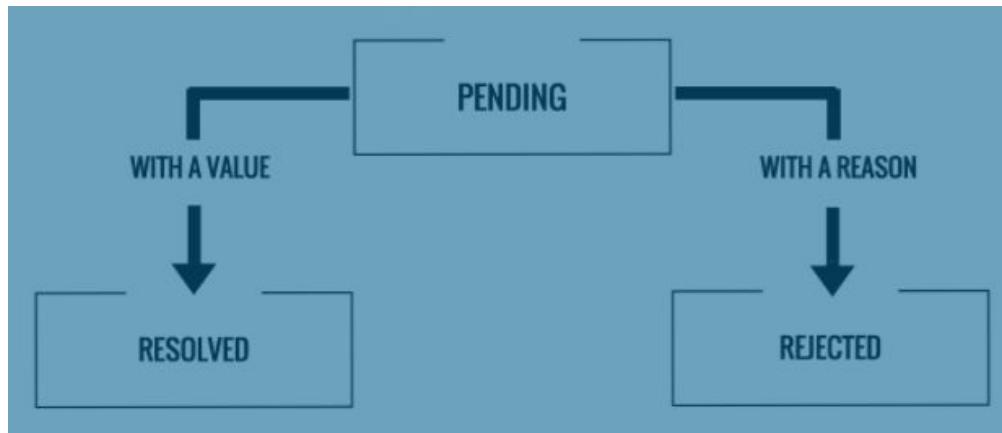
# Pre-discussed: Promises in Javascript

A promise is a commitment in programming—a guarantee that something will happen in the future, but not an immediate action. It's a placeholder for future work.



# Pre-discussed: Different promise states

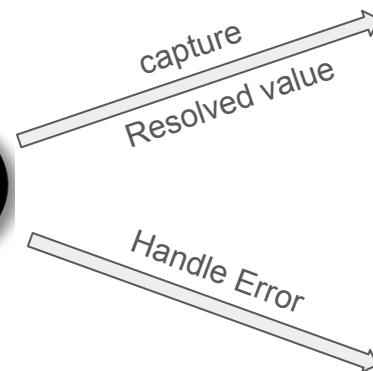
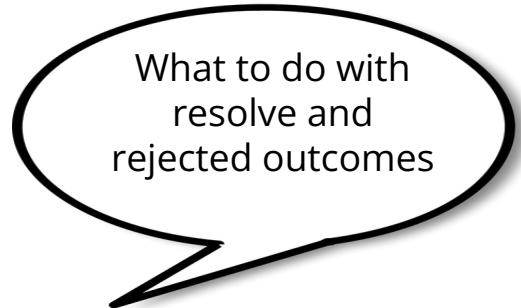
Promises in JavaScript exist in three states: pending, fulfilled, and rejected.



Initially, a promise remains in a pending state, and after some time it either gets resolved/fulfilled or rejected.

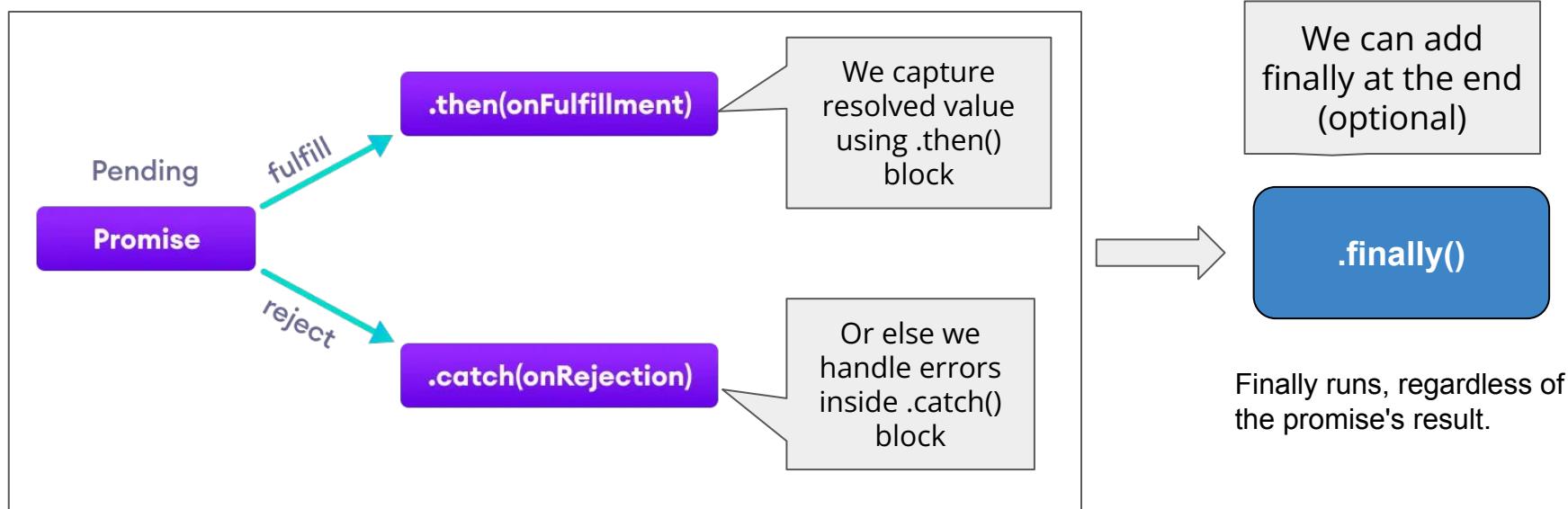
# Pre-discussed: Handling Promise

As we have learned, a promise is initially in a pending state, and then after a while it either gets resolved or rejected.



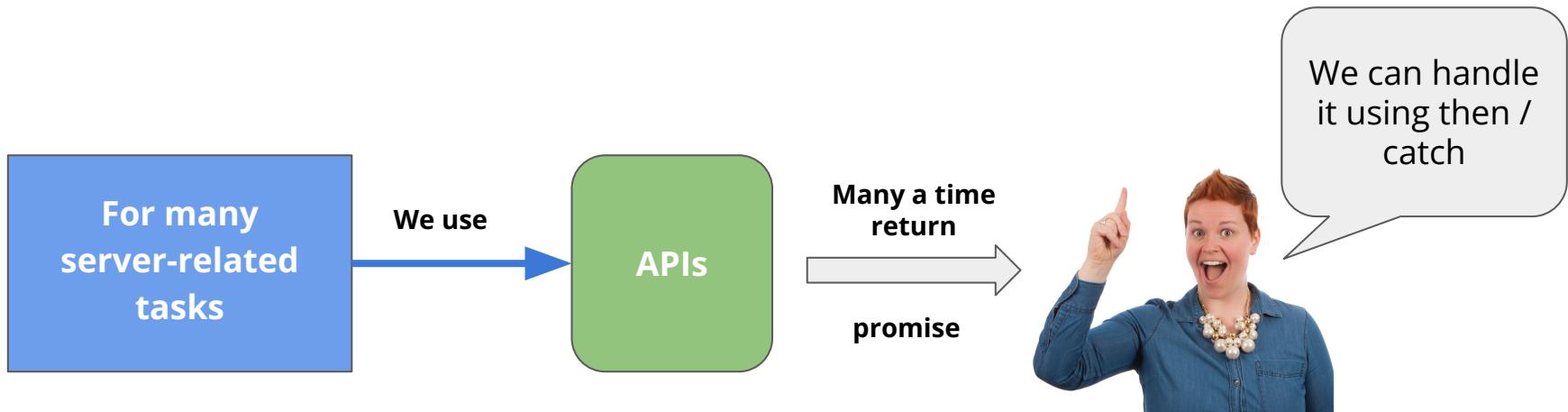
# Pre-discussed: Handling Promise

We hope a promise resolves/fulfilled and gives us a value to work with in the `then` block. However, if it gets rejected, we handle it in the `catch` block.



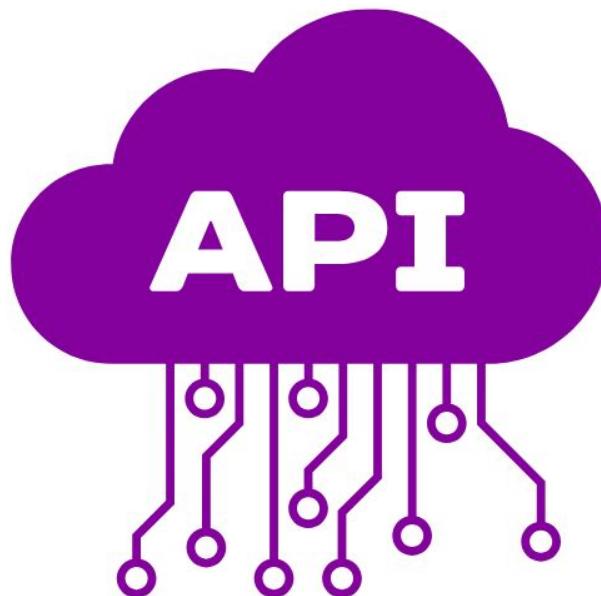
# Promises in APIs

Applications often fetch external data using APIs. Since APIs return data asynchronously, they often return Promises.



# But what are they exactly?

Before we jump into how to handle promises returned by APIs, let's understand what they are.

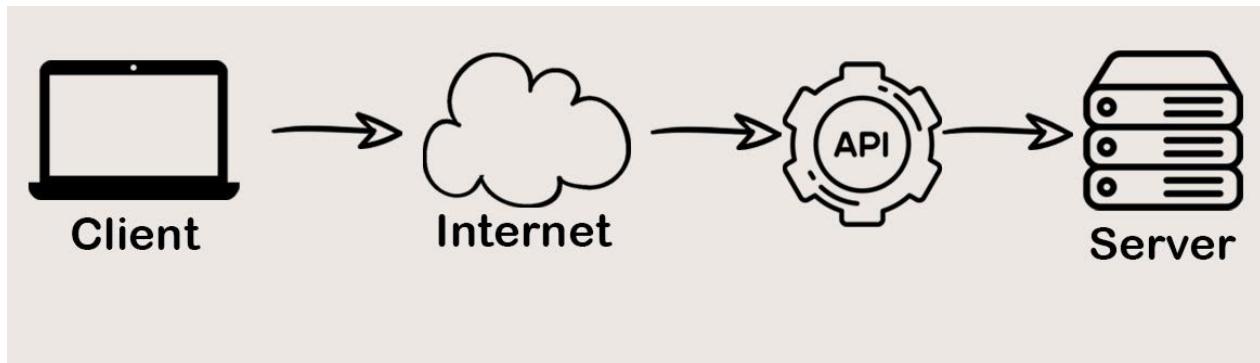


They seem to be some kind of interface, a kind of gateway to other places.

