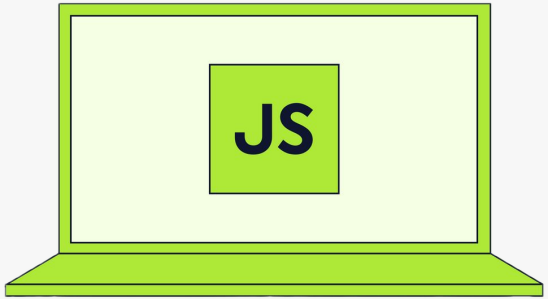
Newton School of Technology

The Complete Javascript Course



@newtonschool

Lecture 11: Understanding APIs

-Narendra Kumar

JS

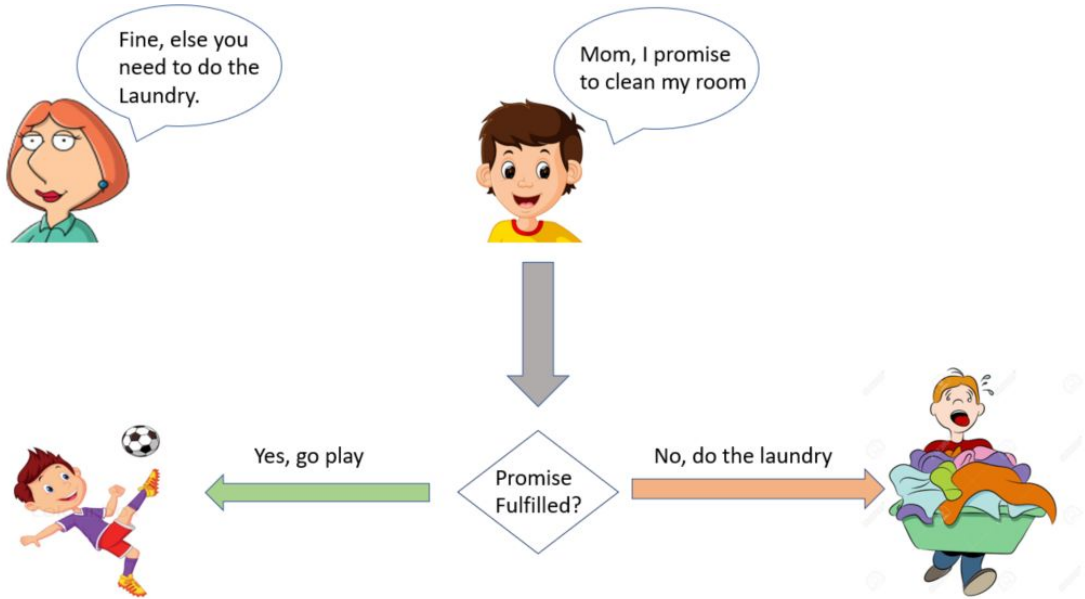
Table of Contents

Newton School of Technology

Quick Recap

Pre-discussed: Making a Promise

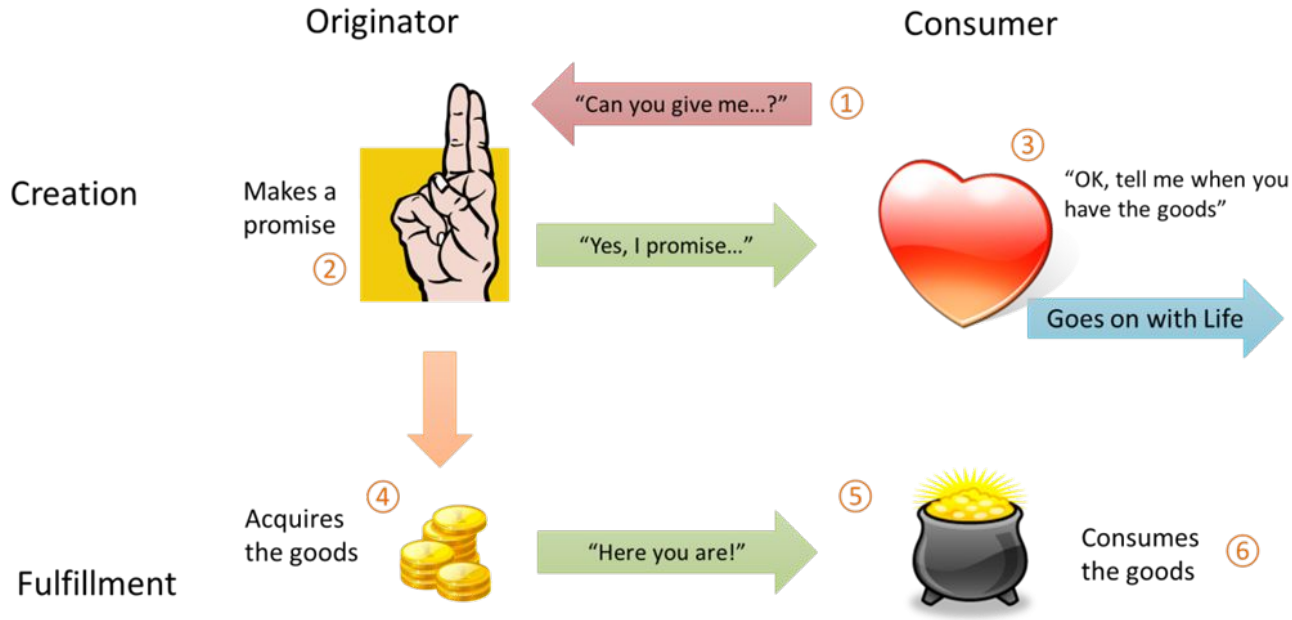
If you did then you must know how valuable promises are. A promise in real life is a commitment or guarantee made by someone.



Sadly, few promises remain unfulfilled.

Pre-discussed: Stages of promise

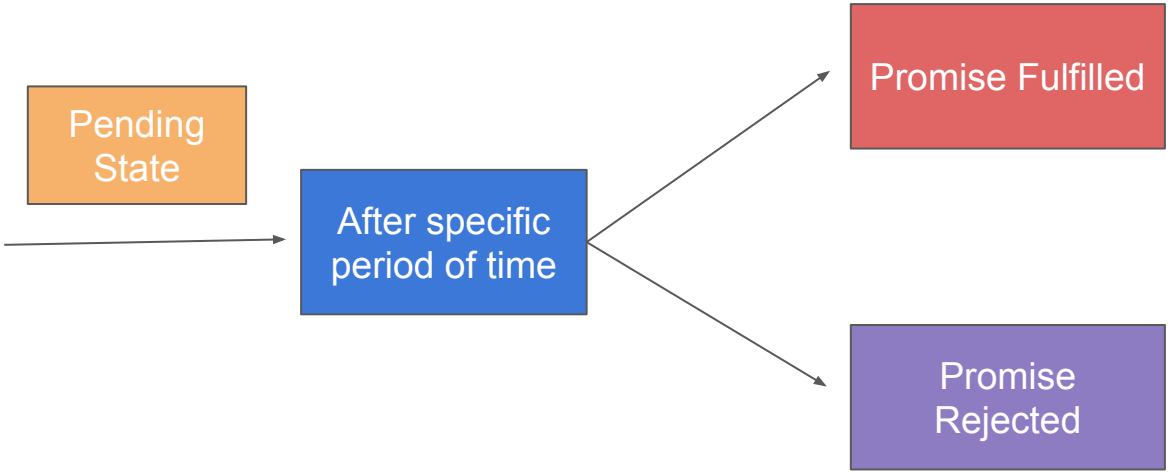
If you did then you must know how valuable promises are. A promise in real life is a commitment or guarantee made by someone.



