AngularJS

Table of Content

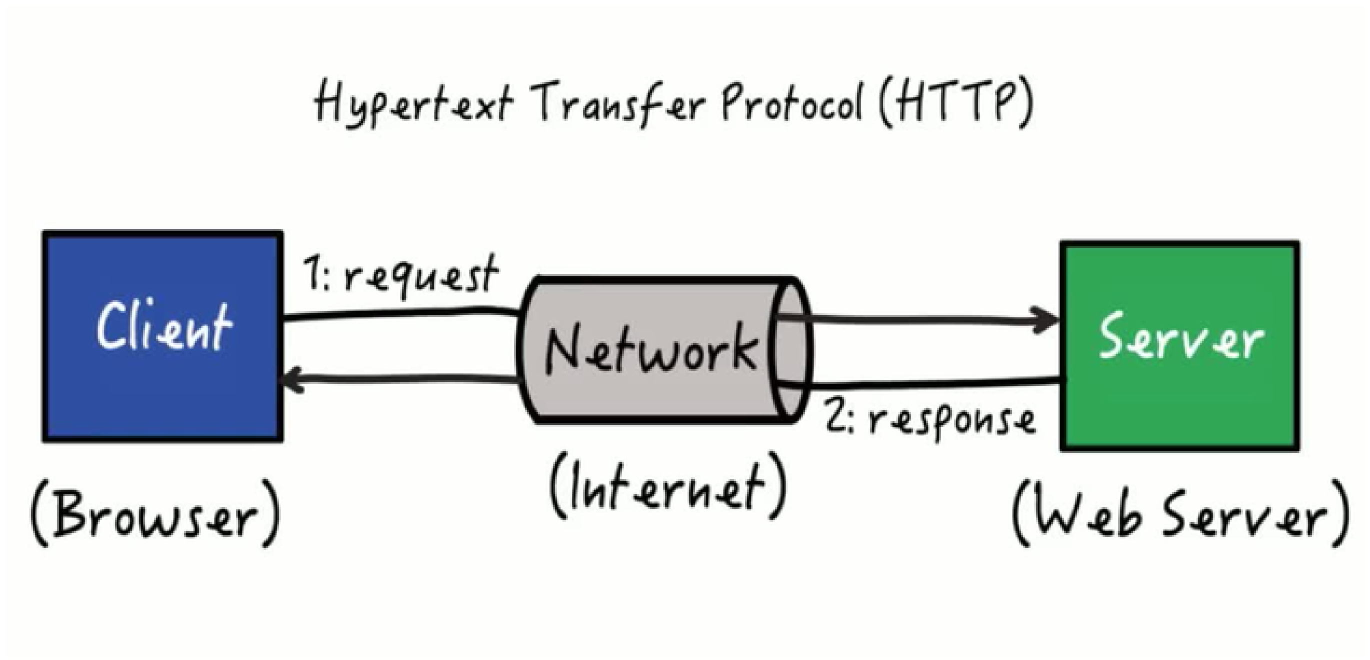
Web Services Overview and Introduction to Node.js

Introduction to HTTP

What is HTTP

- HTTP stands for Hyper Text Transfer Protocol
- The primary function of HTTP is to establish a connection with the server and send HTML pages back to the user's browser.
- HTTP is an application protocol that runs on top of the TCP/IP suite of protocols, which forms the foundation of the internet.

TCP/IP - Transmission Control Protocol/Internet Protocol



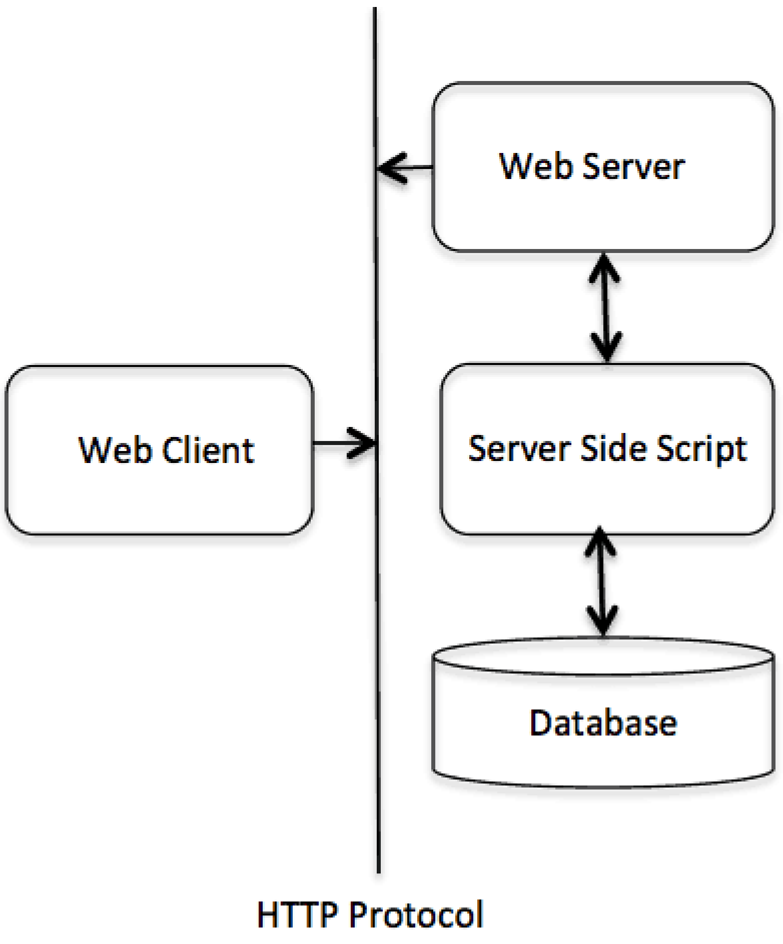
Features of HTTP

There are three basic features of HTTP

- **HTTP is connectionless** : The HTTP client, i.e., a browser initiates an HTTP request and after a request is made, the client waits for the response. The server processes the request and sends a response back after which client disconnect the connection. So client and server knows about each other during current request and response only. Further requests are made on new connection like client and server are new to each other.
- **HTTP is media independent**: It means, any type of data can be sent by HTTP as long as both the client and the server know how to handle the data content. It is required for the client as well as the server to specify the content type using appropriate MIME-type.
- **HTTP is stateless:** As mentioned above, HTTP is connectionless and it is a direct result of HTTP being a stateless protocol. The server and client are aware of each other only during a current request. Afterwards, both of them forget about each other. Due to this nature of the protocol, neither the client nor the browser can retain information between different requests across the web pages.

HTTP Architecture

The following diagram shows a very basic architecture of a web application and depicts where HTTP sits:



The HTTP protocol is a request/response protocol based on the client/server based architecture where web browsers, robots and search engines, etc. act like HTTP clients, and the Web server acts as a server.

Client The HTTP client sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a TCP/IP connection.

Server The HTTP server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity meta information, and possible entity-body content.

Advantages of HTTP

1. **Addressing**: HTTP uses advanced scheme of addressing. It assigns IP address with recognizable names so that it can be identified easily in the World Wide Web. Compared to the standard procedure of IP address with a series of numbers, using this the public can easily engage with the internet.

2. **Flexibility** : Whenever there are additional capabilities needed by an application, HTTP has the capability to download extensions or plugins and display the relevant data. These can include Flash players and Acrobat reader.

3. **Security** : In HTTP each files is downloaded from an independent connection and then gets closed. Due to this no more than one single element of a webpage gets transferred. Therefore, the chance of interception during transmission is minimized here.

4. **Latency** : Only when the connection is established, the handshaking process will take place in HTTP. Hence, there will be no handshaking procedure following a request. This significantly reduces latency in the connection.

5. **Accessibility** : When the page is loaded for the first time, all of the HTTP pages gets stored inside the internet caches known as the page cache. Therefore, once the page is visited again, the content is loaded quickly.

Disadvantages of HTTP

1. **Data Integrity** : Since there are no any encryption methods used in HTTP, there are chances of someone altering the content. That is the reason why HTTP is considered to be an insecure method prone to data integrity.

2. **Data Privacy** : Privacy is another problem faced in a HTTP connection. If any hacker manages to intercept the request they can view all the content present in the web page. Besides that they can also gather confidential informations such as the username and the password.

3. **Server Availability** : Even if HTTP receives all the data that it needs, clients does not take measures to close the connection. Therefore, during this time period, server will not be present.

4. **Administrative Overhead** : For transmitting a web page, a HTTP needs to create multiple connections. This causes administrative overhead in the connection.