# Understanding APIs

# What is an API?

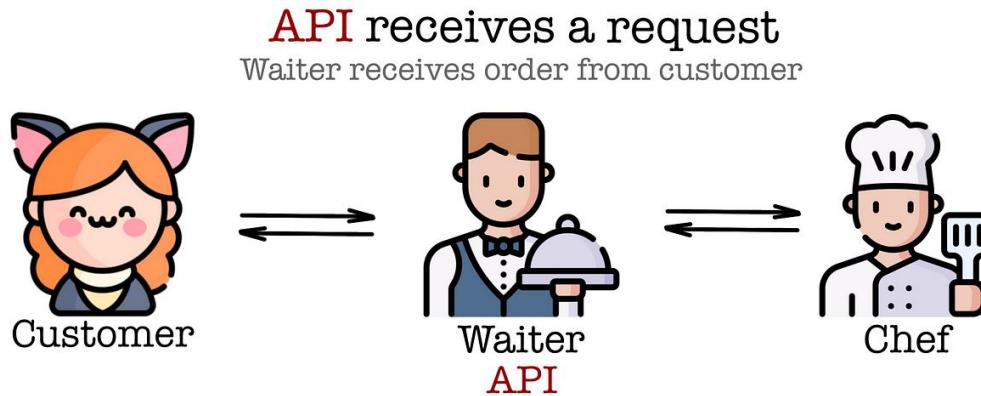
**API** stands for **Application Programming Interface**. An **API** is like a **messenger** that connects two different apps or programs to talk to each other.



Understood??  
  
Let's  
understand it  
with an analogy

# Let's understand it with an analogy

In this analogy, the waiter is like an API—taking the guest's order (request) to the chef (server) and bringing back the dish (response),



**API collects and processes a response, then returns with that response**

As waiter would take order from customer, report it to chef and delivers the answer - completed meal from kitchen

The waiter is the entry point for communication with the chef, just like an API connects different software systems.

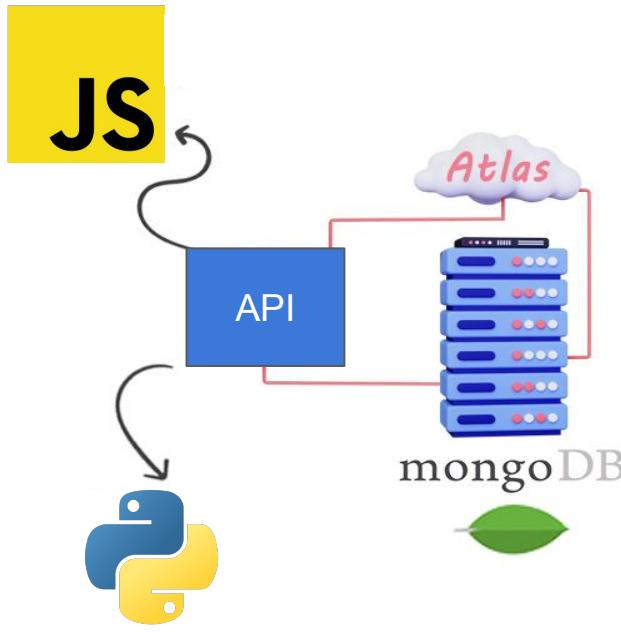
# API: Bridge Between User and Server

Just like waiter in the previous analogy, API **stands in between end user and server**, facilitating communication.



# But why not communicate directly?

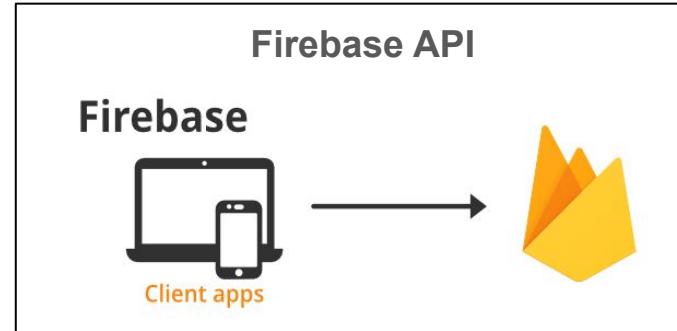
Software often uses different languages and environments, making direct communication difficult. APIs bridge that gap, allowing apps to understand each other.



Services like MongoDB and software like ChatGPT provide APIs that allow different languages to communicate efficiently.

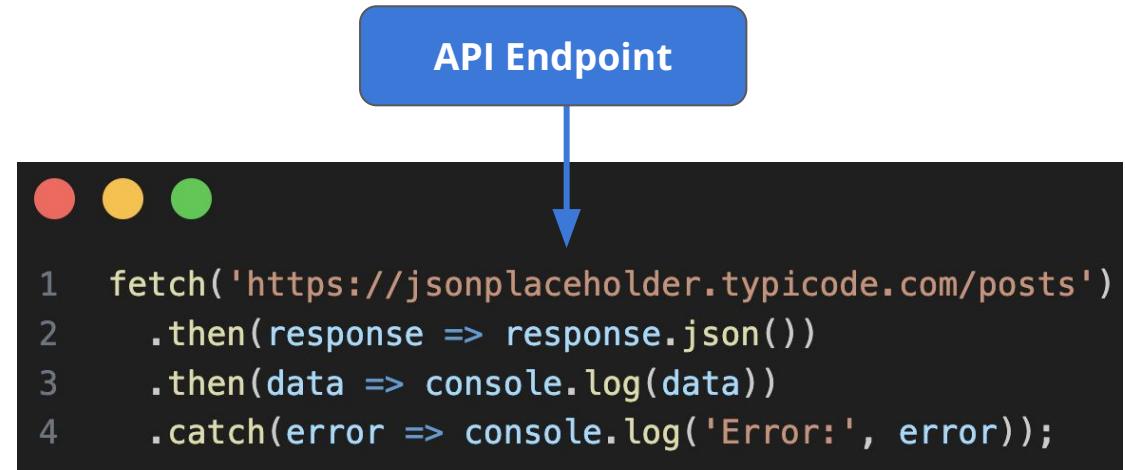
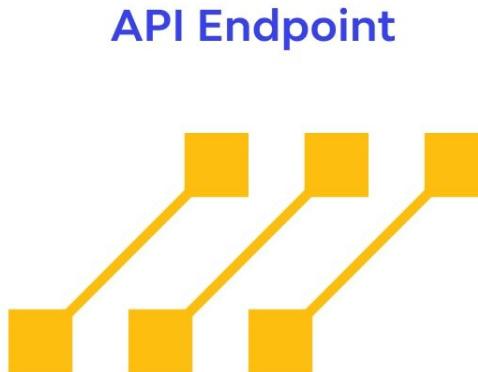
# Popular Examples of APIs in Action

Let's have a look at few popular APIs:



# API Endpoints: How to Access APIs?

An API endpoint is a specific URL using which an API can be accessed by a client application.

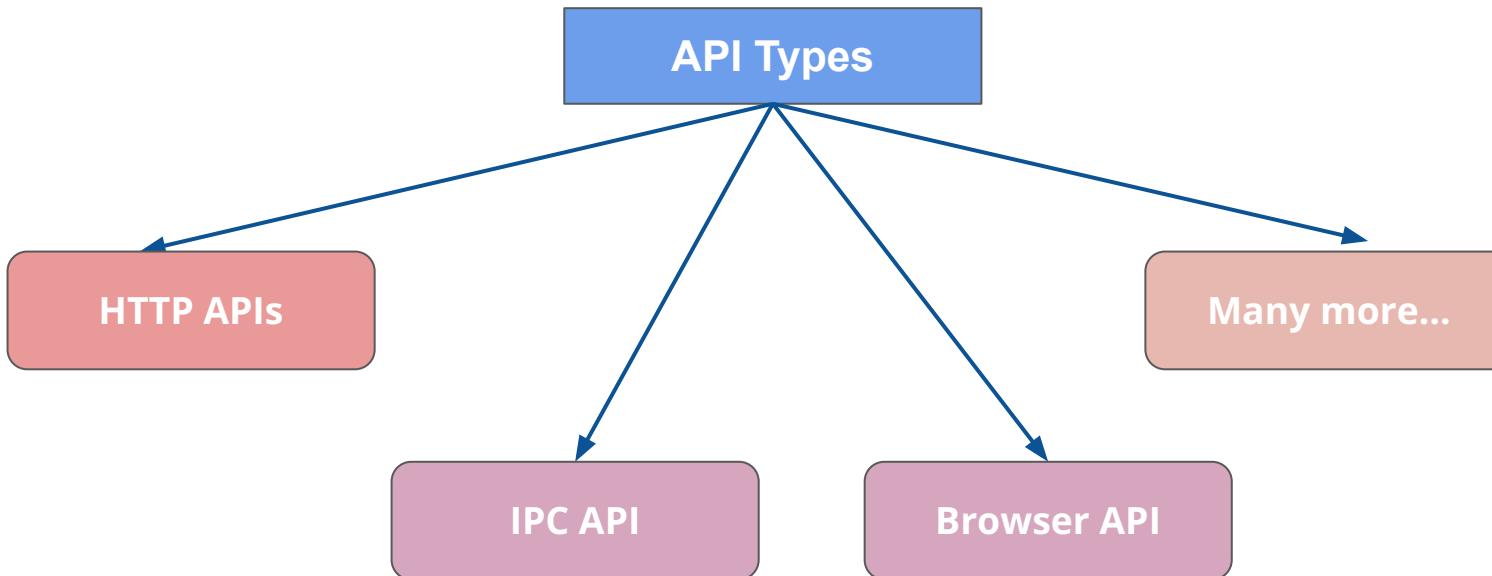


The diagram shows a blue rounded rectangle labeled "API Endpoint" at the top, with a blue arrow pointing down to a dark gray rectangular box containing code. The code is a JavaScript fetch request:

```
1  fetch('https://jsonplaceholder.typicode.com/posts')  
2    .then(response => response.json())  
3    .then(data => console.log(data))  
4    .catch(error => console.log('Error:', error));
```