Pre-discussed: Promises in Javascript

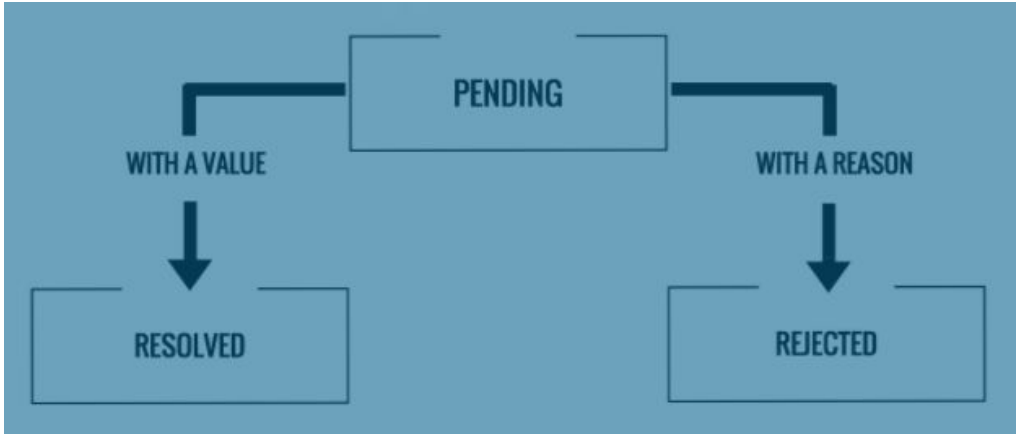
Newton School of Technology

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

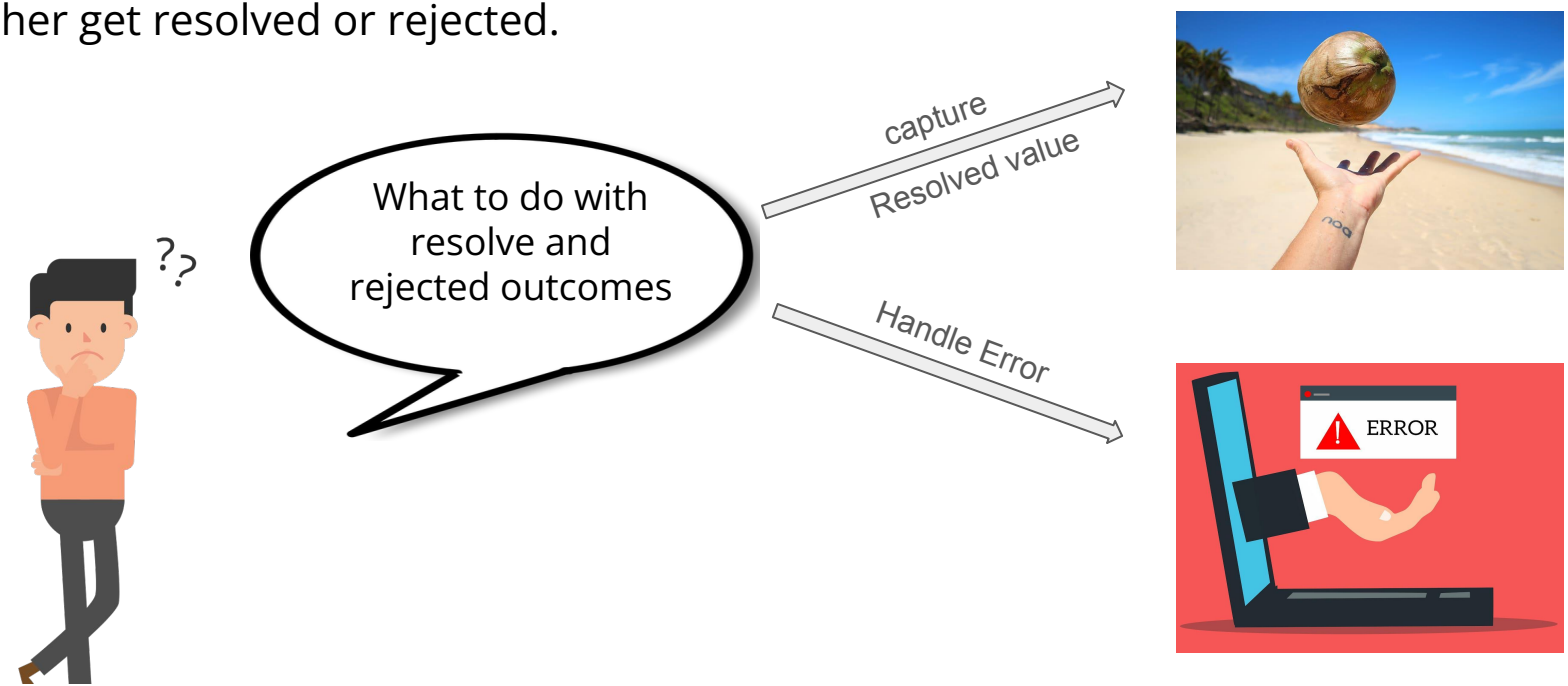
Promises in javascript exists in three states namingly, pending, fulfilled, and rejected.



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

Pre-discussed: Handling Promise

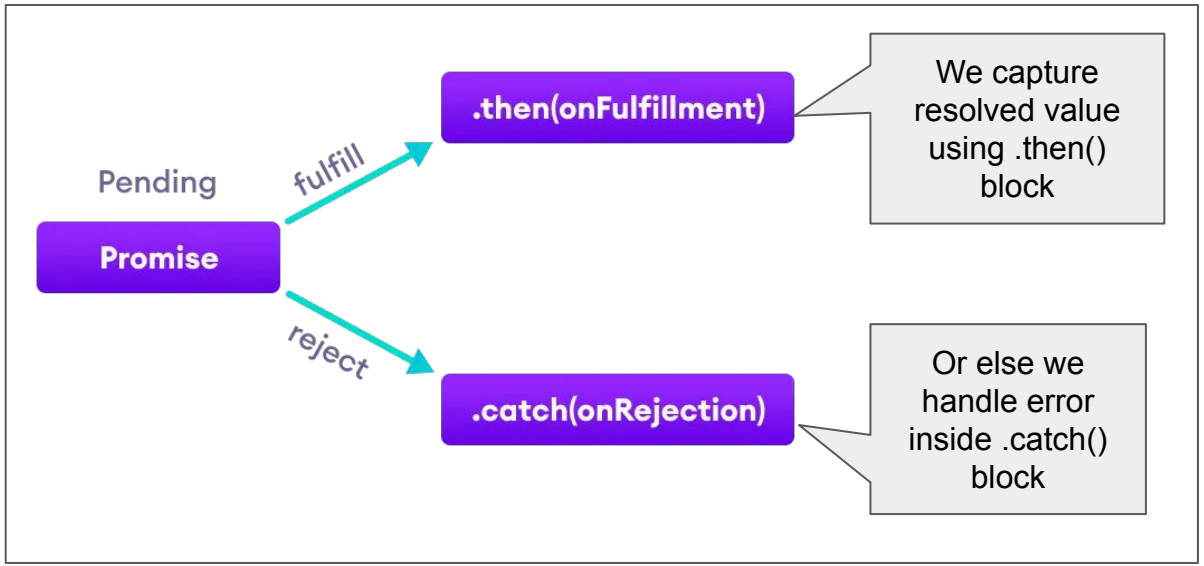
As we have learned that promise is initially at pending state and then after a while it either get resolved or rejected.



Pre-discussed: Handling Promise

Newton School of Technology

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.



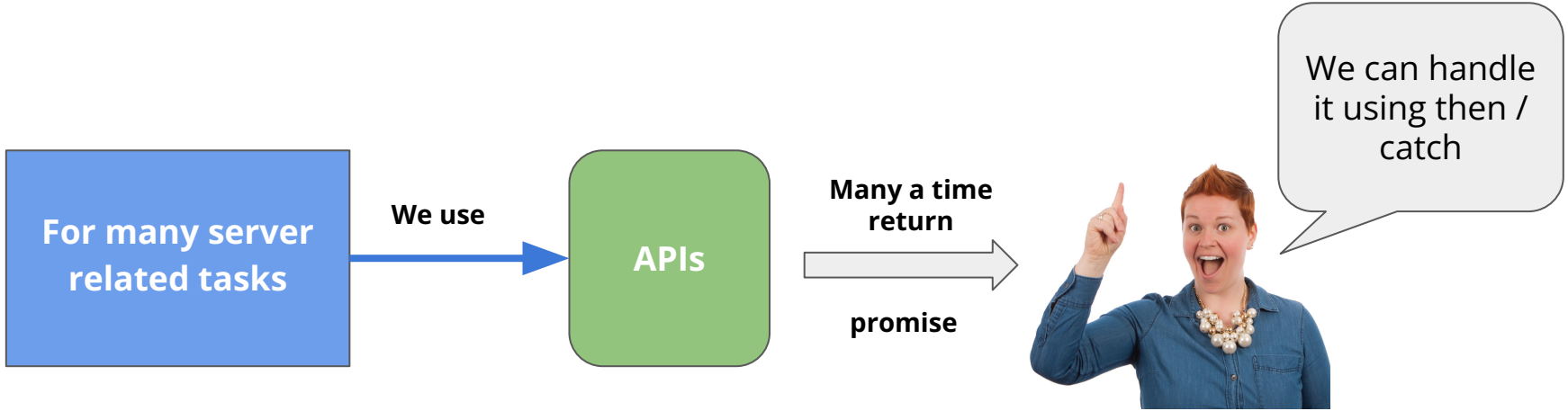
We can add finally at the end(optional)

.finally()

Finally runs, regardless of the promise's result.