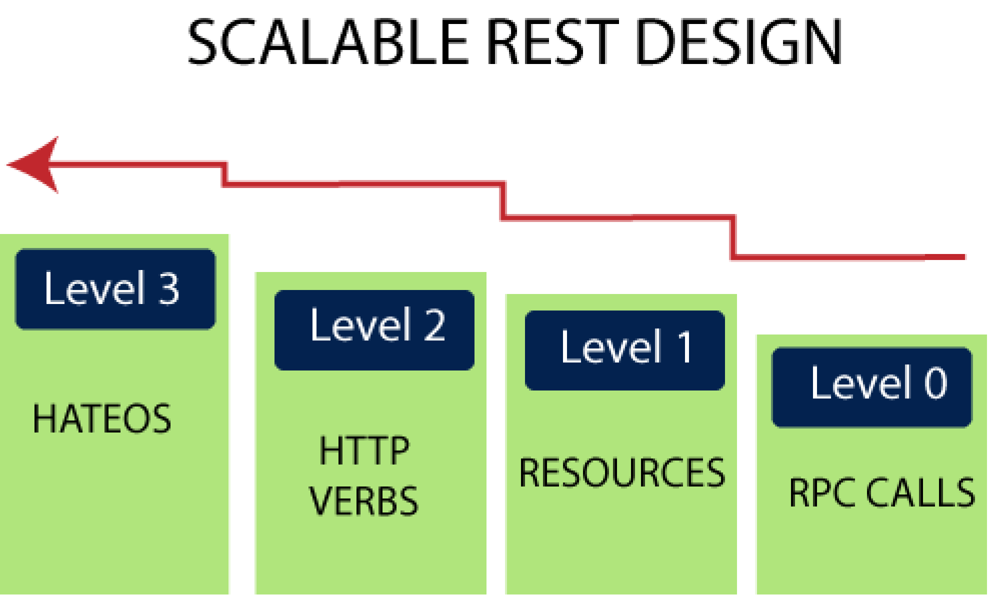
5. **IoT Device Support** : HTTP uses more number of system resources which leads to more power consumption. Since IoT device today contain wireless sensor networks, it is not suitable to use HTTP.

The Richardson Maturity Model

- Richardson Maturity Model is a model developed by Lenoard Richardson.
- It grades APIs by their RestFul maturity.
- It breaks down the principal element of the REST approach into four levels (0 to 3).

There are four levels:

- **Level 0: The Swamp of POX**
- **Level 1: Resources**
- **Level 2: HTTP Verbs**
- **Level 3: Hypermedia Control**



Level 0 The Swamp of POX

Level 0 is also often referred to as POX (Plain Old XML). At level 0, HTTP is used only as a transport protocol. For zero maturity level services, we use a single URI and a single HTTP method. We send a request to the same URI for obtaining and posting the data. Only the POST method can be used. for example, A particular company can have a lot of customers or users. We have only one endpoint for all the customers. All operations are performed via the POST method.

```
To get the data: POST http://localhost:8080/users
To post the data: POST http://localhost:8080/users
```

Level 1 Resources

In level 1 , each resource is mapped to a specific URI. However, only one HTTP method (POST) is used for retrieving and creating data. for example, we need to access the employees working in a company.

```
To add an employee to a particular department:
POST/department/<department-id>/employee
To access a specific employee :
POST/department/<department-id>/employee/<employee-id>
```

Level 2 HTTP Verbs

At Level 2 requests are sent with the correct HTTP verb. A correct HTTP response code is returned for each request.

For example: To get the users of the company, we send a request with the URI

```
http://localhost:8080/users and the server sends proper response 200 OK.
```

Level 3 Hypermedia Control

Level 3 is the highest level. It is the combination of level 2 and HATEOAS. It also provides support for HATEOAS. It is helpful in self-documentation.

For example, if we send a GET request for users, we will get a response for users in JSON format with self-documenting Hypermedia.

Why Web Services?

1. Exposing the Existing Function on the network

- A web service is a unit of managed code that can be remotely invoked using HTTP. That is, it can be activated using HTTP requests. Web services allow you to expose the functionality of your existing code over the network. Once it is exposed on the network, other applications can use the functionality of your program.

2. Interoperability

- Web services allow various applications to talk to each other and share data and services among themselves. Other applications can also use the web services. For example, a VB or .NET application can talk to Java web services and vice versa. Web services are used to make the application platform and technology independent.

3. Standardized Protocol

- Web services use standardized industry standard protocol for the communication. All the four layers (Service Transport, XML Messaging, Service Description, and Service Discovery layers) use well-defined protocols in the web services protocol stack. This standardization of protocol stack gives the business many advantages such as a wide range of choices, reduction in the cost due to competition, and increase in the quality.

4. Low Cost Communication

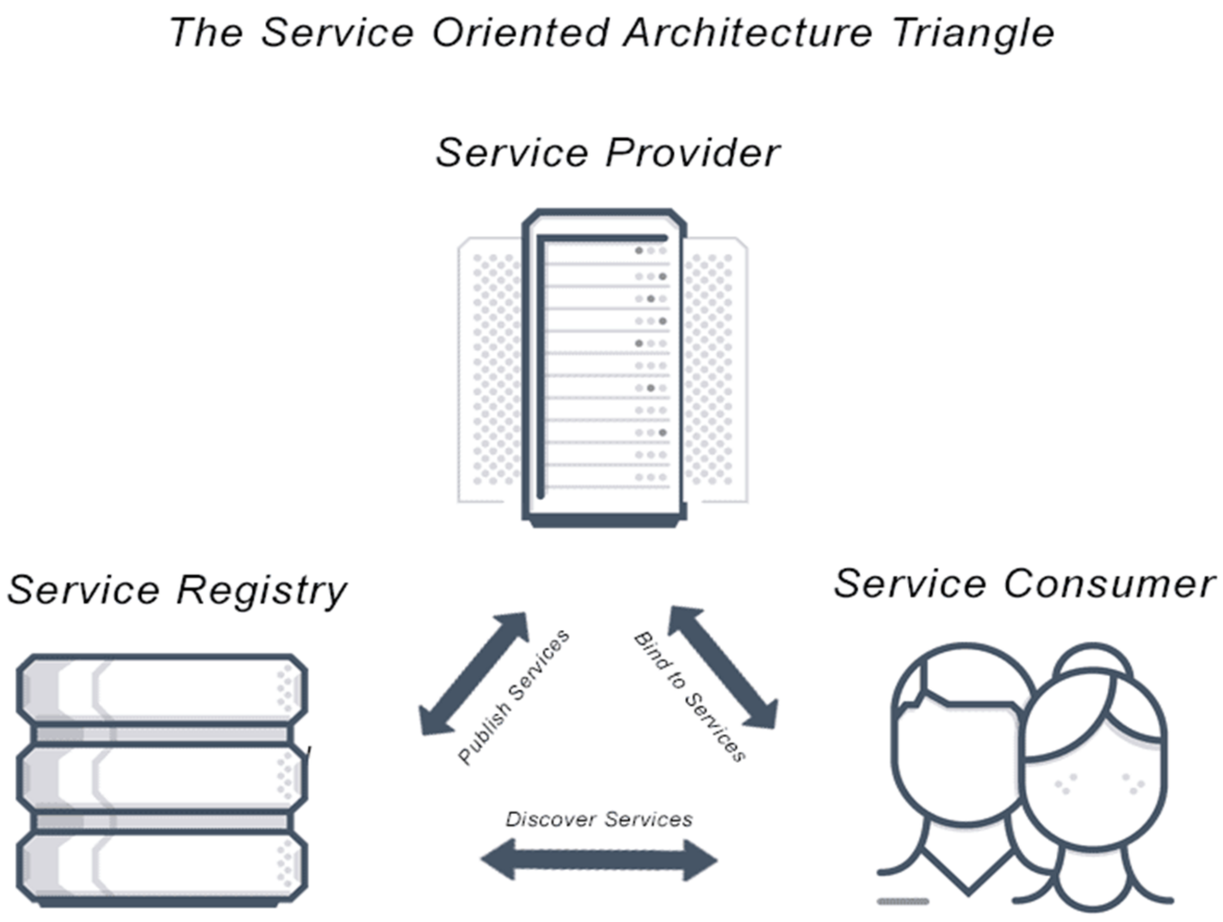
- Web services use SOAP over HTTP protocol, so you can use your existing low-cost internet for implementing web services. This solution is much less costly compared to proprietary solutions like EDI/B2B. Besides SOAP over HTTP, web services can also be implemented on other reliable transport mechanisms like FTP.