# Types of APIs

There are different types of APIs. Here are few of them:



# HTTP REST API

## Understanding with fetch()

# Rest API

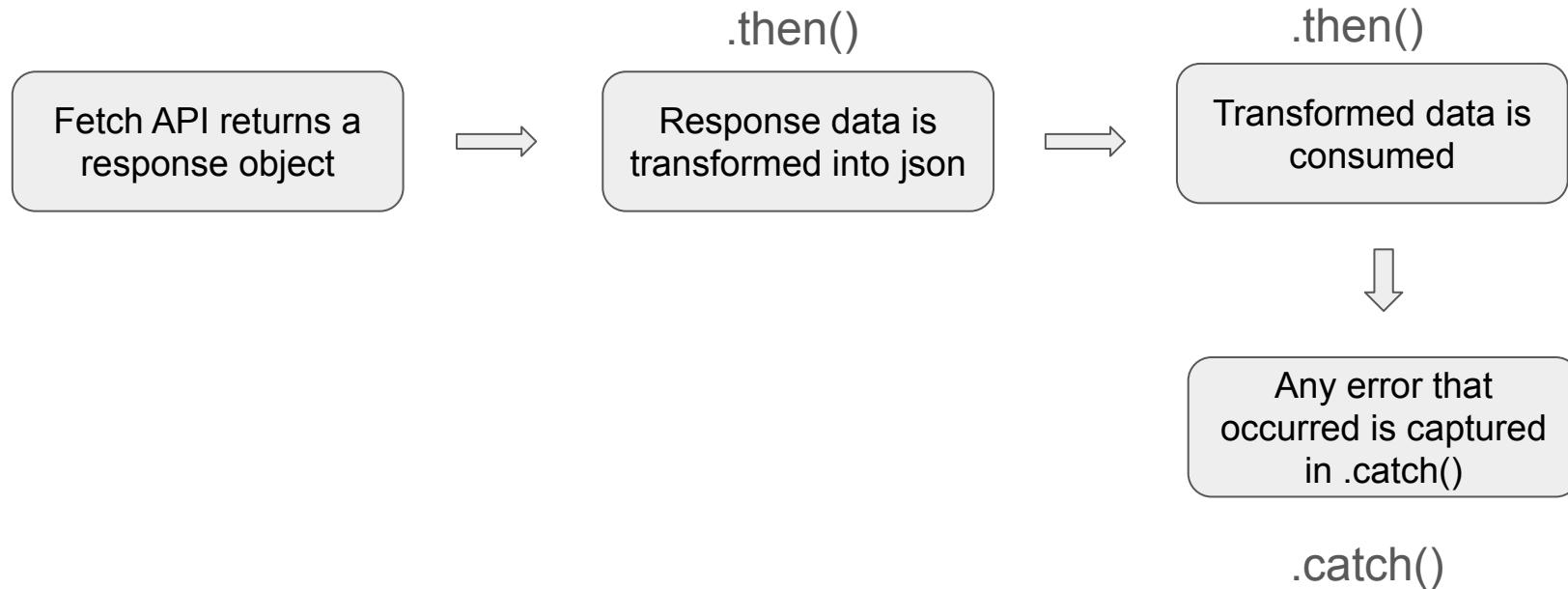
A REST API allows apps to communicate over the internet using simple commands like GET, POST, PUT, and DELETE. Each request is independent and doesn't remember the previous one.



We provide them with simple HTTP requests like GET, POST, etc., and we get data in response.

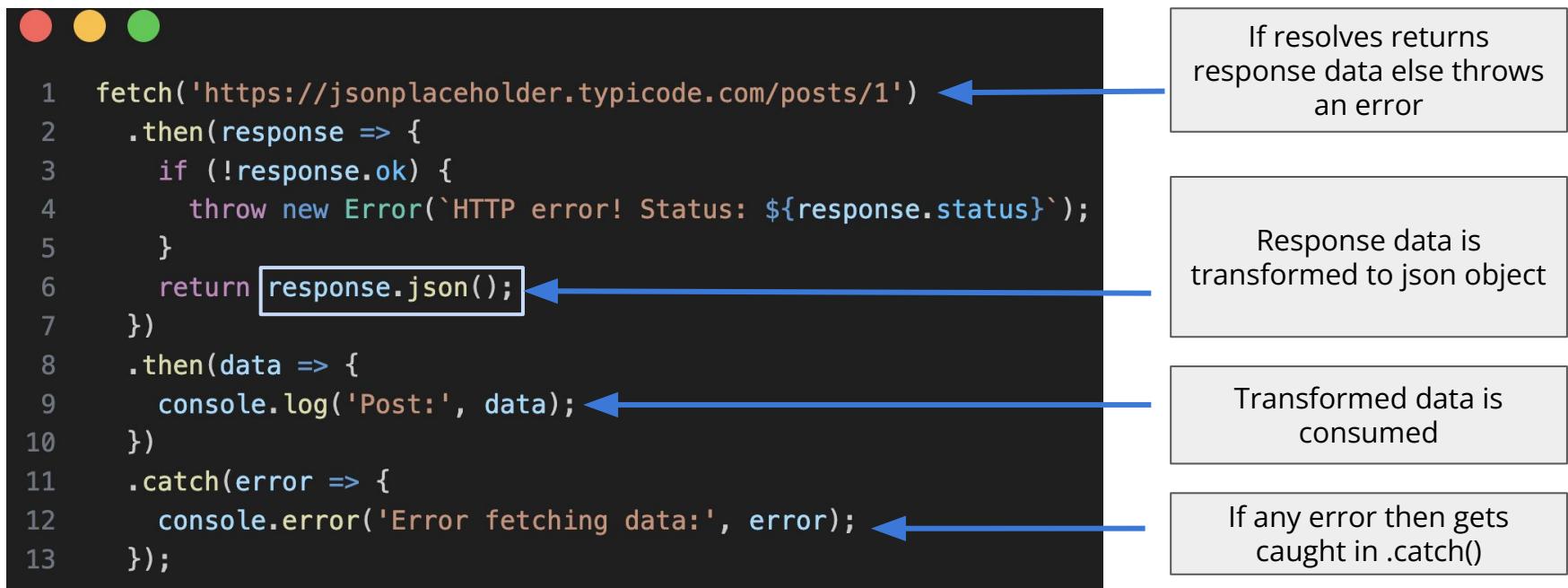
# Rest API: fetch api example

The `fetch()` API returns a promise that resolves to a response. `.then()` handles and parses the data, while `.catch()` handles errors.



# fetch API: then/catch

The `fetch()` API returns a promise that resolves to a response. `.then()` handles and parses the data, while `.catch()` handles errors.



# Settling fetch API: then/catch

Any promise gets settled if it either gets resolved or gets rejected.



```
1  fetch('https://jsonplaceholder.typicode.com/posts/1')
2    .then(response => {
3      if (!response.ok) {
4        throw new Error(`HTTP error! Status: ${response.status}`);
5      }
6      return response.json(); ←
7    })
8    .then(data => {
9      console.log('Post:', data);
10     })
11   .catch(error => {
12     console.error('Error fetching data:', error);
13   });

```

Either returns  
response data or  
throws an error

Based on results, i.e.  
response or error, it  
gets passed either to  
.then() or .catch()

# fetch API: then/catch chaining

Here fetch returns a promise; when the promise gets resolved, it returns a response object, which is received by the first block, and values get passed down in the chain.

.then() chaining,  
values returned  
from previous  
get captured in  
next .then()

```
1  fetch('https://jsonplaceholder.typicode.com/posts/1')
2    .then(response => {
3      if (!response.ok) {
4        throw new Error(`HTTP error! Status: ${response.status}`);
5      }
6      return response.json();
7    })
8    .then(data => {
9      console.log('Post:', data);
10     })
11   .catch(error => {
12     console.error('Error fetching data:', error);
13   });

```

# fetch: then/catch Data transformation

The received data from the fetch is in the form of a string, which needs to be transformed into a JSON object for further processing.

```
1  fetch('https://jsonplaceholder.typicode.com/posts/1')
2    .then(response => {
3      if (!response.ok) {
4        throw new Error(`HTTP error! Status: ${response.status}`);
5      }
6      return response.json(); ←
7    })
8    .then(data => {
9      console.log('Post:', data);
10     })
11   .catch(error => {
12     console.error('Error fetching data:', error);
13   });

```

Response data is  
transformed into json  
object

# fetch: then/catch Data transformation

The received data from the fetch is in the form of a string, which needs to be transformed into a JSON object for further processing.

```
1  fetch('https://jsonplaceholder.typicode.com/posts/1')
2    .then(response => {
3      if (!response.ok) {
4        throw new Error(`HTTP error! Status: ${response.status}`);
5      }
6      return response.json();
7    })
8    .then(data => {
9      console.log('Post:', data); ←
10     })
11    .catch(error => {
12      console.error('Error fetching data:', error);
13    });

```

Transformed response  
data returned by .then()  
gets received by next  
.then() in the chain

# fetch: then/catch Catching the Error

Error occurred whether due to server or error thrown from anywhere inside our code gets caught in the .catch().