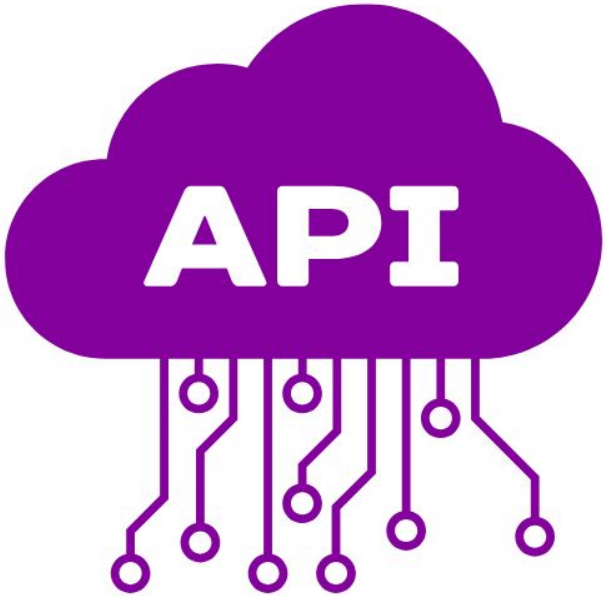
Promises in APIs

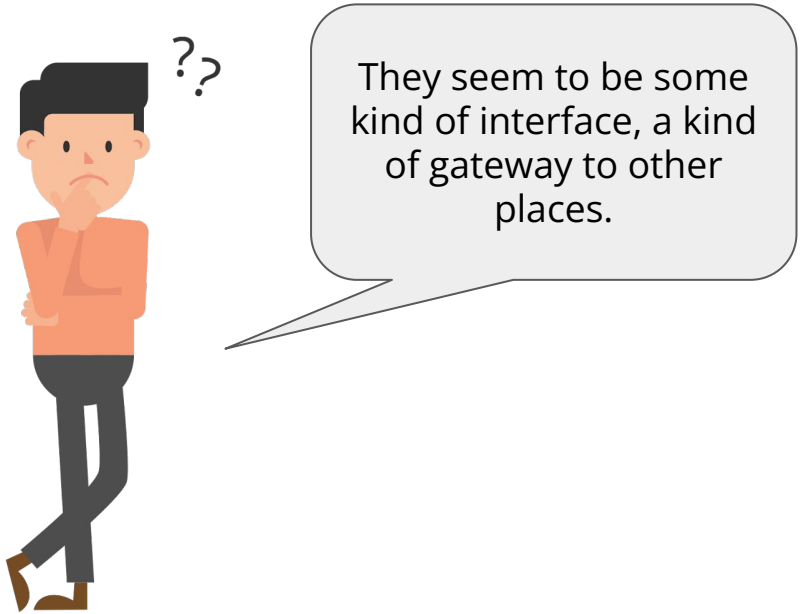
Applications often fetch external data using APIs. Since APIs return data asynchronously, they often return Promises.



But What they are exactly?

Before we jump into how to handle promises returned by APIs lets understand what they are.



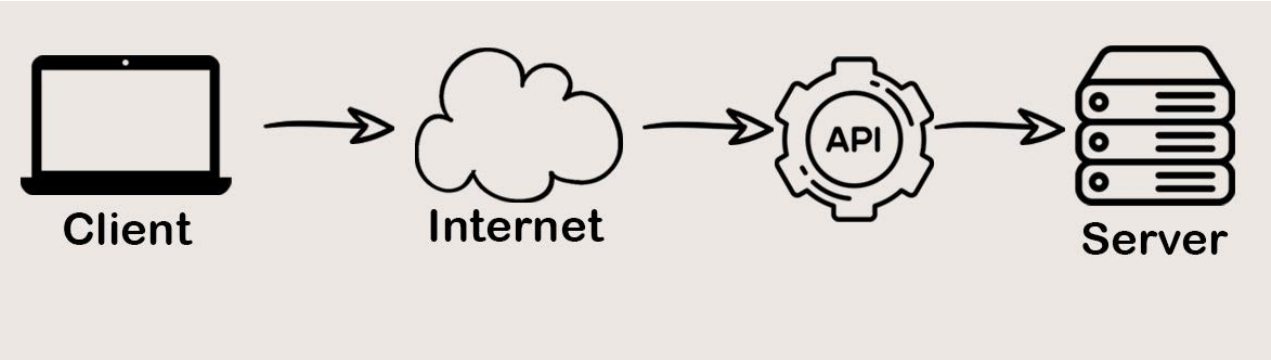


Newton School of Technology

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 talk to each other.

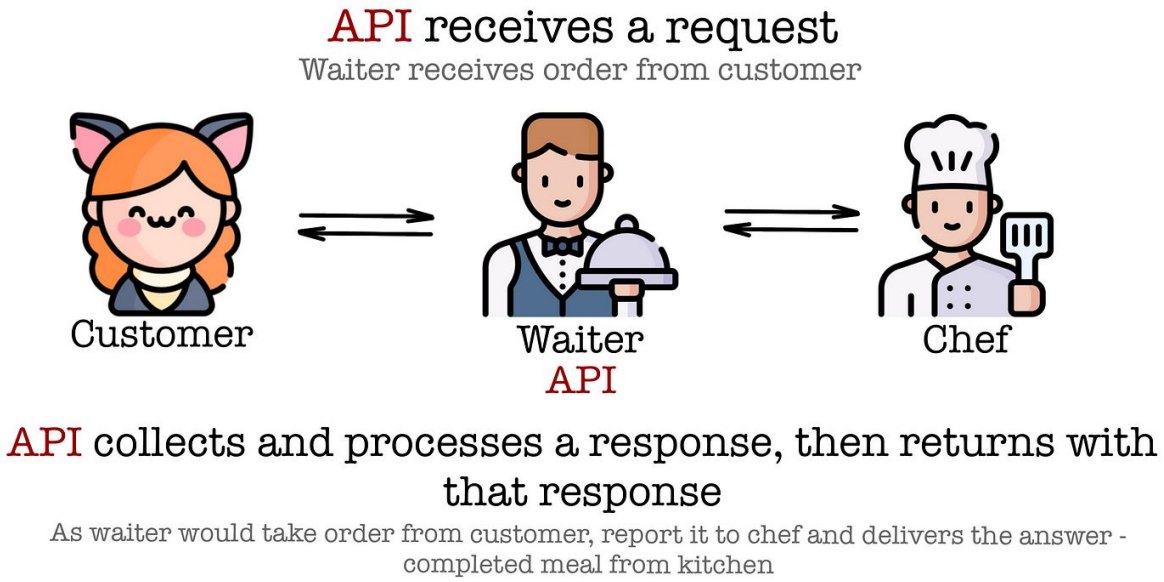


Understood??

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),



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

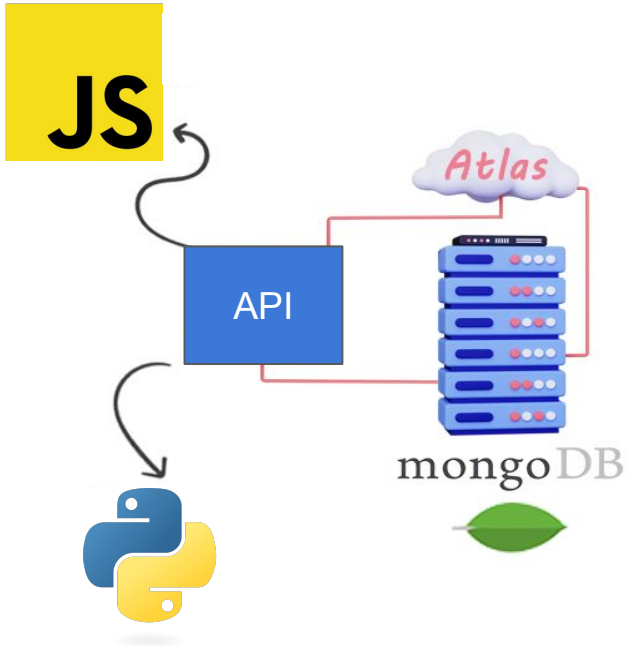
Newton School of Technology

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

But Why Not Communicate Directly?

Newton School of Technology

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

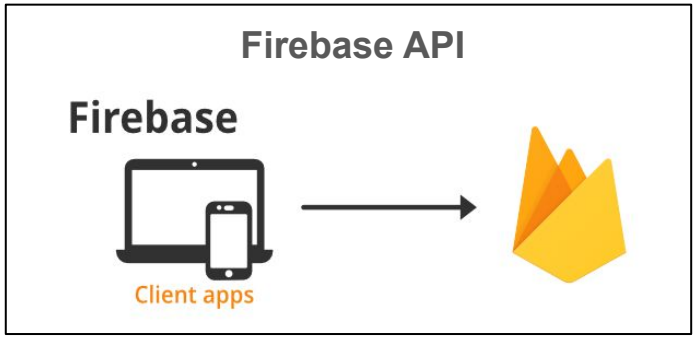


Services like MongoDB and software like ChatGPT provide APIs that allow different languages to communicate efficiently.