Service Oriented Architecture

- **What is Service Oriented Architecture**
- **What are the benefits SOA?**
- **Example of SOA?**

What is Service Oriented Architecture

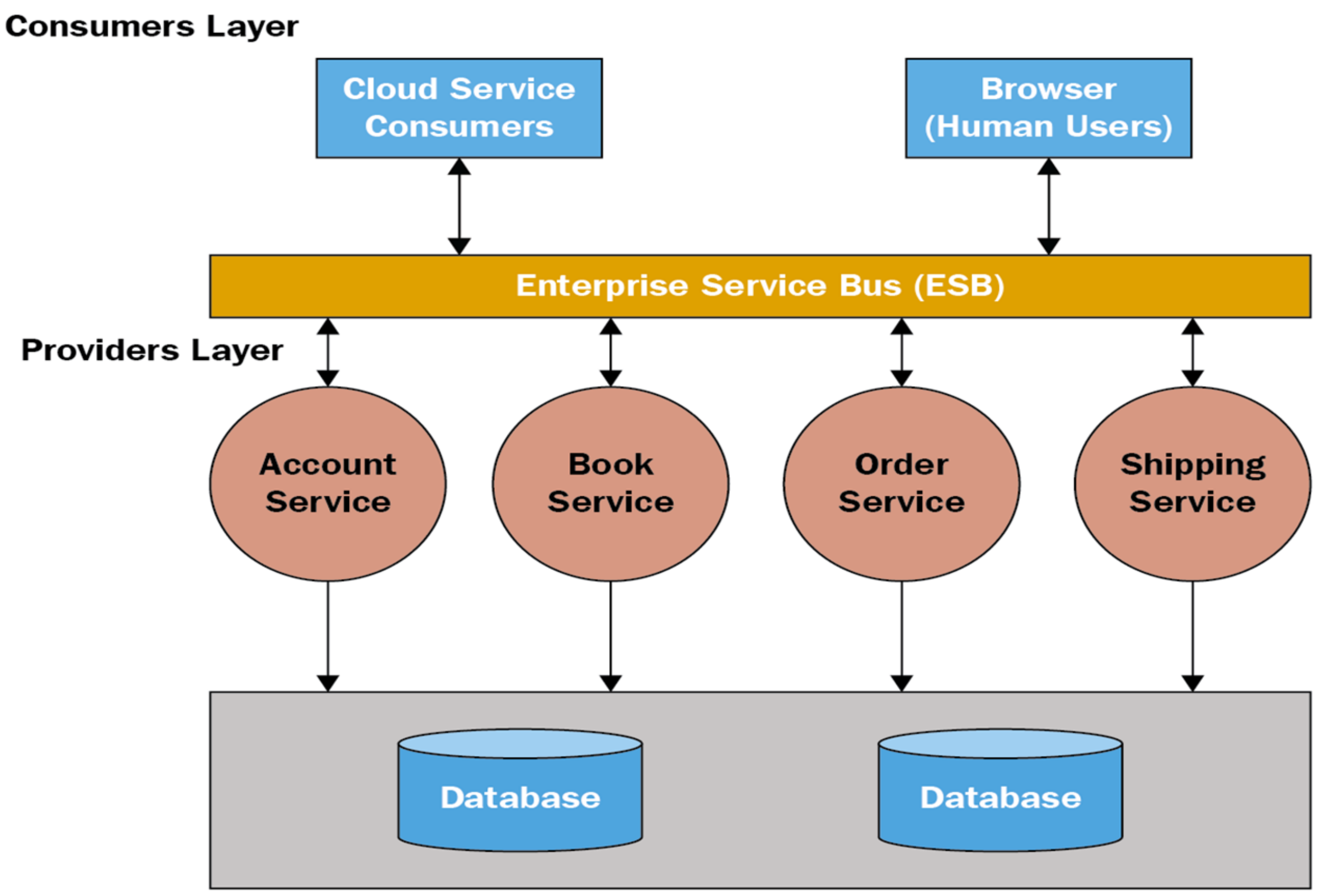
- Service-oriented architecture (SOA) is a method of software development that uses software components called services to create business applications.
- Each service provides a business capability, and services can also communicate with each other across platforms and languages.



What are the benefits SOA?

- Faster time to market
- Efficient maintenance
- Greater adaptability
- Service Reusability

Example of SOA?

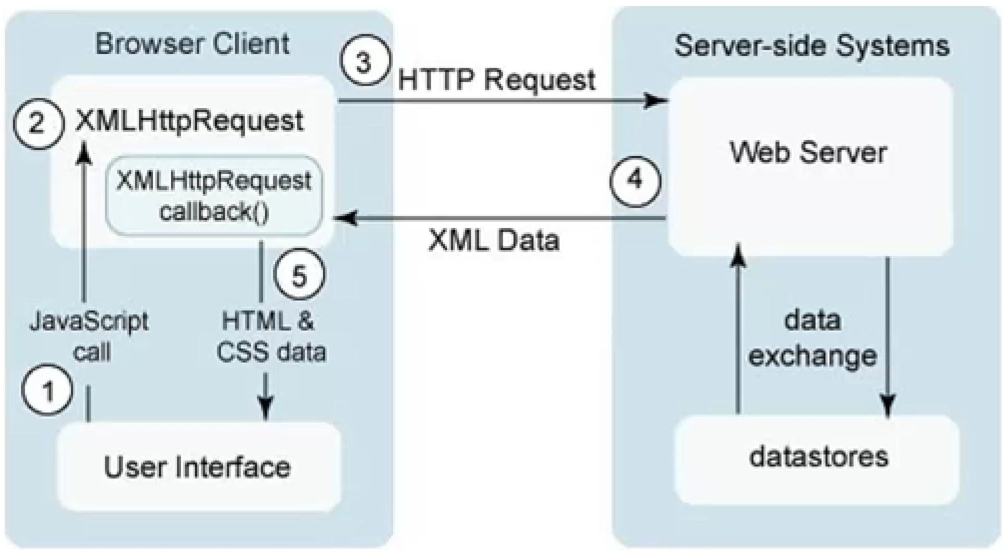


HTTP and XML

- HTTP and XML are two different technologies used in web development, and they serve different purposes.
- HTTP (Hypertext Transfer Protocol) is a protocol that enables communication between web servers and web browsers. It is the foundation of data communication on the World Wide Web. HTTP is responsible for handling requests and responses between the client and the server, and it allows for the transfer of various types of data, including HTML pages, images, audio, and video files.
- XML (Extensible Markup Language), on the other hand, is a markup language that is used to describe data. It provides a way to structure data in a format that is both human-readable and machine-readable. XML is widely used for exchanging data between different systems, and it allows for the creation of custom tags and attributes to describe data in a flexible and extensible manner.

XMLHttpRequest vs HttpRequest

- XMLHttpRequest is a standard javascript object that allows you to make HTTP Requests from the browser in javascript.
- HttpRequest is a server side object that represents a request to the server.



SOAP and WSDL

SOAP	WSDL
SOAP is a protocol used for exchanging structured information between systems over the internet	WSDL is a language used to describe web services and their interfaces
SOAP messages are in XML format	WSDL documents are in XML format
SOAP provides a framework for invoking remote procedures or exchanging data between different systems	WSDL provides a description of the operations provided by a web service
SOAP messages use various protocols, including HTTP, SMTP, and FTP	WSDL can describe the protocols of a web service, such as SOAP, HTTP, and SMTP
SOAP is based on a client-server architecture	WSDL describes the interfaces of web services that are accessed by clients
SOAP messages are dependent on the underlying protocol used for communication	WSDL is independent of any particular protocol
SOAP messages can use various programming languages, including Java, C, and PHP	WSDL is created and parsed using tools such as WSDL editors, XML editors, and web service development kits
SOAP messages are compatible with various web services standards.including WSDL, UDDI, and XML Schema	WSDL can describe any web service that provides a SOAP interface
SOAP can enable communication between different systems	WSDL can describe the interfaces of web services that are accessed by clients
SOAP is more towards the message format and protocol for exchanging data	WSDL is more towards the description of the operations provided by a web service

What is Node.js?

Node.js logo

- Node is an open-source, `asynchronous event-driven` JavaScript `runtime environment` that allows developers to build and run JavaScript applications outside of a web browser. It is designed to be efficient, scalable, and suitable for building a wide range of networked and server-side applications. Node.js is built on the `V8 JavaScript engine` developed by Google, which is the same engine used in the Google Chrome web browser. V8 is written in C++.

- JavaScript was developed to add interactivity to webpages. That is it was meant to run only on browsers. Each browser invented their own Javascript engine which runs Javascript on Browsers. `Google Chrome` has `V8`, `Mozilla` has `Spider Monkey` and `Safari` uses `Javascript Core`. Each engine has their own mechanism for performance. In the year 2009, `Ryan Dahl` came up with a solution to run Javascript out of the browser with the help of V8 Javascript engine.

- Javascript doesn't come with DOM, timers and webapis. These are provided by the browser. We have a global document object in browser. This provides api to manipulate DOM nodes in browser. Browser parses the html and css and exposes apis to Javascript scope to enable DOM manipulation. DOM is not the default of Javascript. `Browser + Event Loop` powers up Javascript.

- Even If we have V8 engine, we lack `Event Loop` which makes Javascript asynchronous. `Timers`, `WebApis` and `Apis` to access the system resources and Filesystem. This is where `Libuv` comes in. It provides all the other tools required to make Node.js complete.