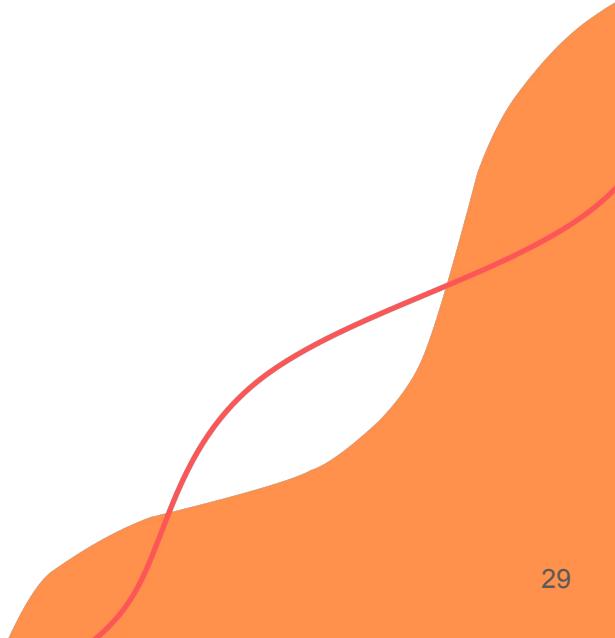
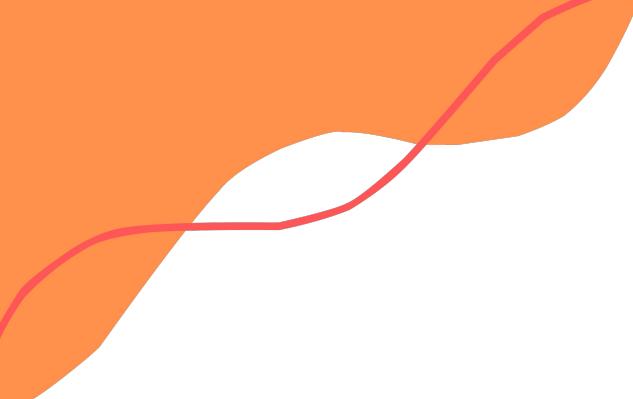


```
1  fetch('https://jsonplaceholder.typicode.com/posts/1')
2    .then(response => {
3      if (!response.ok) {
4        throw new Error(`HTTP error! Status: ${response.status}`);
5      }
6      return response.json();
7    })
8    .then(data => {
9      console.log('Post:', data);
10   })
11   .catch(error => {
12     console.error('Error fetching data:', error);
13   });

```

Error thrown from  
inside our code

Error get captured in  
.catch()



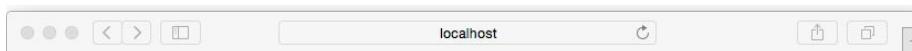
# **then/catch**

## **Workshop**

# Workshop: Question 1

You are building a simple weather application that fetches data from a weather API. If the fetch operation fails, you want to display an error message to the user. How would you structure the promise to handle this error?

**How would you handle errors using catch()?**



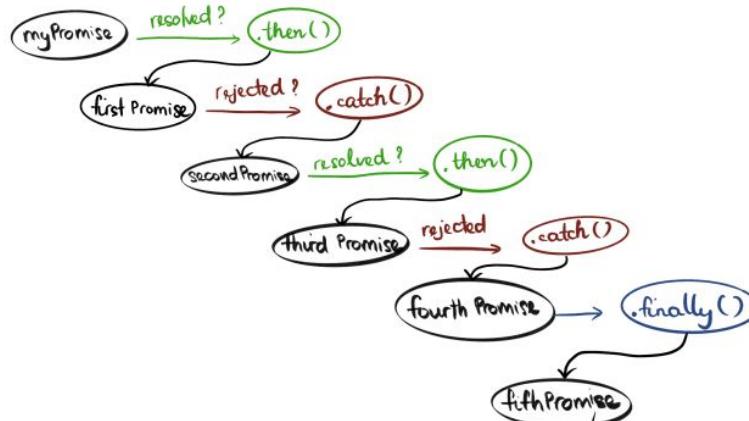
## Server Error

We're sorry! The server has encountered an internal error and was unable to complete your request. Please contact the system administrator for more information.

# Workshop: Question 2

You need to fetch user data from an API and then fetch their favorite products from another API.

**How would you chain these two promises and handle any error that may occur in either of the steps?**



# fetch API: HTTP methods

Fetch APIs use standard HTTP methods to perform actions on resources. Each method has a specific purpose and meaning.



**GET**

Retrieve a  
resource



**POST**

Create a  
resource



**PUT**

Replace a  
resource



**PATCH**

Update a  
resource



**DELETE**

Delete a  
resource

# fetch() - GET method

Let's understand REST API with an example of fetch API. If we don't specify anything in fetch, then it does GET request by default. A GET request retrieves a resource from the server.



URL for the GET request

```
1 fetch('https://jsonplaceholder.typicode.com/posts')
2   .then(response => response.json())
3   .then(data => console.log(data))
4   .catch(error => console.log('Error:', error));
```

It retrieves data from the API endpoint, expecting the response to be a stream of bytes convertible to JSON format which process using the `.json()` method.

# fetch() - POST method

If we want to send some data instead of requesting, we can do a post request with appropriate data specifying what data is being sent in the body.

```
● ● ●  
1  fetch('https://jsonplaceholder.typicode.com/posts', {  
2      method: 'POST', →  
3      headers: {  
4          'Content-Type': 'application/json' →  
5      },  
6      body: JSON.stringify({  
7          title: 'New Post',  
8          body: 'This is a new post', userId: 1 →  
9      })  
10 }  
11     .then(response => response.json())  
12     .then(data => console.log(data))  
13     .catch(error => console.log('Error:', error));
```

To send data, we do POST request

Specifying data format is JSON

In the body, we convert the JSON data to a string so it can be sent over the network.

# fetch() - PATCH method

PATCH request updates a resource or data stored in the server.



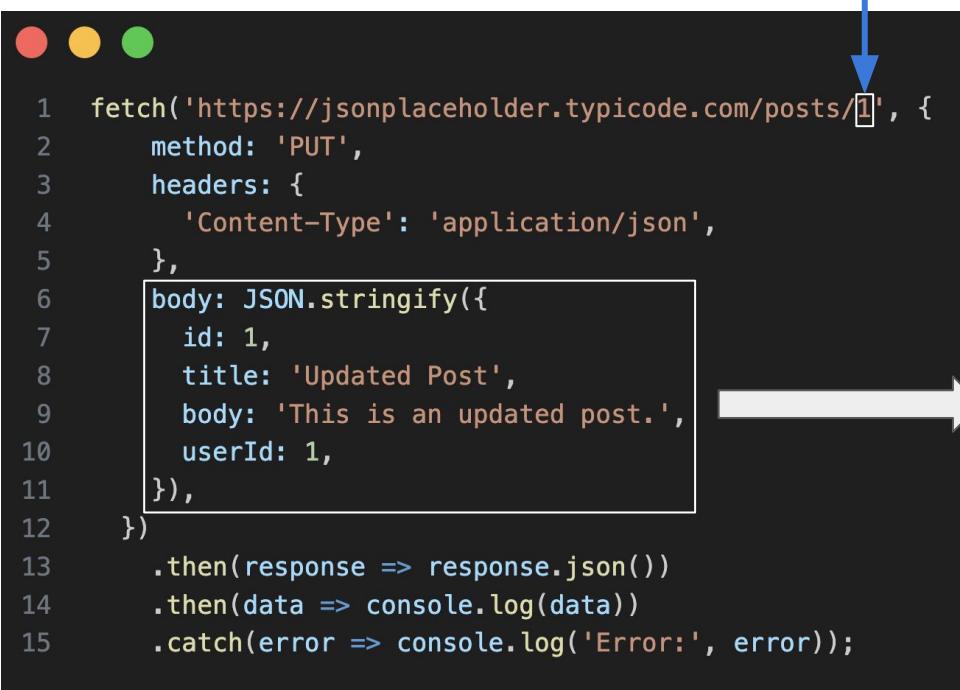
```
1  fetch('https://jsonplaceholder.typicode.com/posts/1', {  
2      method: 'PATCH',  
3      headers: {  
4          'Content-Type': 'application/json'  
5      },  
6      body: JSON.stringify({ title: 'Updated Post Title' })  
7  })  
8      .then(response => response.json())  
9      .then(data => console.log(data))  
10     .catch(error => console.log('Error:', error));
```

We need to pass an identifier to identify which item to patch in database

And need to supply the value to update.

# fetch() - PUT method

Similarly we can perform PUT request:-



```
1  fetch('https://jsonplaceholder.typicode.com/posts/1', {  
2      method: 'PUT',  
3      headers: {  
4          'Content-Type': 'application/json',  
5      },  
6      body: JSON.stringify({  
7          id: 1,  
8          title: 'Updated Post',  
9          body: 'This is an updated post.',  
10         userId: 1,  
11     }),  
12 })  
13     .then(response => response.json())  
14     .then(data => console.log(data))  
15     .catch(error => console.log('Error:', error));
```

Instead of updating the resource, PUT request replaces the entire resource.

This new data would replace the data entry with identifier 1

# fetch() - DELETE method

In the same manner as that of PATCH we can perform deletion.



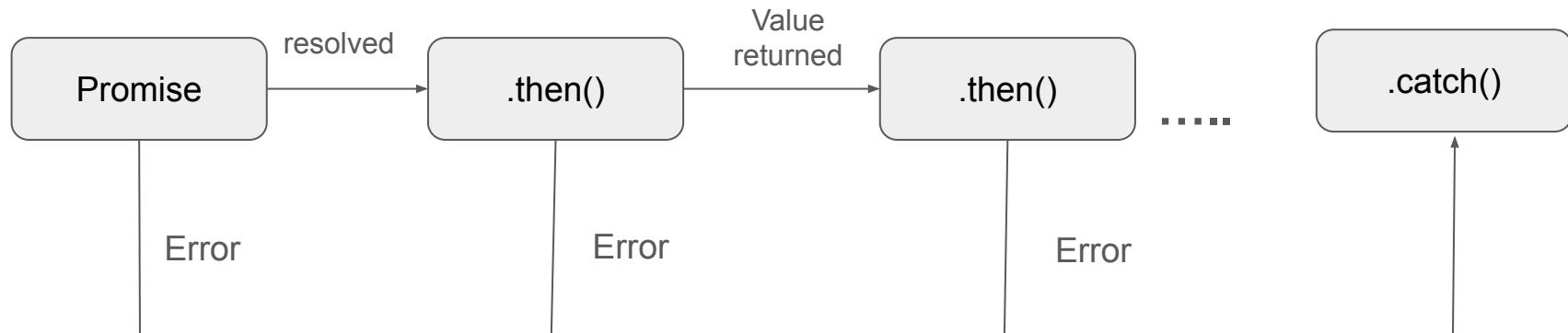
```
1 fetch('https://jsonplaceholder.typicode.com/posts/1', {  
2   method: 'DELETE'  
3 })  
4   .then(response => response.json())  
5   .then(data => console.log('Deleted:', data))  
6   .catch(error => console.log('Error:', error));
```

Unlike PATCH we  
don't need to  
pass body in  
DELETE request.