Popular Examples of APIs in Action

Let’s have a look at few popular apis:-

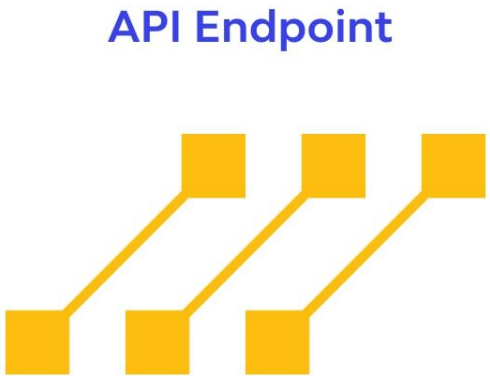
Remember how often we used the Fetch API in previous lectures

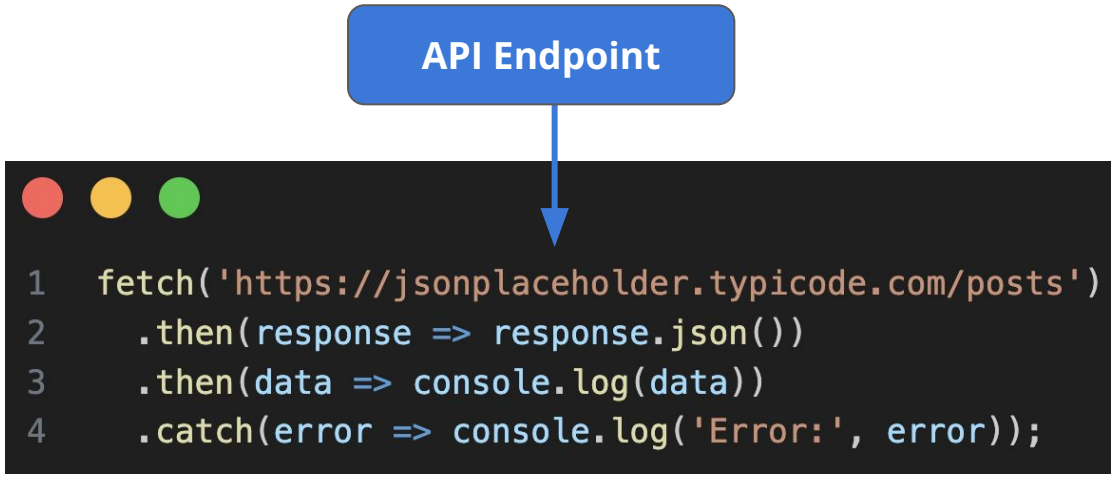


API Endpoints: How to Access APIs?

Newton School of Technology

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

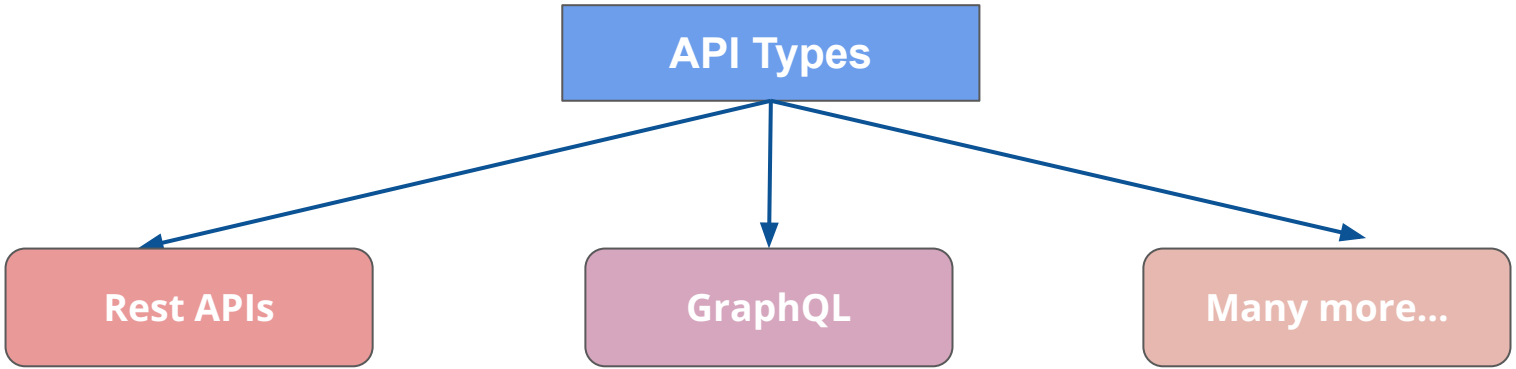




Types of APIs

Newton School of Technology

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



Newton School of Technology

Rest API

Understanding with fetch()

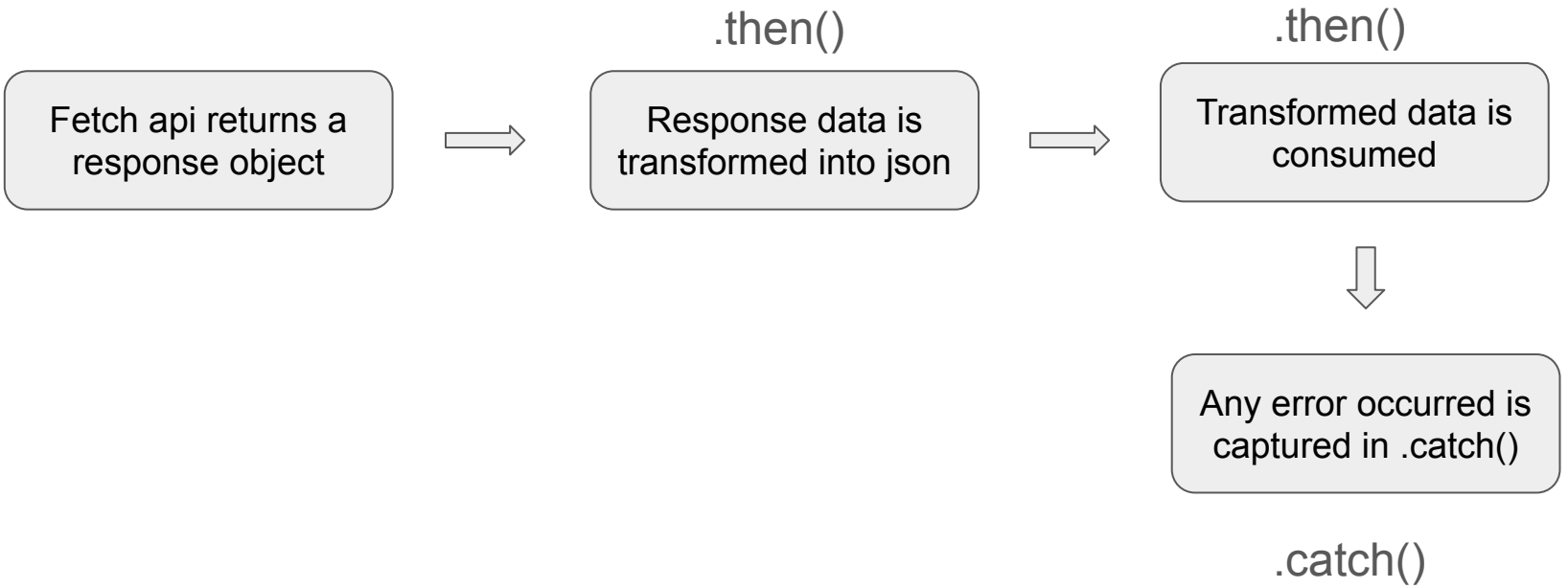
Rest API

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 simple http request 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

Newton School of Technology

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

```
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   });
```

If resolves returns response data else throws an error

Response data is transformed to json object

Transformed data is consumed

If any error then gets caught in .catch()

Settling fetch API: then/catch

Newton School of Technology

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 a 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

Newton School of Technology

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

.then() chaining, values returned from previous gets 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

Newton School of Technology

The received data from the fetch is in form of 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

Newton School of Technology

The received data from the fetch is in form of 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

Newton School of Technology

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()

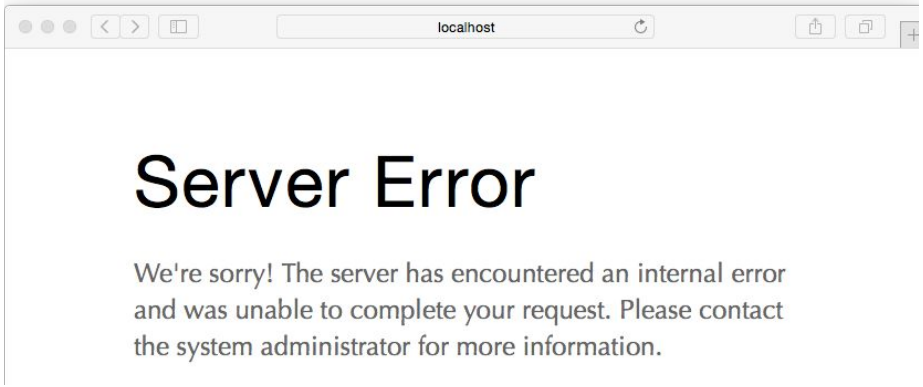
then/catch Workshop

Workshop: Question 1

Newton School of Technology

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 you would handle error using catch()?