- `JavaScript` is `Single Threaded` and `Blocking`. With the help of `Event Loop` in both browser and Node, it attains the capability of being `Non Blocking`. Main code execution is handled by Javascript engine. So any code that is time consuming will block the other processes. Hence it's recommended to not used Node.js for computationally intensive tasks. Libuv enables us to make use of other threads while main thread is busy executing the code. Node.js exposes `Async methods` which runs on a seprate thread and calls the callback when the said operation is done.

Setting Up Node.js Environment

Setting up a Node.js environment involves installing Node.js and npm (Node Package Manager), setting up a project, and configuring essential tools. Here’s a step-by-step guide to help you get started:

Install Node.js and npm

WINDOWS

- Download Node.js Installer:
    - Go to the Node.js official website.
    - Download the LTS (Long Term Support) version for Windows.
- Install Node.js:
    - Run the downloaded installer.
    - Follow the installation instructions, ensuring that you check the box to add Node.js to your PATH.
- Verify Installation:
    - Open a command prompt and run the following commands to check the installed versions:

```
node -v
npm -v
```

MACOS

- Download Node.js Installer:
    - Go to the Node.js official website.
    - Download the LTS version for macOS.
- Install Node.js:
    - Run the downloaded installer and follow the installation instructions.
- Verify Installation:
    - Open a terminal and run:

```
node -v
npm -v
```

LINUX

- Open a terminal and run the following commands to install nvm:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.1/install.sh |
bash
source ~/.bashrc  # or source ~/.zshrc for zsh users
```

- Install Node.js using nvm:

```
nvm install --lts
nvm use --lts
```

- Verify installation:

```
node -v
npm -v
```

JavaScript Fundamentals

Introduction to JavaScript

What is JavaScript?

- JavaScript is a high-level, interpreted programming language primarily known for its use in web development. It was created by Brendan Eich in 1995 while he was working at Netscape Communications Corporation. Initially named "LiveScript," it was later renamed JavaScript, despite having few similarities with Java.

- JavaScript is widely used for creating interactive and dynamic web pages. It allows developers to add functionality to web pages, manipulate content, respond to user actions, and communicate with servers asynchronously. JavaScript code is typically embedded directly into HTML documents or included from external files.

Key features of JavaScript include:

1. **Client-Side Scripting:** JavaScript runs on the client's web browser, enabling dynamic updates to web pages without requiring a server roundtrip.
2. **Interactivity:** It enables developers to create interactive elements like dropdown menus, forms validation, sliders, and more, enhancing user experience.
3. **Asynchronous Programming:** JavaScript supports asynchronous programming, allowing tasks to be performed concurrently without blocking other operations. This is crucial for handling events, making

AJAX requests, and fetching data from servers without page reloads.
4. **DOM Manipulation:** JavaScript interacts with the Document Object Model (DOM), enabling developers to manipulate HTML elements, change styles, add or remove elements dynamically, and respond to user events.
5. **Versatility:** Beyond web development, JavaScript has expanded its reach to server-side programming (Node.js), mobile app development (React Native, NativeScript), desktop applications (Electron), game development, IoT (Internet of Things), and more.
6. **Rich Ecosystem:** JavaScript has a vast ecosystem of libraries and frameworks like React.js, AngularJS, Vue.js, jQuery, Express.js, and many more, providing developers with tools to streamline development and build complex applications efficiently.

Features of JavaScript

JavaScript boasts a wide array of features that make it a versatile and powerful programming language. Here are some key features:

1. **Interpreted Language:** JavaScript is an interpreted language, meaning that it doesn't require compilation before execution. Browsers directly interpret JavaScript code.

2. **Dynamic Typing:** JavaScript is dynamically typed, meaning you don't need to specify the data type of a variable explicitly. Variables can hold values of any data type and can change types during the execution of a program.

3. **Prototype-based Object-Oriented:** JavaScript is based on prototypes rather than classes. Objects can inherit properties and methods from other objects.

4. **Functions as First-Class Citizens:** In JavaScript, functions are treated as first-class citizens, which means they can be passed as arguments to other functions, returned from functions, assigned to variables, and stored in data structures.

5. **Closures:** JavaScript supports closures, allowing functions to retain access to variables from their containing scope even after the outer function has finished executing.

6. **Asynchronous Programming:** JavaScript is designed for asynchronous programming, which is essential for handling tasks such as fetching data from servers, responding to user actions without blocking the main thread, and executing code concurrently.

7. **Event-Driven Programming:** In web development, JavaScript facilitates event-driven programming, where functions are executed in response to events like user interactions (clicks, keypresses, mouse movements) or changes in the application's state.

8. **DOM Manipulation:** JavaScript interacts with the Document Object Model (DOM), enabling developers to manipulate HTML elements, change their attributes, styles, and content dynamically, which is fundamental for creating interactive web pages.

9. **Cross-platform Compatibility:** JavaScript is supported by all modern web browsers, making it a cross-platform language. It's not limited to web browsers; it can also run on server-side environments like Node.js, enabling full-stack development with a single language.

10. **Rich Ecosystem:** JavaScript has a vast ecosystem of libraries, frameworks, and tools that extend its capabilities and streamline development. Frameworks like React.js, Angular, and Vue.js, along with libraries like jQuery, provide developers with powerful tools for building complex applications efficiently.

These features, along with its simplicity and ubiquity, contribute to JavaScript's popularity and widespread use in various domains, including web development, server-side programming, mobile app development, and more.

History of JavaScript

JavaScript, initially named LiveScript, was created by Brendan Eich in 1995 while he was working at Netscape Communications Corporation. The development of JavaScript was spurred by the need for a scripting language that could be embedded in web browsers to make web pages more dynamic and interactive.

Here's a brief timeline of JavaScript's history:

1. **1995:** JavaScript was introduced in Netscape Navigator 2.0 as LiveScript. It was later renamed JavaScript as part of a marketing agreement between Netscape and Sun Microsystems, leveraging the popularity of Java at the time.

2. **1996:** Microsoft introduced its own version of JavaScript called JScript in Internet Explorer 3.0. While JScript was similar to JavaScript, there were some differences, leading to compatibility issues between browsers.

3. **1997:** The European Computer Manufacturers Association (ECMA) standardized JavaScript, resulting in ECMAScript, the official specification for the language. ECMAScript 1 was based on JavaScript 1.1 and served as the foundation for subsequent versions of the language.

4. **1999:** ECMAScript 3 was released, introducing significant improvements and becoming the widely adopted standard for JavaScript. It remained the dominant version for many years.

5. **2005:** Ajax (Asynchronous JavaScript and XML) gained popularity, revolutionizing web development by enabling asynchronous communication between the browser and the server, leading to more dynamic and interactive web applications.

6. **2009:** ECMAScript 5 was released, introducing new features such as strict mode, JSON support, and additional array methods. This version further solidified JavaScript's position as a powerful and versatile language.

7. **2015:** ECMAScript 6 (also known as ECMAScript 2015 or ES6) was a major update to the language, introducing significant enhancements such as arrow functions, classes, modules, and Promises. ES6 marked a significant milestone in JavaScript's evolution and served as the foundation for future versions.

8. **Subsequent Years:** The ECMAScript specification continued to evolve, with new features and improvements introduced in subsequent versions, including ES7 (2016), ES8 (2017), ES9 (2018), ES10 (2019), and ES11 (2020).

JavaScript has become one of the most widely used programming languages globally, powering the interactive features and functionality of countless websites and web applications. Its versatility, ease of use, and continuous evolution have contributed to its enduring popularity among developers.

Variables and Data Types

- **JavaScript Variable**
- **JavaScript Global Variable**
- **JavaScript Data Types**

JavaScript Variable

In JavaScript, variables are used to store data values. Here are the basics of working with variables:

1.Declaring Variables: Use `let`, `const`, or `va r` to declare variables.

```
let x = 5; // Declaring a variable using let
const pi = 3.14; // Declaring a constant variable
var name = "John"; // Declaring a variable using var (older way, not recommended
in modern JavaScript)
console.log(x); // Output: 5
console.log(pi); // Output: 3.14
console.log(name); // Output: John
```

2.Variable Names: Variable names can contain letters, digits, underscores, and dollar signs. They cannot start with a digit.

Examples: `myVariable`, `name_123`, `$price`.

```
let firstName = "Muralitharan";
let userAge = 30;
let isLoggedIn = true;
let favoriteFruits = ["apple", "banana", "orange"];
let personInfo = { name: "Muralitharan", age: 30 };
const PI = 3.14;
const MAX_VALUE = 100;
console.log("First Name:", firstName);
console.log("User Age:", userAge);
console.log("Is Logged In:", isLoggedIn);
console.log("Favorite Fruits:", favoriteFruits);
console.log("Person Info:", personInfo);
console.log("PI Value:", PI);
console.log("Max Value:", MAX_VALUE);
```

3.Data Types: Variables can hold different types of data such as numbers, strings, booleans, arrays, objects, etc.