# Error Handling

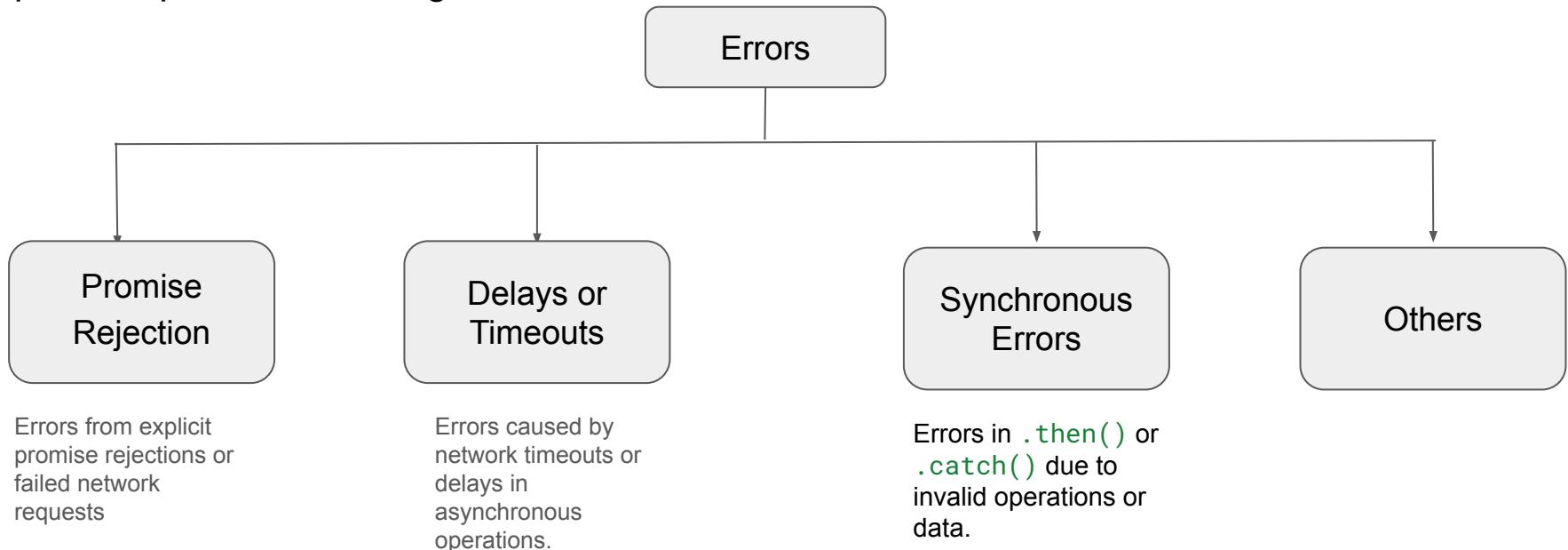
# Error Handling: Catch block

The `.catch()` block is used to handle any errors that occur during the execution of a promise or any promise chain. It catches both rejections and exceptions that occur in the promise lifecycle.



# Handling Errors

There are following types of errors which can be thrown during the execution of a promise/promise handling.



# Promise Rejection

**Promise rejection errors** occur when a promise is explicitly rejected or if an operation inside the promise fails.

A promise can be manually rejected using `reject()`, or due to network/HTTP failures like 404 or 500 errors, especially in API requests.



# Promise Rejection: Manual rejection

**Promise rejection errors** occur when a promise is explicitly rejected or if an operation inside the promise fails.

```
1 let myPromise = new Promise((resolve, reject) => {
2     reject('Something went wrong');
3 });
4
5 myPromise
6     .catch(error => {
7         console.error('Caught error:', error);
8     });

```

Here we simply caught error  
and displayed using  
console.error()

# Promise Rejection: Network Http failures

For network or HTTP failures, check the response status and handle errors using `.catch()` or inside the `fetch()` response block.

```
1  fetch('https://invalidurl.com')
2    .then(response => {
3      if (!response.ok) {
4        throw new Error('Network error');
5      }
6      return response.json();
7    })
8    .catch(error => {
9      console.error('Network error:', error);
10 });

```

It might happened that you got a response but it is a failed response, in that case you need to throw an error from inside `.then()`

# Delays or Timeouts

To handle timeouts, create a timeout promise and reject if the operation exceeds the allowed time.

```
1 let timeoutPromise = new Promise((resolve, reject) => {
2     setTimeout(() => reject('Operation timed out'), 5000);
3 });
4
5 timeoutPromise.catch(error => {
6     console.error('Timeout error:', error);
7 });
```

Most asynchronous APIs, especially network requests like HTTP fetch calls, do have a built-in timeout to prevent the request from hanging indefinitely

# Synchronous Errors

Synchronous errors like invalid input/data, type errors etc, could occur for which we need to throw an error manually while creating the promise.



```
1 let processData = new Promise((resolve, reject) => {
2     let input = null;
3     if (!input) {
4         reject('Invalid data');
5     }
6 });
7
8 processData.catch(error => {
9     console.error('Data error:', error);
10});
```



ERROR

Invalid data

# Handling Errors at different stages

We can add error handlers at different points to manage errors at each stage.

```
1  fetch('https://jsonplaceholder.typicode.com/posts')
2  .then(response => {
3      if (!response.ok) {
4          throw new Error('Network error: Failed to fetch posts');
5      }
6      return response.json(); // Parse the JSON data
7  })
8  .then(posts => {
9      return fetch('https://jsonplaceholder.typicode.com/comments')
10     .then(response => {
11         if (!response.ok) {
12             throw new Error('Network error: Failed to fetch comments');
13         }
14         return response.json(); // Parse the comments
15     })
16     .catch(commentError => {
17         console.error('Error in fetching comments:', commentError);
18         throw new Error('Comments stage failed');
19     });
20 }
21 .catch(error => {
22     // General error handling for any unhandled errors in chain
23     console.error('General error occurred:', error);
24 })
```

Generally we place multiple .catch() to handle specific type of error at different places

Throws an error only if comment fetching fails.

Handles errors anywhere in the code.

# Recovering from errors

Recovering from Errors refers to the process of handling errors in a way that allows the program to continue running or perform a fallback action instead of crashing.



```
1  fetch('https://jsonplaceholder.typicode.com/posts')
2  .then(response => {
3    if (!response.ok) {
4      throw new Error('Network error: Failed to fetch posts');
5    }
6    return response.json();
7  })
8  .then(data => {
9    console.log('Fetched posts:', data);
10 })
11 .catch(error => {
12   console.error('Error occurred:', error);
13   // Fallback: Provide default data or retry
14   const fallbackData = [{ id: 1, title: 'Fallback post' }];
15   console.log('Using fallback data:', fallbackData);
16 });


```

Let's say that fetch operation failed to retrieve data from the server, in that case we can provide some fallback data.

# Recovering from errors

Or we can retry by sending one more fetch request to the server:-

```
1  function fetchWithRetry(url, retries = 3) {  
2      return fetch(url)  
3          .then(response => {  
4              if (!response.ok) {  
5                  throw new Error('Network error');  
6              }  
7              return response.json();  
8          })  
9          .catch(error => {  
10              if (retries > 0) {  
11                  console.log(`Retrying... Attempts left: ${retries}`);  
12                  return fetchWithRetry(url, retries - 1); // Retry  
13              } else {  
14                  console.error('Max retries reached');  
15                  throw error; // Fail after retries  
16              }  
17          });  
18      }  
19  
20  fetchWithRetry('https://jsonplaceholder.typicode.com/posts');
```

Here, we are recursively calling `fetchWithRetry()`, in case network error occurs along with limiting the number of retries.

# Recovering from errors

Or else we can gracefully show minimal or different content

```
1  fetch('https://jsonplaceholder.typicode.com/posts')
2    .then(response => response.json())
3    .then(data => {
4      // Process data
5    })
6    .catch(error => {
7      console.error('Showing limited content');
8      // Show minimal or default content
9      displayMinimalContent();
10 });

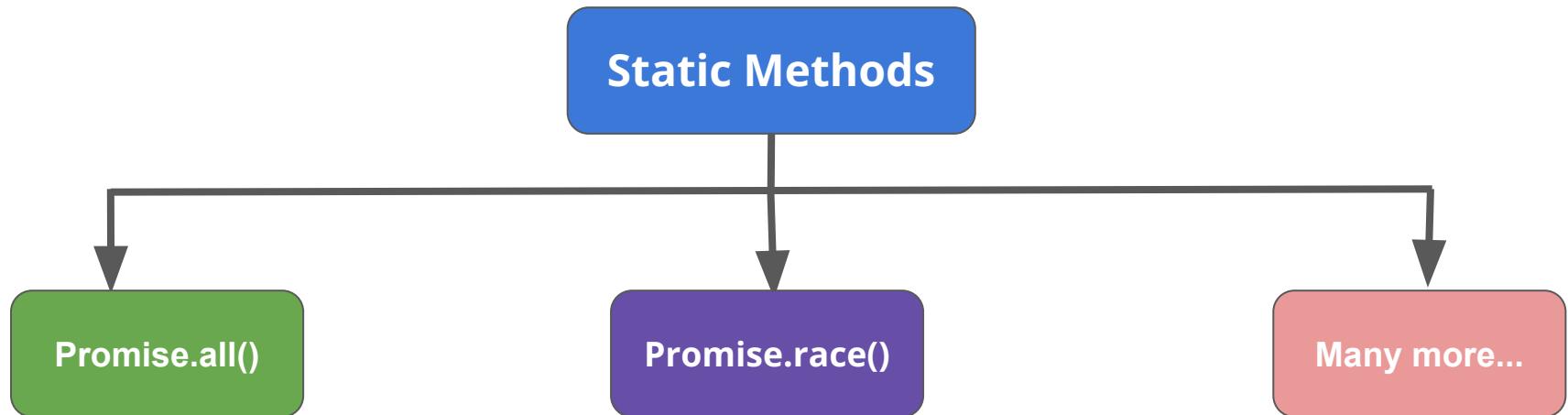
```

Or we can simply state  
“Difficulty in fetching data, try  
again later”.