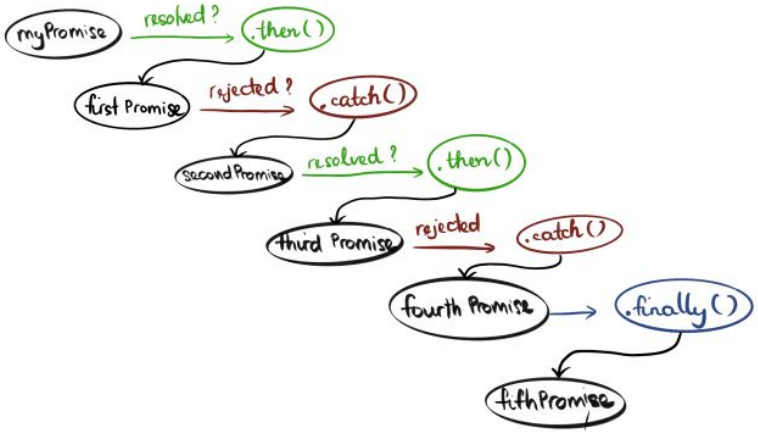


Workshop: Question 2

Newton School of Technology

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

Newton School of Technology

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

fetch() - GET method

Newton School of Technology

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. 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 would retrieve data from the API endpoint.

The response is typically in JSON format, and we can process it using .json().

fetch() - POST method

Newton School of Technology

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

Newton School of Technology

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

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

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

And need to supply the value to update.

fetch() - PUT method

Newton School of Technology

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

Newton School of Technology

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

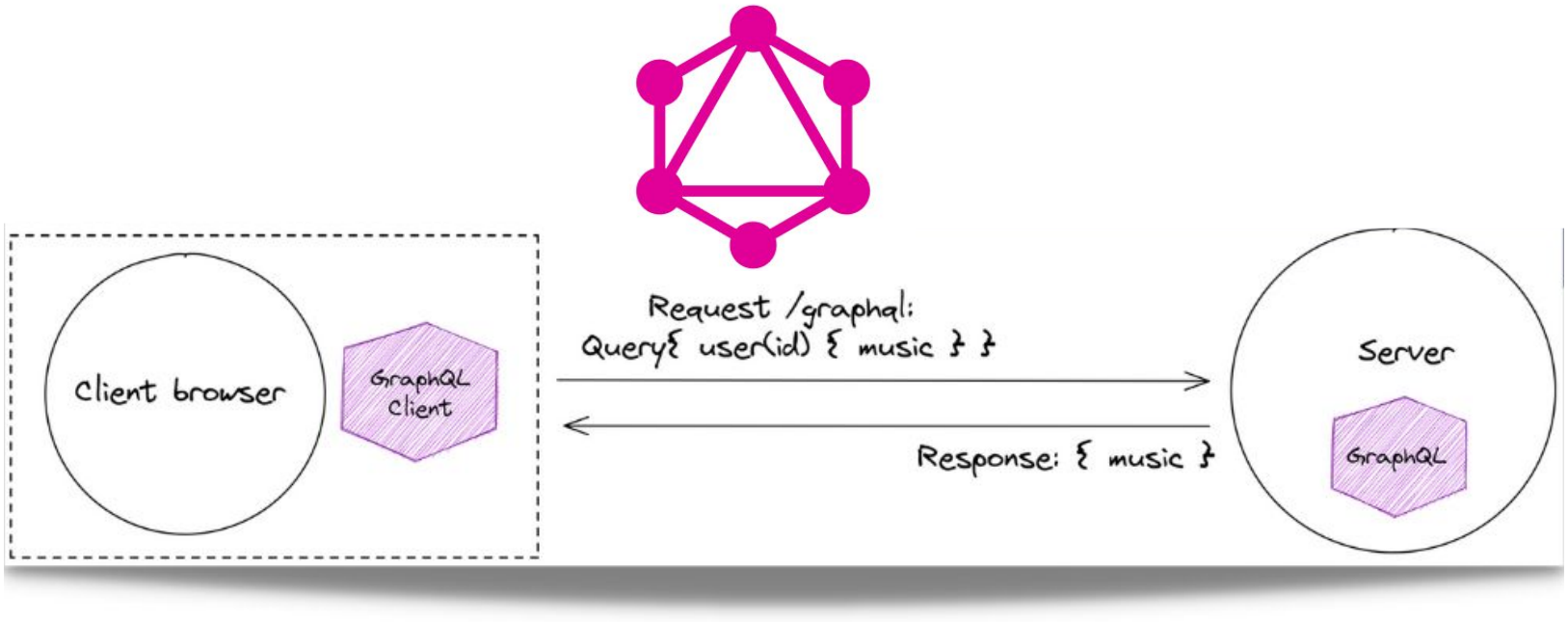
```
1  fetch('https://jsonplaceholder.typicode.com/posts/1', {
2      method: 'DELETE'
3  })
4      .then(response => response.json())
5      .then(data => console.log('Deleted:', data))
6      .catch(error => console.log('Error:', error));
```

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

GraphQL API: Querying the database

Newton School of Technology

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

Newton School of Technology

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.

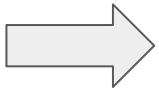
GraphQL API: Sending Request Query

Newton School of Technology

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

```
query {
    user(id: "1") {
      name
      email
    }
}
```

Query you perform



```
{
    "data": {
      "user": {
        "name": "John Doe",
        "email": "john.doe@example.com"
      }
    }
}
```

Result you get

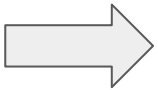
GraphQL API: Sending Request Query

Newton School of Technology

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

```
query {
    user(id: "1") {
      name
      email
    }
}
```

Query you perform



```
{
    "data": {
      "user": {
        "name": "John Doe",
        "email": "john.doe@example.com"
      }
    }
}
```