```
let message = "Hello, world!"; // String
let age = 30; // Number
let isTrue = true; // Boolean
let fruits = ["apple", "banana", "orange"]; // Array
let person = { name: "John", age: 30 }; // Object
console.log("Message:", message);
console.log("Age:", age);
console.log("Is True:", isTrue);
console.log("Fruits:", fruits);
console.log("Person:", person);
```

4.Variable Scope: Variables can have global scope (accessible throughout the program) or local scope (accessible only within a specific block of code, like inside a function).

```
let globalVariable = "I'm global";
function myFunction() {
   let localVariable = "I'm local";
   console.log(globalVariable); // Accessing global variable
   console.log(localVariable); // Accessing local variable
}
myFunction();
// console.log(localVariable); // This would cause an e rror because localVariable
is not accessible outside the function
```

5.Variable Reassignment: You can change the value of a variable after declaring it.

```
let number = 10;
console.log(number); // Output: 10
number = 20;
console.log(number); // Output: 20
```

Variables are fundamental in JavaScript and are used extensively in writing scripts for web development, server-side development, and more.

JavaScript Global Variable

Global variables in JavaScript are variables declared outside of any function. They are accessible from any part of the code, including inside functions. Here's an example:

```
let globalVariable = "I'm global";

function myFunction() {
    console.log(globalVariable); // Accessing globalVariable inside the function
}

myFunction(); // Output: I'm global
console.log(globalVariable); // Output: I'm global
```

However, using global variables extensively is not recommended because they can lead to issues such as variable name clashes, unintended side effects, and difficulty in debugging. It's often better to use local variables within functions and pass data as arguments or return values when needed.

JavaScript Data Types

JavaScript supports several data types that are used to represent different kinds of values. Here are the basic data types in JavaScript:

1.Primitive Data Types:

The predefined data types provided by JavaScript language are known as primitive data types. Primitive data types are also known as in-built data types.

- **Number**: Represents numeric values.

  Example: `let num = 42;`

- **String**: Represents textual data.

  Example: `let message = "Hello, world!";`

- **Boolean**: Represents true or false values.

  Example: `let isTrue = true;`

- **Undefined**: Represents a variable that has been declared but not assigned a value.

  Example: `let x;`

- **Null**: Represents the intentional absence of any value.

  Example: `let y = null;`

- **Symbol** (ES6+): Represents unique identifiers.

  Example: `const key = Symbol();`

2.Composite Data Types:

A data type is known as a composite data type when it represents a number of similar or different data under a single declaration of variable i.e., a data type that has multiple values grouped together. There are mainly three types of composite data types named as below –

- **Object**: Represents a collection of key -value pairs.

  Example:`let person = { name: "John", age: 30 };`

- **Array**: Represents a collection of elements, indexed by integers.

  Example: `let numbers = [1, 2, 3, 4];`

- **Function**: Functions are a type of object but can be called like a regular function.

  Example: `function add(a, b) { return a + b; }`

```
let message = "Hello, world!"; // String
let age = 30; // Number
let isTrue = true; // Boolean
let fruits = ["apple", "banana", "orange"]; // Array
let person = { name: "John", age: 30 }; // Object

console.log("Message:", message);
console.log("Age:", age);
console.log("Is True:", isTrue);
console.log("Fruits:", fruits);
console.log("Person:", person);
```

JavaScript is dynamically typed, meaning you don't need to explicitly declare the data type of a variable; the type is automatically determined based on the value assigned to it.

Operators and Expressions

JavaScript operators are symbols used to perform operations on operands (variables, values, or expressions). Here are some of the key operators in JavaScript:

1. **Arithmetic Operators**
2. **Assignment Operators**
3. **Comparison Operators**
4. **Logical Operators**
5. **Unary Operators**
6. **Conditional Operator**
7. **Bitwise Operators**

Arithmetic Operators

Arithmetic operators are symbols used to perform mathematical operations on numerical values. Here are the common arithmetic operators:

- **Addition:** `+`

  Addition is one of the fundamental arithmetic operations, where two or more numbers are combined to find their sum. It's represented by the symbol "+". For instance, if you add 2 and 3 together, you get 5: (2 + 3 = 5). It's a simple but essential concept in mathematics and is used in various fields for calculations and problem-solving.

  ```
  let num1 = 5;
  let num2 = 10;

  let sum = num1 + num2;

  console.log("The sum of", num1, "and", num2, "is", sum);
  ```

- **Subtraction:** `-`

  Subtraction is another basic arithmetic operation, denoted by the symbol "-". It involves taking away one number from another to find the difference between them. For example, subtracting 3 from 7 gives you 4: (7 - 3 = 4). Subtraction is commonly used for finding the difference between quantities, determining changes over time, and many other applications in mathematics and everyday life.

  ```
  let num1 = 10;
  let num2 = 5;

  let difference = num1 - num2;

  console.log("The difference between", num1, "and", num2, "is", difference);
  ```

- **Multiplication:** `*`

  Multiplication is an arithmetic operation used to find the result of repeated addition of two numbers. It's represented by the symbol "*", often read as "times" or "multiplied by". For example, multiplying 3 by 4 gives you 12: (3 \times 4 = 12). Multiplication is a fundamental operation in mathematics and has various applications in areas such as algebra, geometry, and physics. It's used for scaling, finding areas and volumes, and many other mathematical operations.

  ```
  let num1 = 10;
  let num2 = 5;

  let difference = num1 * num2;

  console.log("The difference between", num1, "and", num2, "is", difference);
  ```

- **Division:** `/`

  Division is an arithmetic operation used to distribute a quantity into equal parts or to find out how many times one number is contained within another. It's represented by the symbol "÷" or "/", and it's read as "divided by". For example, dividing 10 by 2 gives you 5: (\frac{10}{2} = 5). Division is the inverse operation of multiplication. It's essential for various mathematical tasks, such as finding averages, solving proportions, and determining rates of change.

  ```
  let num1 = 10;
  let num2 = 5;

  let difference = num1 / num2;

  console.log("The difference between", num1, "and", num2, "is", difference);
  ```

- **Modulus (Remainder):** `%`

The modulus operator, represented by the symbol "%", calculates the remainder of a division operation between two numbers. For example, if you divide 10 by 3, the quotient is 3 with a remainder of 1. So, 10 % 3 equals 1. Modulus is commonly used in programming for tasks such as determining if a number is even or odd, cyclic operations, and ensuring values stay within a certain range.

```
let num1 = 10;
let num2 = 5;

let difference = num1 % num2;

console.log("The difference between", num1, "and", num2, "is", difference);
```

- **Increment:** `++`

  Increment refers to increasing a value by a certain amount, typically by one unit. In programming, an increment operation often involves adding 1 to the current value of a variable. For example, if you have a variable `x` with an initial value of 5, incrementing `x` would result in its value becoming 6. This operation is commonly represented by the "++" operator, such as `x++` in many programming languages. Incrementing variables is frequently used in loops, counters, and various algorithms to track progress or manipulate data.

  ```
  let num = 5;

  // Incrementing the value of num by 1
  num++;

  console.log("After incrementing, num is", num);
  ```

- **Decrement:** `--`

  Decrement refers to decreasing a value by a certain amount, usually by one unit. In programming, a decrement operation often involves subtracting 1 from the current value of a variable. For example, if you have a variable `x` with an initial value of 5, decrementing `x` would result in its value becoming 4. This operation is typically represented by the "--" operator, such as `x--` in many programming languages. Decrementing variables is commonly used in loops, countdowns, and various algorithms where values need to be reduced iteratively.

  ```
  let num = 5;

  // Incrementing the value of num by 1
  num--;

  console.log("After incrementing, num is", num);
  ```

Assignment Operators

Assignment operators are used in programming languages to assign values to variables. They combine the assignment operation "=" with another operation, such as addition, subtraction, multiplication, or division. This allows for concise expression of operations where the value of a variable is updated based on its current value and another value.

- **Assignment:** `=`

  The assignment operator, represented by "=", is used to assign a value to a variable. It takes the value on its right side and stores it in the variable on its left side.

  ```
  let num1 = 5;
  let num2;

  num2 = num1;

  console.log("num1:", num1);
  console.log("num2:", num2);
  ```

- **Addition assignment:** `+=`

  The addition assignment operator, represented by +=, is used to add the value of the right operand to the value of the left operand and then assign the result to the left operand. Essentially, it's a shorthand for performing addition and assignment in a single step.

  ```
  let num1 = 5;
  let num2 = 10;

  num1 += num2;

  console.log("After addition assignment, num1 is", num1);
  ```

- **Subtraction assignment:** `-=`

  The subtraction assignment operator, represented by -=, is used to subtract the value of the right operand from the value of the left operand and then assign the result to the left operand. Similar to addition assignment, it provides a shorthand for performing subtraction and assignment in a single step.

  ```
  let num1 = 5;
  let num2 = 10;

  num1 -= num2;

  console.log("After addition assignment, num1 is", num1);
  ```

- **Multiplication assignment:** `*=`

The multiplication assignment operator, represented by *=, is used to multiply the value of the left operand by the value of the right operand and then assign the result to the left operand. It's a shorthand for performing multiplication and assignment in a single step.