# Promise: Static Methods

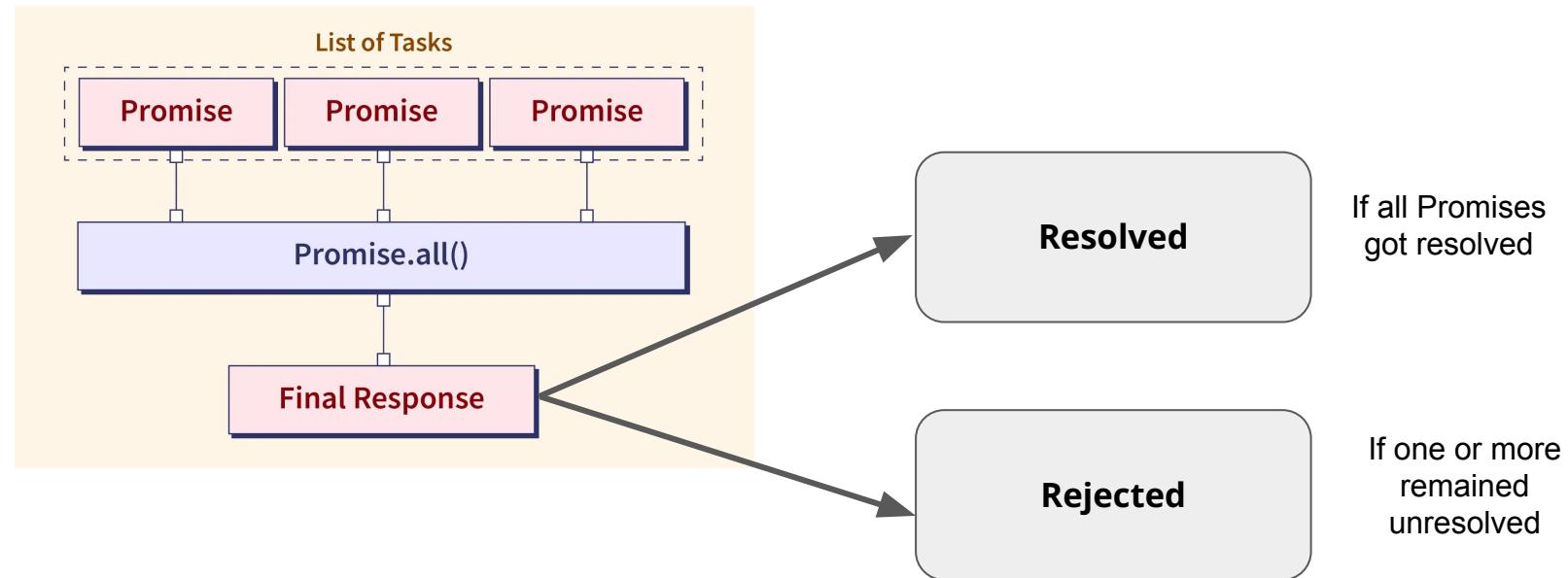
# Promise: What are Static methods?

Static methods of the `Promise` class are called on the `Promise` constructor to manage multiple promises or handle resolution globally.



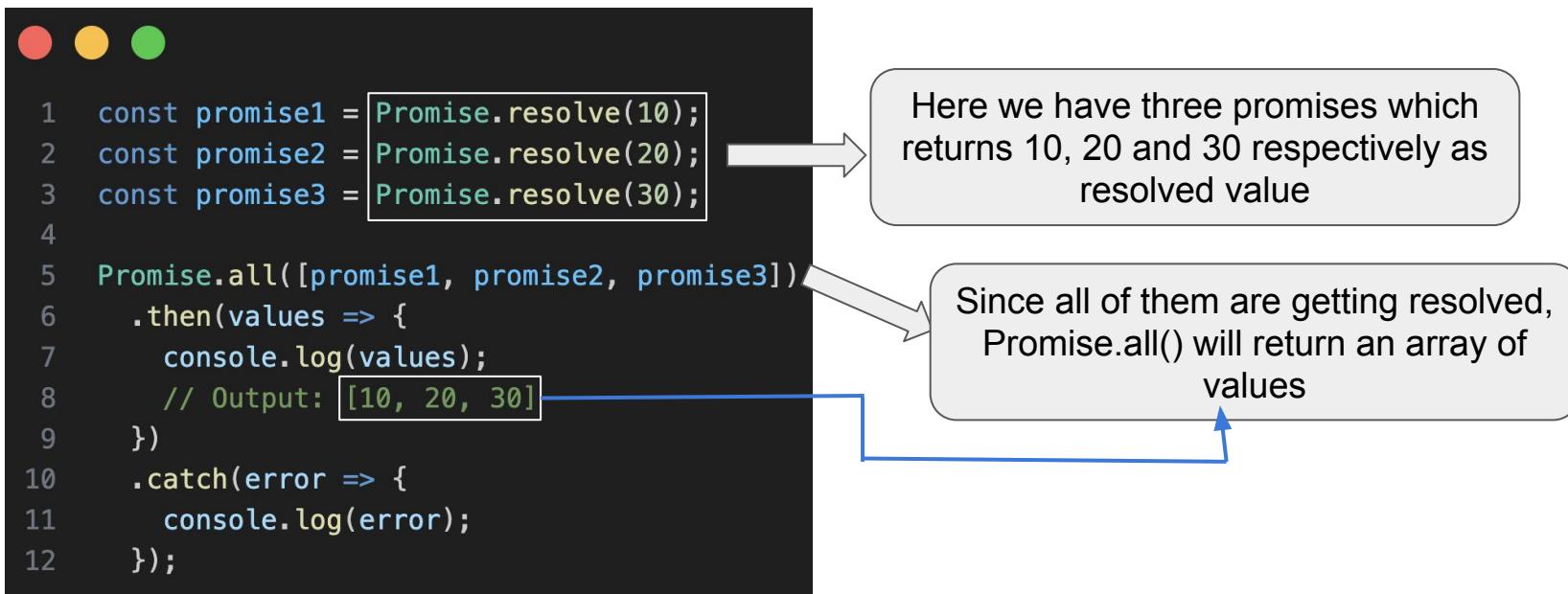
# Static Method: Promise.all()

Promise.all() is a static method that takes an array of promises and returns a single promise. It resolves when all promises resolve or rejects as soon as any promise rejects.



# Promise.all(): Example

Promise.all() is a static method that takes an array of promises and returns a single promise. It resolves when all promises resolve or rejects as soon as any promise rejects.



```
1 const promise1 = Promise.resolve(10);
2 const promise2 = Promise.resolve(20);
3 const promise3 = Promise.resolve(30);
4
5 Promise.all([promise1, promise2, promise3])
6   .then(values => {
7     console.log(values);
8     // Output: [10, 20, 30]
9   })
10  .catch(error => {
11    console.log(error);
12  });
```

Here we have three promises which returns 10, 20 and 30 respectively as resolved value

Since all of them are getting resolved, Promise.all() will return an array of values

# Promise.all(): Example

Let's say one of the promise is not getting resolved.

```
1 const promise1 = Promise.resolve(10);
2 const promise2 = Promise.reject('Error occurred');
3 const promise3 = Promise.resolve(30);
4
5 Promise.all([promise1, promise2, promise3])
6   .then(values => {
7     console.log(values);
8     // This will not execute
9   })
10  .catch(error => {
11    console.log(error);
12    // Output: 'Error occurred'
13  });
```

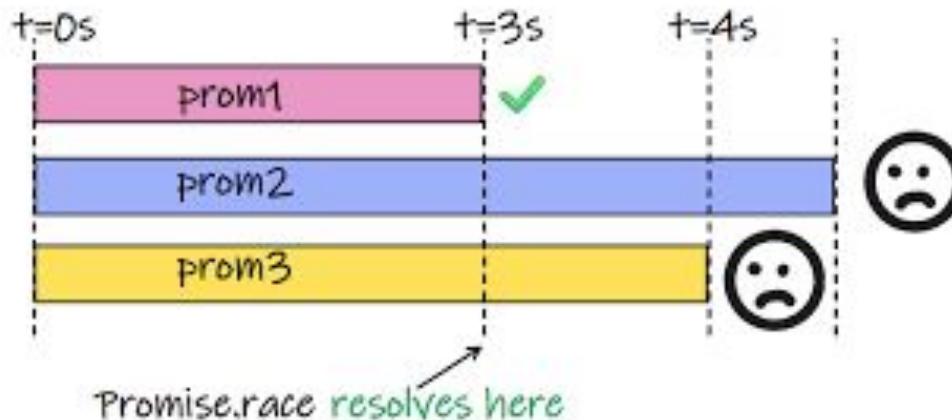
This promise is being rejected

So instead an error would be thrown

And error gets captured by catch block

# Static Method: Promise.race()

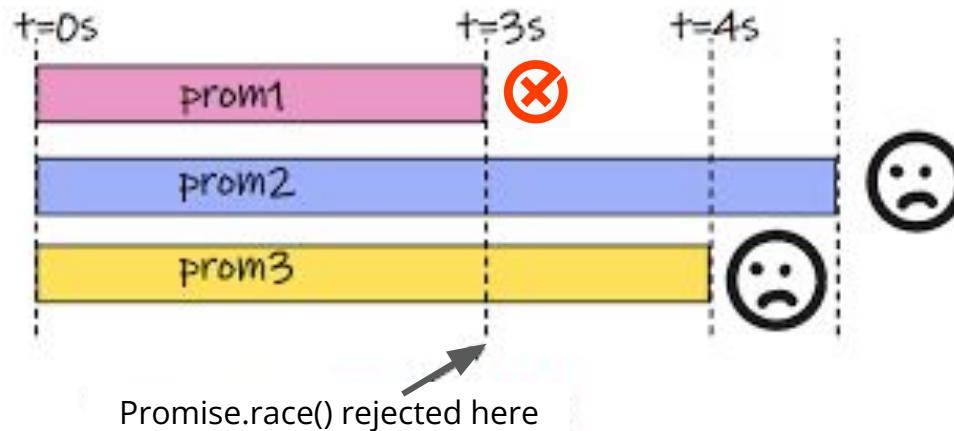
`Promise.race()` is a static method that takes an array of promises and returns a single promise, which resolves or rejects as soon as one of the promises settles.