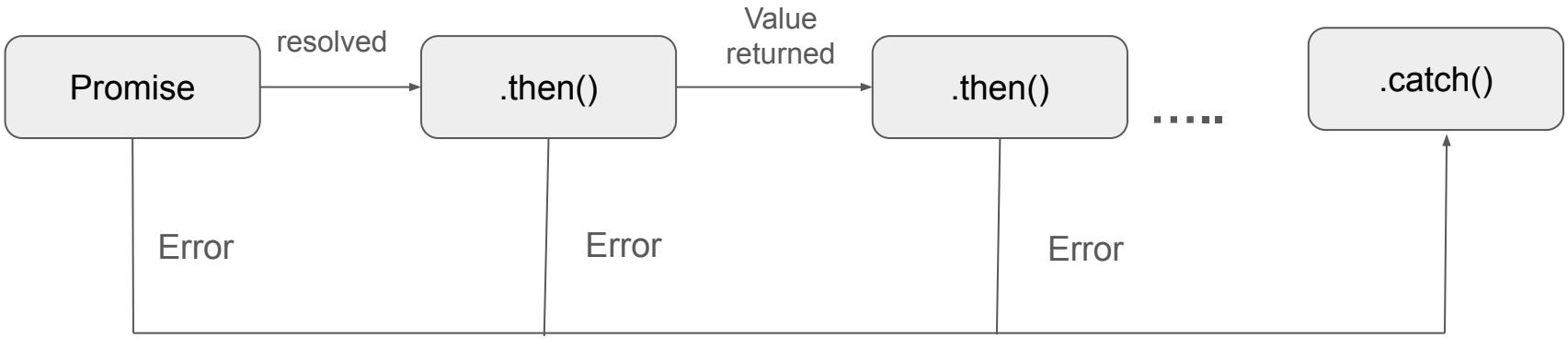
Result you get

Newton School of Technology

Error Handling

Error Handling: Catch block

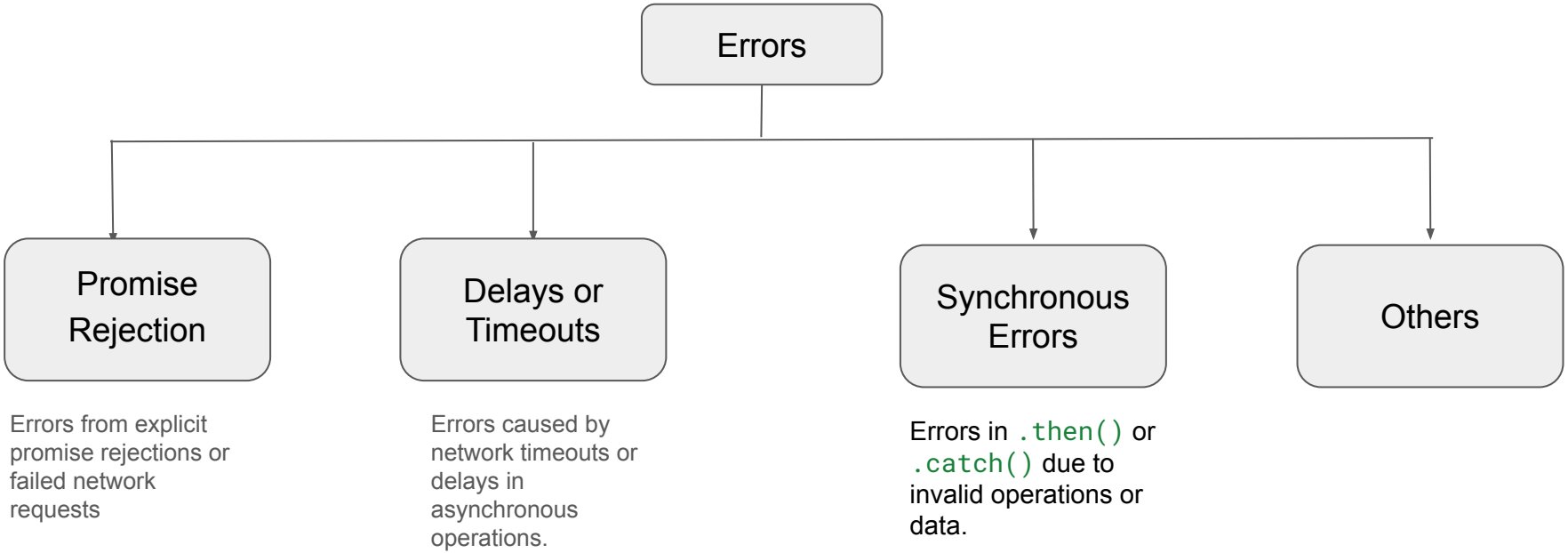
The `.catch()` block is used to handle any errors that occur during the execution of a promise or any promise chain. It catches both rejections and exceptions that occur in the promise lifecycle.



Handling Errors

Newton School of Technology

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



Promise Rejection

Newton School of Technology

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

Newton School of Technology

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

```
let myPromise = new Promise((resolve, reject) => {
    reject('Something went wrong');
});

  myPromise
    .catch(error => {
      console.error('Caught error:', error);
    });
```

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

Promise Rejection: Network Http failures

Newton School of Technology

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

```
fetch('https://invalidurl.com')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network error');
    }
    return response.json();
  })
  .catch(error => {
    console.error('Network error:', error);
  });
```

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

Newton School of Technology

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

```
let timeoutPromise = new Promise((resolve, reject) => {
    setTimeout(() => reject('Operation timed out'), 5000);
});

timeoutPromise.catch(error => {
    console.error('Timeout error:', error);
});
```

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

Newton School of Technology

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

```
let processData = new Promise((resolve, reject) => {
    let input = null;
    if (!input) {
      reject('Invalid data');
    }
});

processData.catch(error => {
    console.error('Data error:', error);
});
```

Invalid data

Handling Errors at different stages

Newton School of Technology

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

```
fetch('https://jsonplaceholder.typicode.com/posts')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network error: Failed to fetch posts');
    }
    return response.json(); // Parse the JSON data
  })
  .then(posts => {
    return fetch('https://jsonplaceholder.typicode.com/comments')
      .then(response => {
        if (!response.ok) {
          throw new Error('Network error: Failed to fetch comments');
        }
        return response.json(); // Parse the comments
      })
      .catch(commentError => {
        console.error('Error in fetching comments:', commentError);
        throw new Error('Comments stage failed');
      });
  })
  .catch(error => {
    // General error handling for any unhandled errors in chain
    console.error('General error occurred:', error);
  });
```

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

Newton School of Technology

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.

```
fetch('https://jsonplaceholder.typicode.com/posts')
.then(response => {
  if (!response.ok) {
    throw new Error('Network error: Failed to fetch posts');
  }
  return response.json();
})
.then(data => {
  console.log('Fetched posts:', data);
})
.catch(error => {
  console.error('Error occurred:', error);
  // Fallback: Provide default data or retry
  const fallbackData = [{ id: 1, title: 'Fallback post' }];
  console.log('Using fallback data:', fallbackData);
});
```

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

Newton School of Technology

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

```
function fetchWithRetry(url, retries = 3) {
    return fetch(url)
      .then(response => {
        if (!response.ok) {
          throw new Error('Network error');
        }
        return response.json();
      })
      .catch(error => {
        if (retries > 0) {
          console.log(`Retrying... Attempts left: ${retries}`);
          return fetchWithRetry(url, retries - 1); // Retry
        } else {
          console.error('Max retries reached');
          throw error; // Fail after retries
        }
      });
}

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

Newton School of Technology

Or else we can graceful show minimal or different content

```
fetch('https://jsonplaceholder.typicode.com/posts')
  .then(response => response.json())
  .then(data => {
    // Process data
  })
  .catch(error => {
    console.error('Showing limited content');
    // Show minimal or default content
    displayMinimalContent();
  });
```

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

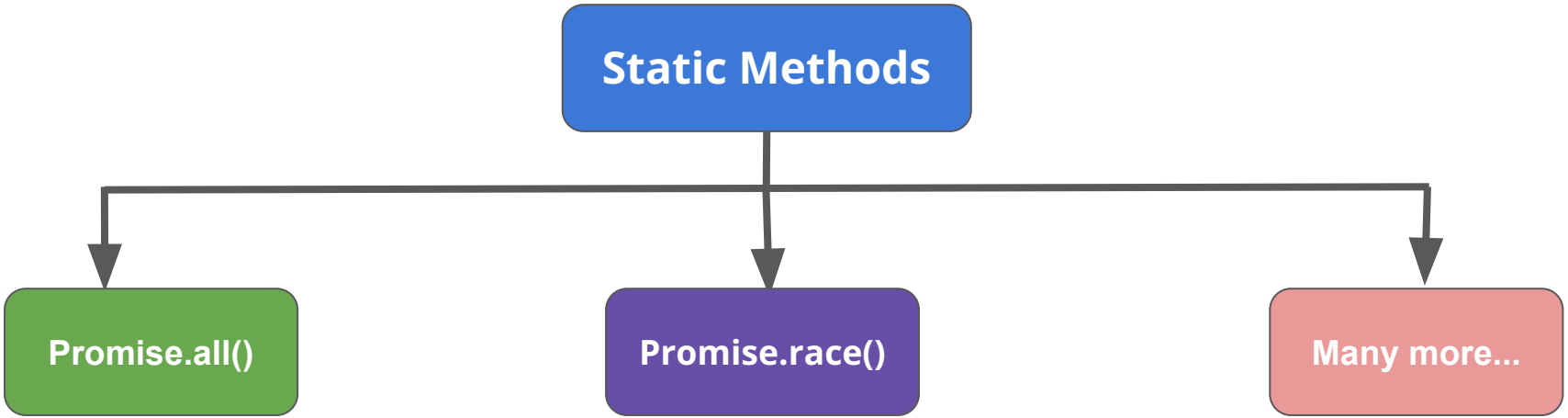
Newton School of Technology

Promise: Static Methods

Promise: What are Static methods?

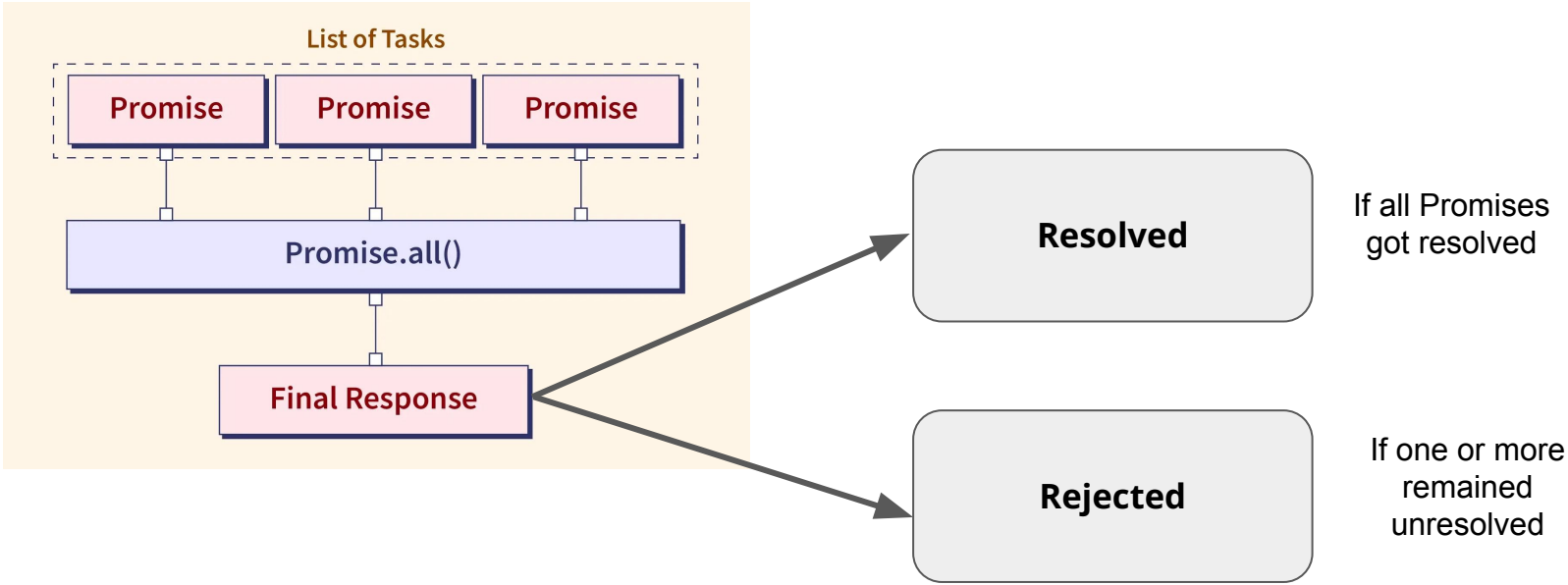
Newton School of Technology

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   });
```