```
let num1 = 5;
let num2 = 10;

num1 *= num2;

console.log("After addition assignment, num1 is", num1);
```

- **Division assignment:** `/=`

  The division assignment operator, represented by /=, is used to divide the value of the left operand by the value of the right operand and then assign the result to the left operand. It's a shorthand for performing division and assignment in a single step.

```
let num1 = 5;
let num2 = 10;

num1 /= num2;

console.log("After addition assignment, num1 is", num1);
```

- **Modulus assignment:** `%=`

  The modulus assignment operator, represented by %=, is used to find the remainder when the value of the left operand is divided by the value of the right operand, and then assign the result to the left operand. It's a shorthand for performing modulus operation and assignment in a single step.

```
let num1 = 5;
let num2 = 10;

num1 %= num2;

console.log("After addition assignment, num1 is", num1);
```

Comparison Operators

Comparison operators are used to compare two values and return a Boolean result (True or False) based on the comparison. Here are common comparison operators:

- **Equal to:** `==` **or** `===` **(strict equality)**

  The "equal to" operator, represented by "==", checks whether two values are equal. It's used to compare the values on both sides of the operator and returns True if they are equal, and False otherwise.

```
let num1 = 5;
let num2 = 10;

if (num1 == num2) {
    console.log("num1 is equal to num2");
} else {
    console.log("num1 is not equal to num2");
}
```

- **Not equal to: `!=` or `!==` (strict inequality)**

  The "not equal to" operator, represented by "!=", checks whether two values are not equal. It returns True if the values on both sides of the operator are different and False if they are equal

  ```
  let num1 = 5;
  let num2 = 10;

  if (num1 !== num2) {
      console.log("num1 is equal to num2");
  } else {
      console.log("num1 is not equal to num2");
  }
  ```

- **Greater than: `>`**

  The "greater than" operator, represented by ">", compares two values and returns True if the value on the left side is greater than the value on the right side. If the value on the left side is not greater, it returns False.

  ```
  let num1 = 10;
  let num2 = 5;

  if (num1 > num2) {
      console.log("num1 is greater than num2");
  } else {
      console.log("num1 is not greater to num2");
  }
  ```

- **Less than: `<`**

  The "less than" operator, represented by "<", compares two values and returns True if the value on the left side is less than the value on the right side. If the value on the left side is not less, it returns False.

```
let num1 = 10;
let num2 = 5;

if (num1 < num2) {
    console.log("num1 is greater than num2");
} else {
    console.log("num1 is not greater to num2");
}
```

- **Greater than or equal to:** `>=`

  The "greater than or equal to" operator, represented by ">=", checks if the value on the left side is greater than or equal to the value on the right side. It returns True if the left side value is greater than or equal to the right side value; otherwise, it returns False.

  ```
  let num1 = 5;
  let num2 = 5;

  if (num1 >= num2) {
      console.log("num1 is greater than or equal to num2");
  } else {
      console.log("num1 is less than num2");
  }
  ```

- **Less than or equal to:** `<=`

  The "less than or equal to" operator, represented by "<=", checks if the value on the left side is less than or equal to the value on the right side. It returns True if the left side value is less than or equal to the right side value; otherwise, it returns False.

  ```
  let num1 = 5;
  let num2 = 10;

  if (num1 <>= num2) {
      console.log("num1 is greater than or equal to num2");
  } else {
      console.log("num1 is less than num2");
  }
  ```

Logical Operators

Logical operators are used to perform logical operations on Boolean values or expressions. Here are the common logical operators:

- **Logical AND:** `&&`

The logical AND operator, represented by and, returns True if both of its operands are True. Otherwise, it returns False.

```
let num1 = 5;
let num2 = 10;

if (num1 > 0 && num2 > 0) {
    console.log("Both num1 and num2 are greater than 0");
} else {
    console.log("At least one of num1 or num2 is not greater than 0");
}
```

- **Logical OR:** `||`

  The logical OR operator, represented by or, returns True if at least one of its operands is True. It returns False only if both operands are False.

```
let num1 = 5;
let num2 = 10;

if (num1 > 0 || num2 > 0) {
    console.log("At least one of num1 or num2 is greater than 0");
} else {
    console.log("Neither num1 nor num2 is greater than 0");
}
```

- **Logical NOT:** `!`

  The logical NOT operator, represented by not, negates the value of its operand. It returns True if the operand is False, and False if the operand is True.

```
let isLoggedIn = true;

if (!isLoggedIn) {
    console.log("User is not logged in");
} else {
    console.log("User is logged in");
}
```

Unary Operators

Unary operators are operators that work with only one operand. They perform various operations such as negation, increment, and decrement on a single operand. Here are some common unary operators:

- **Unary plus:** `+`

The unary plus operator, represented by "+", doesn't change the sign of the operand. It's mainly included for symmetry with the unary minus operator ("*") and is rarely used since it doesn't have any effect on positive values.

```
let strNum = "123";
let num = +strNum;

console.log("String Number:", strNum);
console.log("Converted Number:", num);
```

- **Unary minus:** `-`

  The unary minus operator, represented by "-", negates the value of its operand. It changes the sign of the operand to its opposite.

```
let num = 10;
let negatedNum = -num;

console.log("Original Number:", num);
console.log("Negated Number:", negatedNum);
```

- **Typeof:** `typeof`

  In Python, the equivalent of the typeof operator in JavaScript is type(). The type() function returns the type of the specified object. Here's how it's used:

```
let num = 5;
let str = "Hello";
let bool = true;
let arr = [1, 2, 3];
let obj = { key: "value" };
let func = function() {};

console.log(typeof num);  // Output: "number"
console.log(typeof str);  // Output: "string"
console.log(typeof bool); // Output: "boolean"
console.log(typeof arr);  // Output: "object" (arrays are objects in
JavaScript)
console.log(typeof obj);  // Output: "object"
console.log(typeof func); // Output: "function"
```

- **Delete:** `delete`

  In JavaScript, the delete operator is used to delete properties from objects. It's important to note that it doesn't directly delete variables or identifiers.

```
let obj = { a: 1, b: 2, c: 3 };

console.log("Before delete:", obj); // Output: { a: 1, b: 2, c: 3 }

delete obj.b;

console.log("After delete:", obj); // Output: { a: 1, c: 3 }
```

Conditional Operator

In programming, the conditional operator, also known as the ternary operator, provides a concise way to write conditional statements. It's often used as a shortcut for the if...else statement when the conditions are simple.

- `condition ? expression1 : expression2`

  ```
  let num = 10;
  let result = num > 0 ? "Positive" : "Non-positive";

  console.log("Number is:", result);
  ```

Bitwise Operators

Bitwise operators are operators that manipulate individual bits of binary numbers. They are commonly used in low-level programming for tasks such as data compression, encryption, and optimization. Here are some common bitwise operators:

- **Bitwise AND:** `&`

  The bitwise AND operator (&) in JavaScript performs a bitwise AND operation on each pair of corresponding bits of the operands. Here's a brief explanation and an example:

  - Syntax: operand1 & operand2
  - Explanation: The result of a bitwise AND operation is 1 if both bits at the same position are 1; otherwise, it's 0.

  ```
  let num1 = 5;  // Binary: 101
  let num2 = 3;  // Binary: 011

  let result = num1 & num2;  // Binary AND: 101 & 011 = 001 (Decimal: 1)

  console.log("Result:", result);  // Output: 1
  ```

- **Bitwise OR:** `|`

  The bitwise OR operator (|) in JavaScript performs a bitwise OR operation on each pair of corresponding bits of the operands. Here's an explanation and an example:

  - Syntax: operand1 | operand2
  - Explanation: The result of a bitwise OR operation is 1 if at least one of the bits at the same position is 1; otherwise, it's 0.

  ```
  let num1 = 5;  // Binary: 101
  let num2 = 3;  // Binary: 011

  let result = num1 | num2;  // Binary AND: 101 & 011 = 001 (Decimal: 1)

  console.log("Result:", result);  // Output: 1
  ```

- **Bitwise XOR:** `^`

  The bitwise XOR operator (^) in JavaScript performs a bitwise XOR (exclusive OR) operation on each pair of corresponding bits of the operands. Here's an explanation and an example:

  - Syntax: operand1 ^ operand2
  - Explanation: The result of a bitwise XOR operation is 1 if the bits at the same position are different (one is 0 and the other is 1); otherwise, it's 0.

  ```
  let num1 = 5; // Binary: 0101
  let num2 = 3; // Binary: 0011

  let result = num1 ^ num2; // Binary result: 0110 (Decimal: 6)

  console.log("Result of bitwise XOR:", result);
  ```

- **Bitwise NOT:** `~`

  The bitwise NOT operator (~) in JavaScript performs a bitwise NOT operation on its operand, which is a unary operator (it operates on a single operand). Here's an explanation and an example:

  - Syntax: ~operand
  - Explanation: The result of a bitwise NOT operation is the complement of its operand. It flips each bit of the operand, changing 1s to 0s and 0s to 1s.