Since the first promise in `Promise.race` is settled with state `resolve`, `Promise.race` will be *resolved*.

# Static Method: Promise.race()

Let's say first settled promise gets rejected, Promise.race() would also be settled as rejected.



Since the first promise in Promise.race is settled with state reject, Promise.race will be *rejected*.

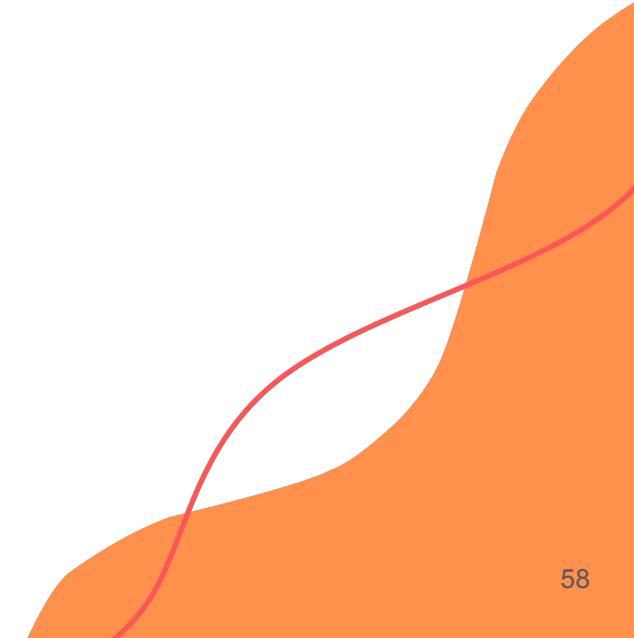
# Promise.all(): Example

We are fetching data from a URL and setting a timeout. If the timeout occurs before the data is fetched, `Promise.race()` will reject; otherwise, it will resolve with the data.



```
1 const fetchData = fetch('https://api.example.com/data');
2 const timeout = new Promise(_, reject) =>
3   setTimeout(() => reject('Request timed out'), 3000)
4 );
5
6 Promise.race([fetchData, timeout])
7   .then(response => {
8     console.log('Data fetched:', response);
9   })
10  .catch(error => {
11    console.error('Error:', error);
12    // Output: 'Request timed out' if timeout occurs
13  });
```

If data is fetched before 3000 milliseconds then promise would be resolved else it would be rejected.



# Promise.all() Workshop

# Workshop: Question

You had a form where users submit their name, email, and age. You have validated it, now you need to consume the promises and process the data only if all validations pass. If any validation fails, an error message should be displayed to the user.

**How can you consume the promise?**

Consume the  
Promise



The form consists of three horizontal input fields. The first field is labeled "Name", the second is labeled "Email", and the third is labeled "Age". Each label is positioned above its corresponding input field.

Consume it  
using  
then/catch

# Workshop: Solution

Let's consume the promise:-

```
1 // Function to handle form submission
2 function handleFormSubmission(name, email, age) {
3     // Validate all fields using promises and consume them
4     Promise.all([
5         validateName(name),
6         validateEmail(email),
7         validateAge(age)
8     ])
9     .then((validationResults) => {
10         // All validations passed, process the form
11         console.log('Form submission successful!');
12         console.log(validationResults);
13         // You can log validation results if needed
14         // Proceed to save data or display success message
15     })
16     .catch((error) => {
17         // Handle the first validation error that occurs
18         console.log('Form submission failed:', error);
19     });
20 }
```

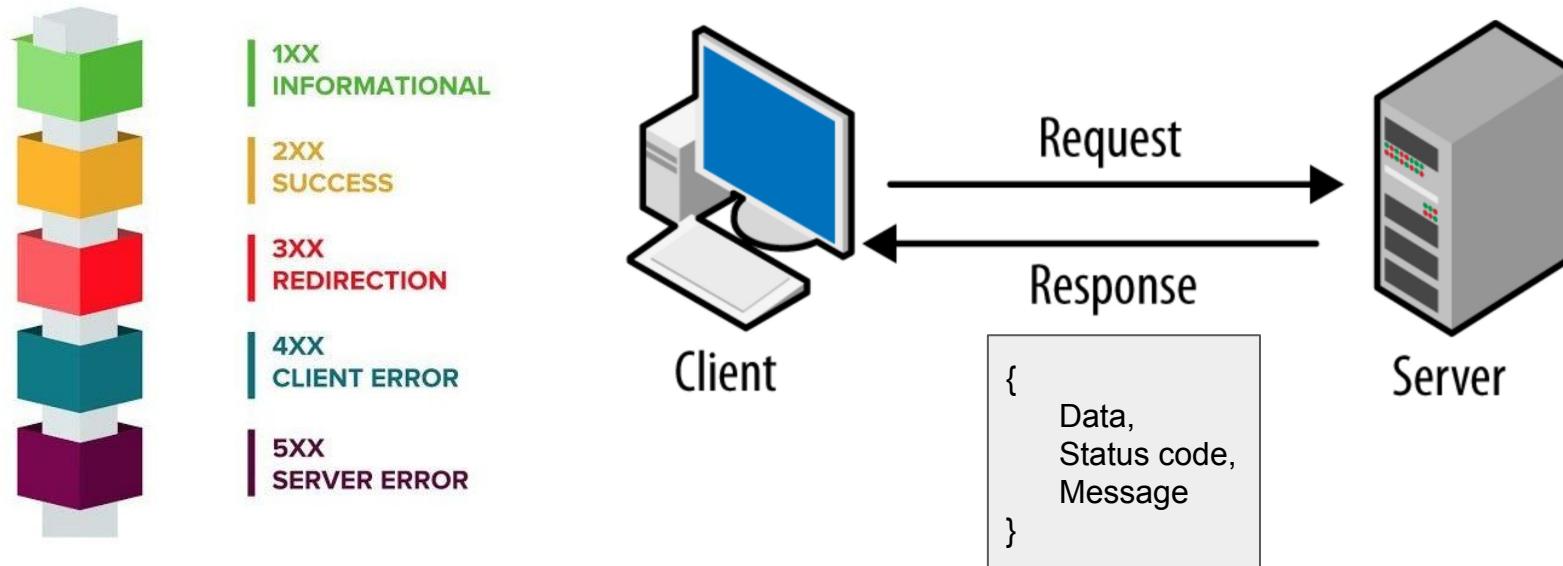
Here we used `Promise.all()` to consume all the promises at once.

# Status Codes

Signalling between Client and Server

# Status Codes: Signaling Success and Failure

When a request is processed, the server signals the outcome (success or failure) using status codes, providing clear communication about the operation's result.



# Status Code: Retrieving Status Code

When you make a fetch request, it returns a response with a status code. You can access the status code using `response.status` to check the outcome of the request.

```
1  fetch('https://jsonplaceholder.typicode.com/posts')
2    .then(response => {
3      console.log('Status Code:', response.status);
4      // Retrieves and logs the status code
5      return response.json();
6      // Process the response body if needed
7    })
8    .then(data => {
9      console.log('Data:', data);
10    })
11   .catch(error => {
12     console.log('Error:', error);
13   });