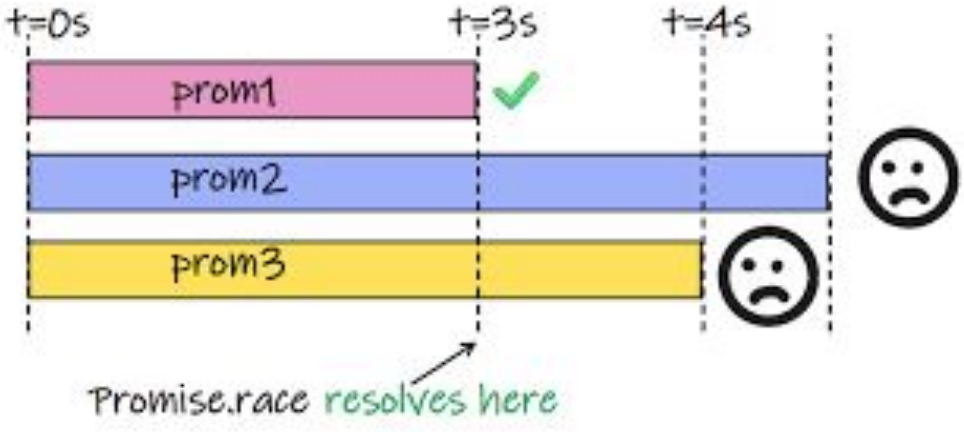
This promise is being rejected

So instead an error would be thrown

And error gets captured by catch block

Static Method: Promise.race()

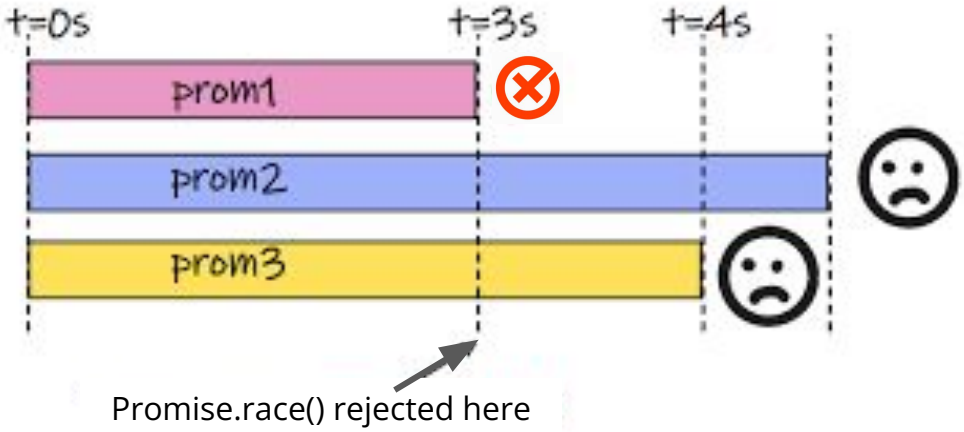
Promise.race() is a static method that takes an array of promises and returns a single promise, which resolves or rejects as soon as one of the promises settles.



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

Newton School of Technology

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()

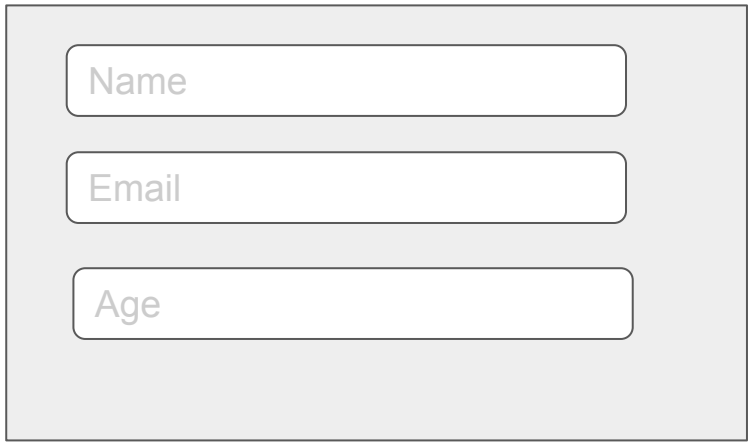
Workshop

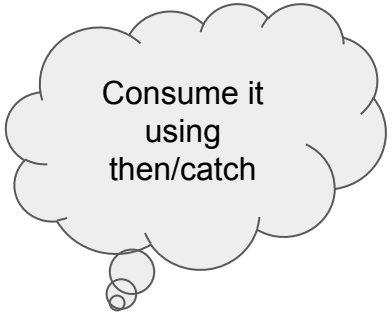
Workshop: Question

Newton School of Technology

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?





Workshop: Solution

Newton School of Technology

Let’s consume the promise:-

```
// Function to handle form submission
function handleFormSubmission(name, email, age) {
    // Validate all fields using promises and consume them
    Promise.all([
      validateName(name),
      validateEmail(email),
      validateAge(age)
    ])
    .then((validationResults) => {
      // All validations passed, process the form
      console.log('Form submission successful!');
      console.log(validationResults);
      // You can log validation results if needed
      // Proceed to save data or display success message
    })
    .catch((error) => {
      // Handle the first validation error that occurs
      console.log('Form submission failed:', error);
    });
}
```