  ```
  let num = 5;
  let bitwiseNotNum = ~num;

  console.log("Original Number:", num);
  console.log("Bitwise NOT Result:", bitwiseNotNum);
  ```

- **Left shift:** `<<`

  The left shift operator (<<) in JavaScript shifts the bits of a number to the left by a specified number of positions. Here's how it works and an example:

    - Syntax: operand << count
    - Explanation: Shifting to the left means moving all bits towards the left by count positions. This effectively multiplies the number by 2 raised to the power of count.

  ```
  let num = 5;
  let shiftedNum = num << 2;

  console.log("Original Number:", num);
  console.log("Shifted Number:", shiftedNum);
  ```

- **Right shift:** `>>`

  The right shift operator (>>) in JavaScript shifts the bits of a number to the right by a specified number of positions. Here's how it works and an example:

    - Syntax: operand >> count
    - Explanation: Shifting to the right means moving all bits towards the right by count positions. For positive numbers, this effectively divides the number by 2 raised to the power of count, and for negative numbers, it depends on the implementation (sign*extension or zero*fill).

  ```
  let num = 5;
  let shiftedNum = num >> 2;

  console.log("Original Number:", num);
  console.log("Shifted Number:", shiftedNum);
  ```

- **Unsigned right shift:** `>>>`

  The unsigned right shift operator (>>>) in JavaScript shifts the bits of a number to the right by a specified number of positions, filling the leftmost positions with zeros. Here's how it works and an example:

    - Syntax: operand >>> count
    - Explanation: Shifting to the right means moving all bits towards the right by count positions. Unlike the right shift operator (>>), the unsigned right shift operator fills the leftmost positions with zeros, regardless of the sign of the number.

  ```
  let num = -8; // Binary representation: 11111111111111111111111111111000
  (32-bit signed integer)
  let shiftedNum = (num >>> 2) & 0x3fffffff; // Simulate unsigned right shift

  console.log("Original Number:", num);
  console.log("Unsigned Right Shift Result:", shiftedNum);
  ```

Understanding and using these operators effectively is key to writing JavaScript code that performs calculations, makes decisions, and manipulates data efficiently.

Control Flow Statements

- **JavaScript If Statement**
- **JavaScript Switch**
- **JavaScript Loop**

JavaScript If Statement

In JavaScript, the `if` statement is used to execute a block of code if a specified condition is true. Here's the basic syntax of an `if` statement:

```
if (condition) {
    // Code block to execute if the condition is true
}
```

For example, let's say we want to check if a number is greater than 10 and then log a message:

```
let number = 15;

if (number > 10) {
    console.log("The number is greater than 10.");
}
```

You can also include an `else` statement to specify a block of code to execute if the condition is false:

```
let number = 5;

if (number > 10) {
    console.log("The number is greater than 10.");
} else {
    console.log("The number is not greater than 10.");
}
```

Additionally, you can use `else if` to specify multiple conditions:

```
let number = 7;

if (number > 10) {
    console.log("The number is greater than 10.");
} else if (number === 10) {
    console.log("The number is equal to 10.");
} else {
    console.log("The number is less than 10.");
}
```

This allows you to handle different scenarios based on various conditions in your code.

JavaScript Switch

The `switch` statement in JavaScript is used to perform different actions based on different conditions. It's an alternative to using multiple `if` statements. Here's the basic syntax of a `switch` statement:

```
switch (expression) {
  case value1:
    // Code block to execute if expression matches value1
    break;
  case value2:
    // Code block to execute if expression matches value2
    break;
  // More cases as needed
  default:
    // Code block to execute if expression doesn't match any case
}
```

Here's an example to illustrate how `switch` works:

```
let day = "Monday";
let message;

switch (day) {
  case "Monday":
    message = "Today is Monday.";
    break;
  case "Tuesday":
    message = "Today is Tuesday.";
    break;
  case "Wednesday":
    message = "Today is Wednesday.";
    break;
  // More cases for other days
  default:
    message = "Unknown day.";
}

console.log(message); // Output: "Today is Monday."
```

In this example, the `switch` statement evaluates the value of the `day` variable and executes the corresponding code block based on the matching case. If none of the cases match, the `default` block is executed. The `break` statement is important to exit the `switch` statement after executing the correct case.

JavaScript Loop

JavaScript provides several types of loops to iterate over data or execute code repeatedly. Here are the main types of loops:

1. For Loop:

Used when you know the number of iterations.

```
for (let i = 0; i < 5; i++) {
    console.log(i);
}
```

2. While Loop:

Used when you don't know the number of iterations before hand

```
let i = 0;
   while (i < 5) {
    console.log(i);
    i++;
}
```

3. Do...While Loop:

Similar to a `while` loop but always executes the code block at least once before checking the condition

```
 let i = 0;
 do {
     console.log(i);
     i++;
     }while (i < 5);
```

4. For...In Loop:

Used to iterate over the properties of an object.

```
const person = {
    name: "John",
    age: 30,
    city: "New York"
};
for (let key in person) {
    console.log(key + ": " + person[key]);
}
```

5. For...Of Loop:

Introduced in ES6, used to iterate over iterable objects like arrays or strings.

```
const fruits = ["apple", "banana", "cherry"];
for (let fruit of fruits) {
    console.log(fruit);
}
```

Each type of loop has its use cases depending on the specific scenario you're dealing with. Experimenting with these loops will give you a good understanding of how to use them effectively in your code.

Functions

- **Functions Definition**
- **Function Parameters**
- **JavaScript Function Invocation**
- **JavaScript Closures**

Functions Definition

- JavaScript functions are defined with the `function` keyword.
- You can use a function declaration or a function expression.

functions are declared with the following syntax:

```
function functionName(parameters) {
  // code to be executed
}
```

Example:

```
function myFunction(a, b) {
  return a * b;
}
```

```
// Functions Definition
function myFunction(a, b) {
    const result = a * b;
    return result;
}

// Example usage of the function
const output = myFunction(5, 3);
console.log("Output of myFunction:", output);
```

Function Expressions

- A JavaScript function can also be defined using an expression.
- A function expression can be stored in a variable:

```
const x = function (a, b) {return a * b};
```

After a function expression has been stored in a variable, the variable can be used as a function:

```
const x = function (a, b) {return a * b};
let z = x(4, 3);
console.log("Output of Function Expressions:", z);
```

- The function above is actually an anonymous function (a function without a name).
- Functions stored in variables do not need function names. They are always invoked (called) using the variable name.

The Function() Constructor

- Functions can also be defined with a built-in JavaScript function constructor called Function().

Example

```
const myFunction = new Function("a", "b", "return a * b");
let x = myFunction(4, 3);
console.log("Output of Function() Constructor:", x);
```

Function Hoisting

- Hoisting is JavaScript's default behavior of moving declarations to the top of the current scope.
- Hoisting applies to variable declarations and to function declarations.

Because of this, JavaScript functions can be called before they are declared:

```
let x = myFunction(5);
console.log("Output of Function Hoisting:", x);

function myFunction(y) {
  return y * y;
}
```

- Functions defined using an expression are not hoisted.

Self-Invoking Functions

- Function expressions can be made "self-invoking".
- A self-invoking expression is invoked (started) automatically, without being called.
- Function expressions will execute automatically if the expression is followed by ().
- You cannot self-invoke a function declaration.

You have to add parentheses around the function to indicate that it is a function expression:

```
(function () {
    let x = "Hello!!";  // I will invoke myself
    console.log("Output of Self-Invoking Functions:", x);
  })();
```

Functions Can Be Used as Values

JavaScript functions can be used as values:

Example

```
function myFunction(a, b) {
  return a * b;
}

let x = myFunction(4, 3);
```

JavaScript functions can be used in expressions:

```
function myFunction(a, b) {
    return a * b;
  }

let x = myFunction(4, 3) * 2;

console.log("Output of Functions Can Be Used as Values:", x)
```

Functions are Objects

- The `typeof` operator in JavaScript returns "function" for functions.
- But, JavaScript functions can best be described as objects.
- JavaScript functions have both properties and methods.

The arguments.length property returns the number of arguments received when the function was invoked:

```
function myFunction(a, b) {
  return arguments.length;
}
```

The `toString()` method returns the function as a string:

```
function myFunction(a, b) {
  return a * b;
}

let text = myFunction.toString();
// returns the function as a string
console.log(text)
```

Arrow Functions

- Arrow functions allows a short syntax for writing function expressions.
- You don't need the `function` keyword, the `return` keyword, and the `curly brackets`.