```

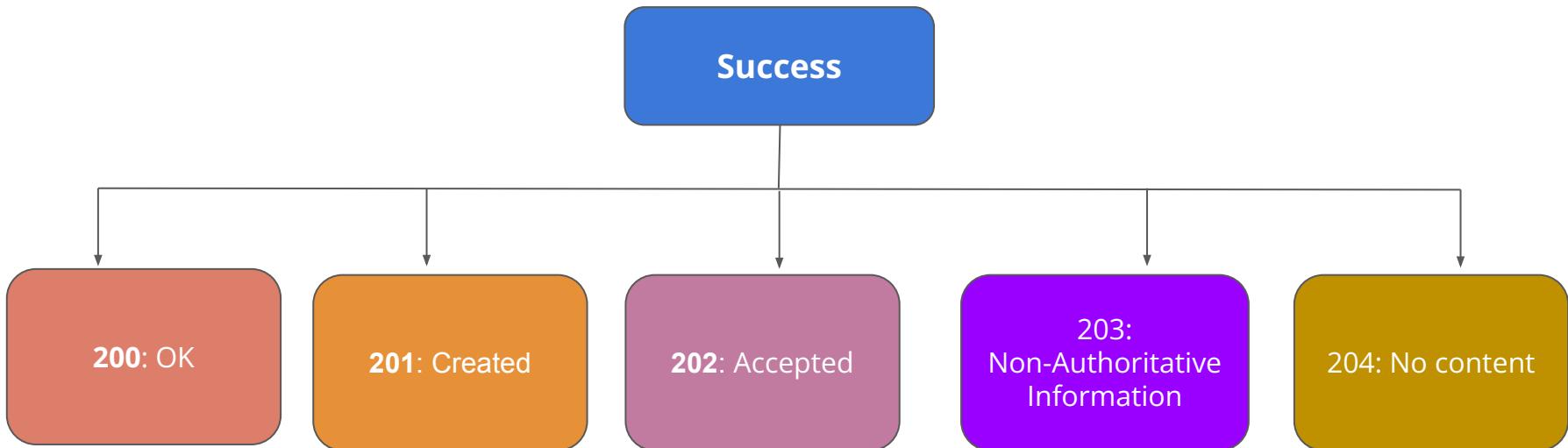
Here we are capturing status code

We can use some other variable let say *data*, then we get status code using:-

*data.status*

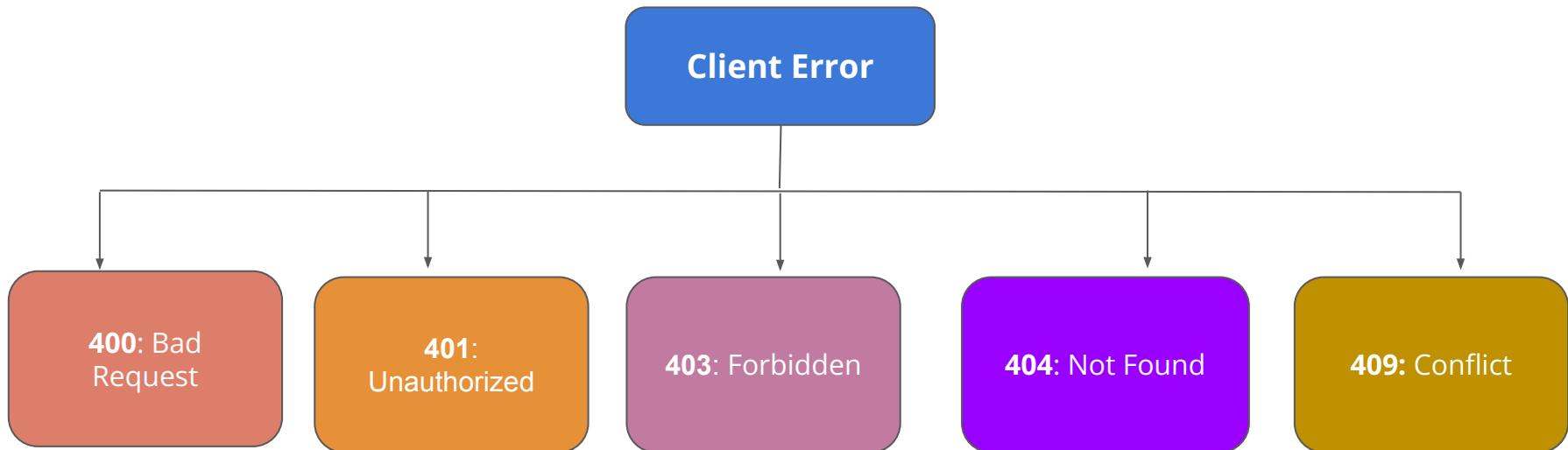
# Status Code: Success

Success status codes (200-299) indicate that the request was successfully processed by the server.



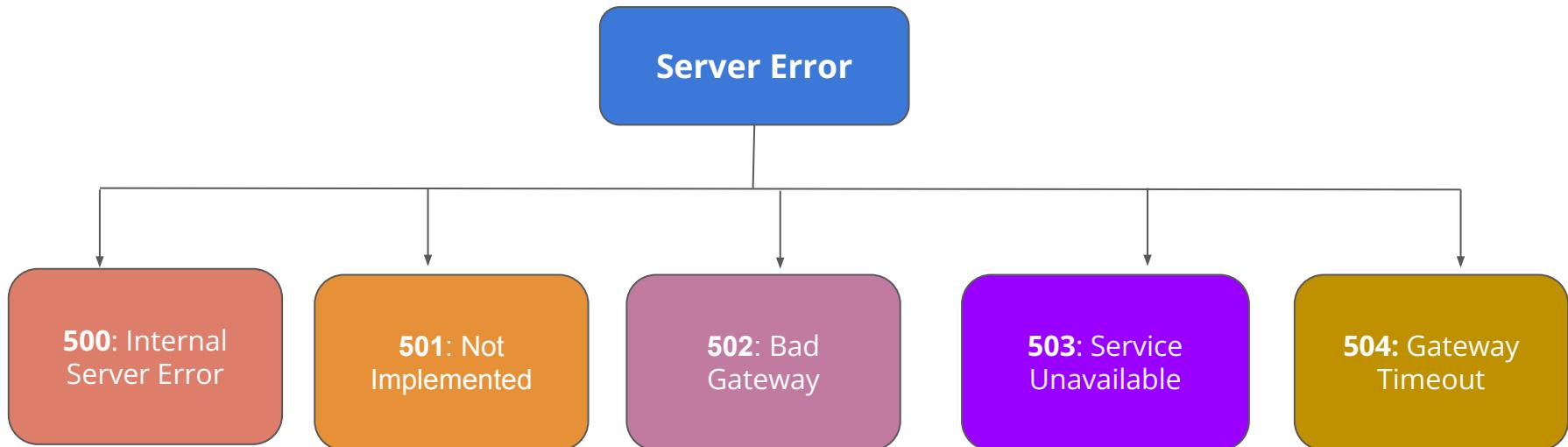
# Status Codes: Client Error

Client error status codes (400-499) indicate that the request was invalid or cannot be processed due to issues on the client side.



# Status Codes: Server Error

Client error status codes (400-499) indicate that the request was invalid or cannot be processed due to issues on the client side.



# In Class Questions

# References

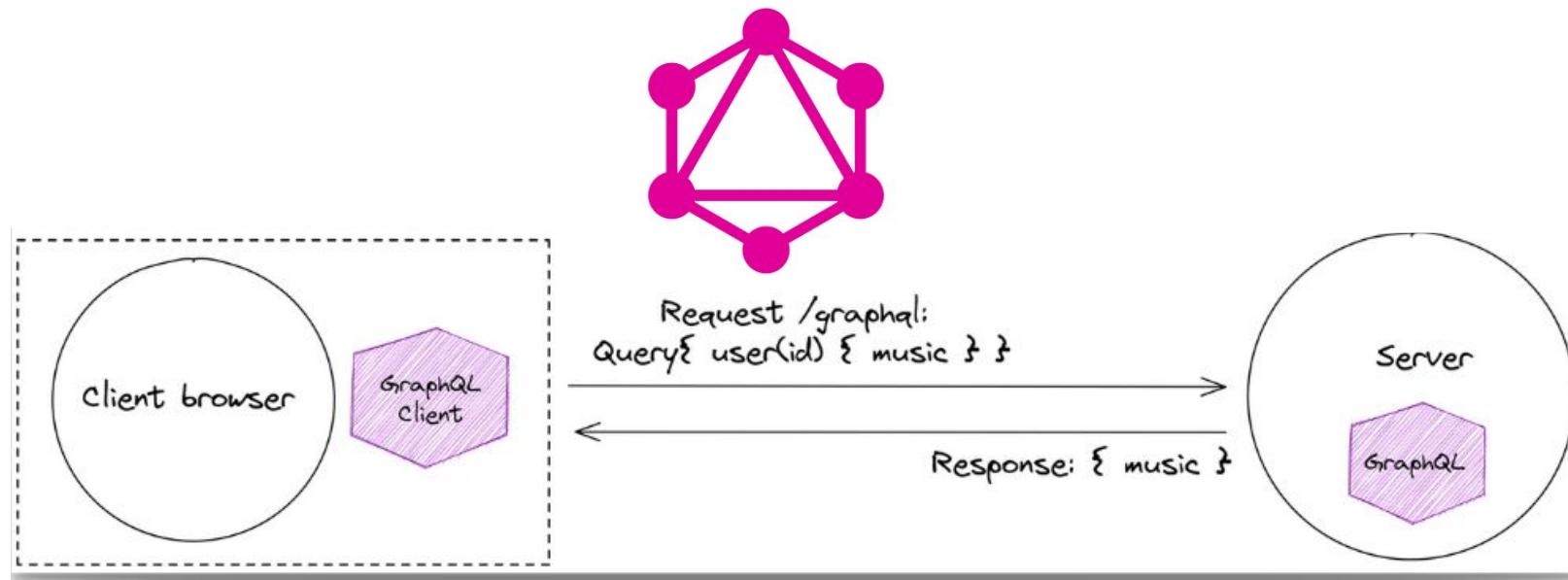
1. **MDN Web Docs - JavaScript:** Comprehensive and beginner-friendly documentation for JavaScript.  
<https://developer.mozilla.org/en-US/docs/Web/JavaScript>
2. **Eloquent JavaScript:** A free online book covering JavaScript fundamentals and advanced topics.  
<https://eloquentjavascript.net/>
3. **JavaScript.info:** A modern guide with interactive tutorials and examples for JavaScript learners.  
<https://javascript.info/>
4. **freeCodeCamp JavaScript Tutorials:** Free interactive lessons and coding challenges to learn JavaScript.  
<https://www.freecodecamp.org/learn/>

**Thanks  
for  
watching!**



# GraphQL API: Querying the database

GraphQL is a query language for APIs that lets clients request exactly the data they need, reducing over-fetching or under-fetching of data.



# GraphQL API

Let's have a look at an example:-

```
{  
  "user": {  
    "id": "1",  
    "name": "John Doe",  
    "email": "john.doe@example.com",  
    "age": 30,  
    "address": {  
      "street": "123 Main St",  
      "city": "New York",  
      "country": "USA"  
    },  
    "posts": [  
      { "id": "101", "title": "GraphQL Basics" },  
      { "id": "102", "title": "Advanced GraphQL" }  
    ]  
  }  
}
```

The server stores the entire dataset, but *you only need "name" and "email"*.

Unlike other APIs that overfetch by retrieving entire data units, ***GraphQL lets you fetch exactly what you need***, avoiding unnecessary data.

# GraphQL API: Sending Request Query

Instead of fetching everything, you can query only what you need:



```
1  query {  
2      user(id: "1") {  
3          name  
4          email  
5      }  
6  }
```



Query you perform



```
1  {  
2      "data": {  
3          "user": {  
4              "name": "John Doe",  
5              "email": "john.doe@example.com"  
6          }  
7      }  
8  }
```

Result you get

# GraphQL API: Sending Request Query

Instead of fetching everything, you can query only what you need:



```
1  query {  
2      user(id: "1") {  
3          name  
4          email  
5      }  
6  }
```



Query you perform



```
1  {  
2      "data": {  
3          "user": {  
4              "name": "John Doe",  
5              "email": "john.doe@example.com"  
6          }  
7      }  
8  }
```

Result you get