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

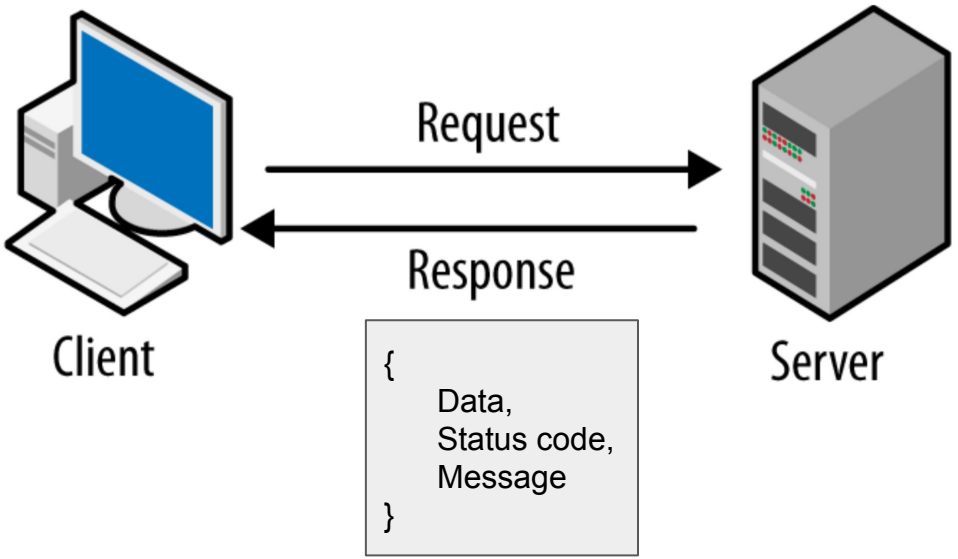
Newton School of Technology

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

Newton School of Technology

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.

```
fetch('https://jsonplaceholder.typicode.com/posts')
    .then(response => {
      console.log('Status Code:', response.status);
      // Retrieves and logs the status code
      return response.json();
      // Process the response body if needed
    })
    .then(data => {
      console.log('Data:', data);
    })
    .catch(error => {
      console.log('Error:', error);
    });
```

Here we are capturing status code

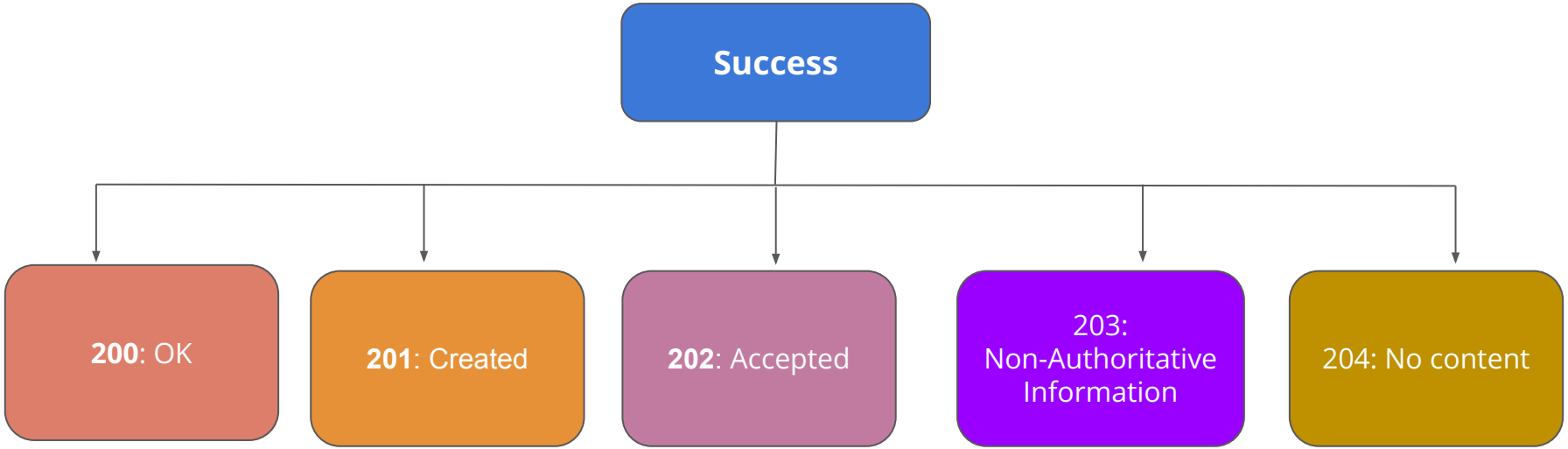
We can use some other variable let say *data*, then we get status code using:-

data.status

Status Code: Success

Newton School of Technology

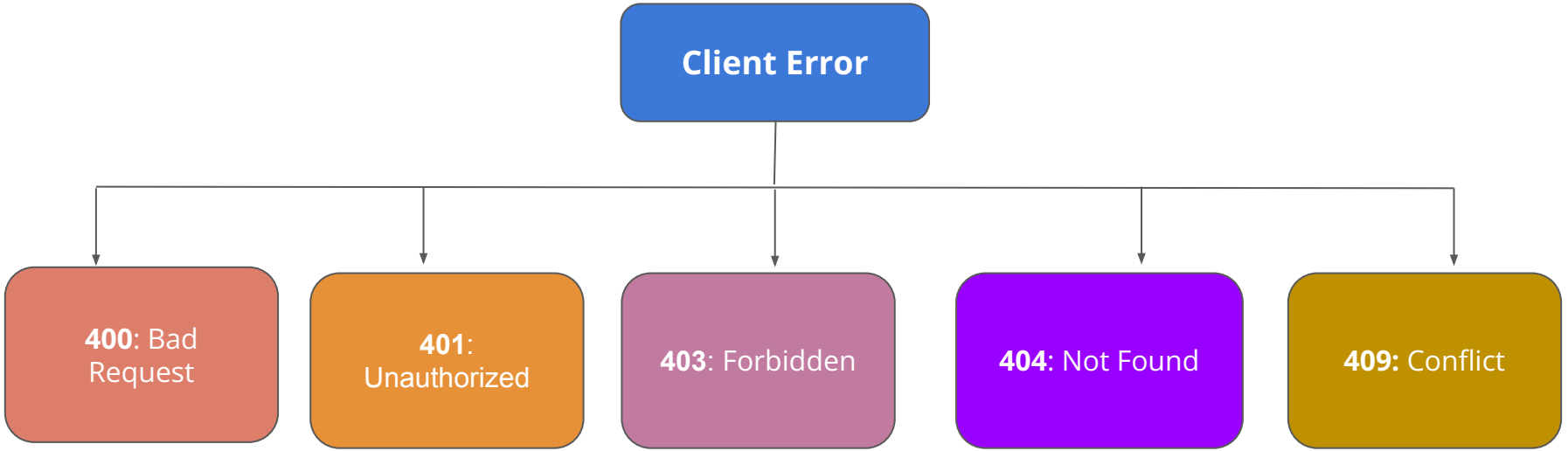
Success status codes (200-299) indicate that the request was successfully processed by the server.



Status Codes: Client Error

Newton School of Technology

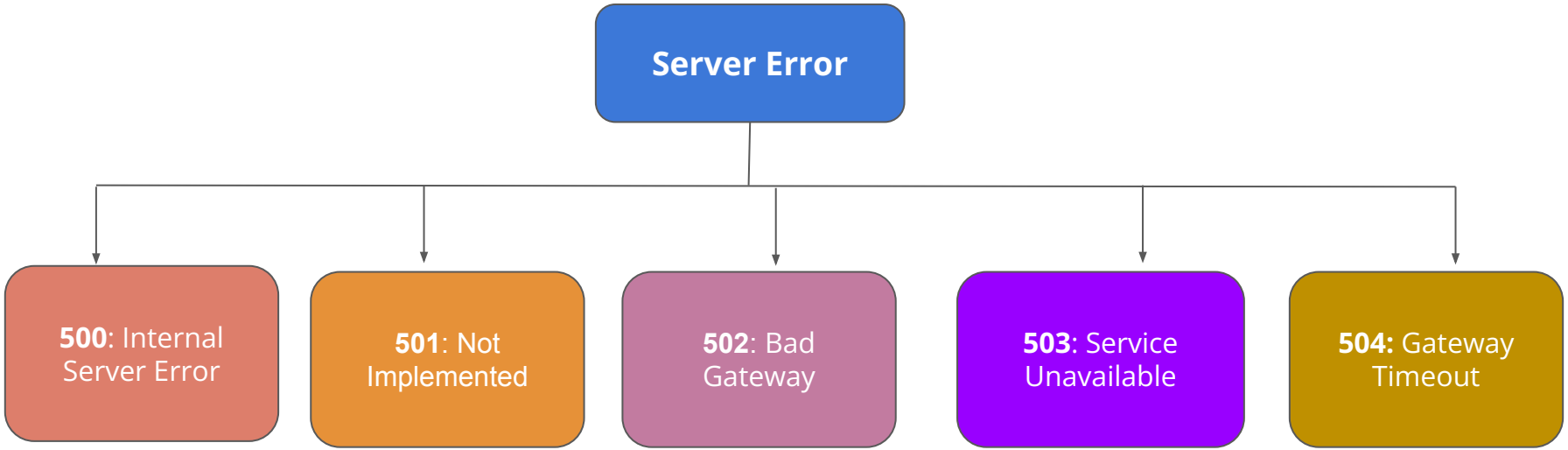
Client error status codes (400-499) indicate that the request was invalid or cannot be processed due to issues on the client side.



Status Codes: Server Error

Newton School of Technology

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

Newton School of Technology

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/

Newton School of Technology

Thanks for watching!