```
// ES5
var x = function(x, y) {
  return x * y;
}

// ES6
const x = (x, y) => x * y;
```

- Arrow functions do not have their own `this`. They are not well suited for defining object methods.
- Arrow functions are not hoisted. They must be defined before they are used.
- Using `const` is safer than using `var`, because a function expression is always constant value.
- You can only omit the `return` keyword and the curly brackets if the function is a single statement. Because of this, it might be a good habit to always keep them:

```
// Arrow Functions

const x = (x, y) => { return x * y };
const output = x(4,5);
console.log("Output of Arrow Functions:",output)
```

Function Parameters

- JavaScript functions do not perform any checking on parameter values (arguments).

1. Parameters vs. Arguments:

- Parameters are the names listed in the function definition.
- Arguments are the real values passed to (and received by) the function.

2. Parameter Rules:

- No Data Type Specification: JavaScript function definitions do not specify data types for parameters.
- No Type Checking: JavaScript functions do not perform type checking on the passed arguments.
- No Argument Count Checking: JavaScript functions do not check the number of arguments received.

3. Default Parameters:

- If a function is called with missing arguments (less than declared), the missing values are set to `undefined`.
- Default values can be assigned to parameters to handle missing arguments.

```
function myFunction(x, y) {
  if (y === undefined) {
    y = 2;
  }
}
```

```
function myFunction(x, y) {
    if (y === undefined) {
      y = 2;
    }
    return x * y;
  }

let output = myFunction(4);

console.log("output of default parameters:", output);
```

4. Default Parameter Values in ES6:

- ES6 allows function parameters to have default values.

```
function myFunction(x, y = 10) {
    return x + y;
  }
let output = myFunction(5); // Returns 15

console.log("output of default parameters in ES6:", output);
```

5. Function Rest Parameter:

- The rest parameter (`...args`) allows a function to treat an indefinite number of arguments as an array.

```
function sum(...args) {
    let sum = 0;
    for (let arg of args) sum += arg;
    return sum;
  }
let x = sum(4, 9, 16, 25, 29, 100, 66, 77);

console.log("Sum =", x);
```

6. The Arguments Object:

- JavaScript functions have a built-in `arguments` object containing an array of the arguments used when the function was called.

```
function findMax() {
  let max = -Infinity;
  for (let i = 0; i < arguments.length; i++) {
    if (arguments[i] > max) {
      max = arguments[i];
    }
  }
  return max;
}
let x = findMax(1, 123, 500, 115, 44, 88);

console.log("Max =",x);
```

- Example of a function using the `arguments` object to sum all input values.

```
// Summing All Input Values:
function sumAll() {
  let sum = 0;
  for (let i = 0; i < arguments.length; i++) {
    sum += arguments[i];
  }
  return sum;
}
let x = sumAll(1, 123, 500, 115, 44, 88);

console.log("SumAll =",x);
```

7. Excess Arguments:

- If a function is called with more arguments than declared, these extra arguments can be accessed using the `arguments` object.

8. Arguments are Passed by Value:

- Function arguments in JavaScript are passed by value. The function gets only the values, not the locations of the arguments.
- If a function changes an argument's value, it does not affect the original value outside the function.

9. Objects are Passed by Reference:

- Object references in JavaScript are values.
- Functions can change object properties, which will affect the original object.
- Changes to object properties are visible outside the function.

JavaScript Function Invocation

The code inside a JavaScript function will execute when "something" invokes it.

Invoking a JavaScript Function

- **Execution Upon Invocation:**
  - The code inside a function is not executed when the function is defined.
  - The code inside a function is executed when the function is invoked.
- **Terminology:**
  - Common terms: "call a function," "invoke a function," "call upon a function," "start a function," "execute a function."
  - "Invoke" is preferred because a JavaScript function can be invoked without being explicitly called.

Invoking a Function as a Function

Example:

```
function myFunction(a, b) {
  return a * b;
}
myFunction(10, 2); // Will return 20
```

- The function above does not belong to any object.
- In a browser, the function becomes a property of the global object (the `window` object in browsers).

Example:

```
function myFunction(a, b) {
  return a * b;
}
window.myFunction(10, 2); // Will also return 20
```

The `this` Keyword

- In JavaScript, `this` refers to an object depending on the context:

- In an object method, `this` refers to the object.
- Alone, `this` refers to the global object.
- In a function, `this` refers to the global object.
- In a function (in strict mode), `this` is `undefined`.
- In an event, `this` refers to the element that received the event.
- Methods like `call()`, `apply()`, and `bind()` can set `this` to any object.

Example:

```
let x = myFunction(); // x will be the window object

function myFunction() {
  return this;
}
```

Invoking a Function as a Method

- Functions can be defined as object methods.

Example:

```
const myObject = {
    firstName: "John",
    lastName: "Doe",
    fullName: function () {
      return this.firstName + " " + this.lastName;
    }
  }
let x = myObject.fullName(); // Will return "John Doe"
console.log("Output for this Keyword:",x);
```

- The `fullName` method belongs to the `myObject` object, making `myObject` the owner of the function.
- `this` refers to the owning object.

Example:

```
const myObject = {
  firstName: "John",
  lastName: "Doe",
  fullName: function () {
    return this;
  }
}
// This will return [object Object] (the owner object)
let x = myObject.fullName();
console.log("Output for this Keyword:",x);
```

Invoking a Function with a Function Constructor

- Using the `new` keyword invokes a function as a constructor, creating a new object.

Example:

```
// Function constructor:
function MyFunction(arg1, arg2) {
  this.firstName = arg1;
  this.lastName = arg2;
}

// Creating a new object
const myObj = new MyFunction("John", "Doe");

// This will return "John"
myObj.firstName;
```

- The constructor invocation creates a new object, inheriting properties and methods from its constructor.
- The `this` keyword in the constructor refers to the new object created.

JavaScript Closures

JavaScript variables can belong to the local or global scope. Global variables can be made local (private) with closures.

Global Variables

A function can access all variables defined inside the function:

```
function myFunction() {
  let a = 4;
  return a * a;
}
```

A function can also access variables defined outside the function:

```
let a = 4;
function myFunction() {
  return a * a;
}
```

In this example, `a` is a global variable. Global variables can be used (and changed) by all other scripts on the page, whereas local variables can only be used inside the function where they are defined.

Variable Lifetime

- Global variables live until the page is discarded, like when you navigate to another page or close the window. Local variables are created when the function is invoked and deleted when the function finishes.

A Counter Dilemma

- Suppose you want to use a variable for counting something, and you want this counter to be available to all functions. Using a global variable:

```
let counter = 0;

function add() {
  counter += 1;
}

add();
add();
add();

console.log(counter); // The counter should now be 3
```

This approach allows any code on the page to change the counter without calling `add()`. To prevent this, the counter should be local to the `add()` function:

```
function add() {
  let counter = 0;
  counter += 1;
}

add();
add();
add();

console.log(counter); // The counter should now be 3. But it is 0
```

This did not work because we displayed the global counter instead of the local counter. Removing the global counter and accessing the local counter by returning it:

```
function add() {
  let counter = 0;
  counter += 1;
  return counter;
}

console.log(add()); // Call add() 3 times
console.log(add());
console.log(add());
```

The counter should now be 3, but it is 1 because we reset the local counter every time we call the function. A JavaScript inner function can solve this.

JavaScript Nested Functions

All functions have access to the global scope. Nested functions have access to the scope "above" them.

```
function add() {
  let counter = 0;
  function plus() {
    counter += 1;
  }
  plus();
  return counter;
}
```

This could have solved the counter dilemma if we could reach the `plus()` function from the outside. We also need to find a way to execute `counter = 0` only once. We need a closure.

JavaScript Closures

Remember self-invoking functions? What does this function do?

```
const add = (function () {
  let counter = 0;
  return function () {
    counter += 1;
    return counter;
  }
})();

console.log(add()); // Call add() 3 times
console.log(add());
console.log(add());
```

Example Explained

- The variable `add` is assigned to the return value of a self-invoking function. The self-invoking function only runs once. It sets the counter to zero (0) and returns a function expression. This way, `add` becomes a function. The "wonderful" part is that it can access the counter in the parent scope.
- This is called a JavaScript closure. It makes it possible for a function to have "private" variables. The counter is protected by the scope of the anonymous function and can only be changed using the `add` function.

Objects

- **Defining JavaScript Objects**

Defining JavaScript Objects

Using an Object Literal

An object literal is a list of property names and values inside curly braces `{}`.

```
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  eyeColor: "blue"
};
```

Using the new Keyword

You can create an empty JavaScript object using `new Object()` and then add properties.

```
const person = new Object();
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";
```

Object Constructor Functions

To create multiple objects of the same type, you can use an object constructor function.

```
function Person(first, last, age, eye) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eye;
}

const myFather = new Person("John", "Doe", 50, "blue");
const myMother = new Person("Sally", "Rally", 48, "green");
const mySister = new Person("Anna", "Rally", 18, "green");
const mySelf = new Person("Johnny", "Rally", 22, "green");
```

Property Default Values

You can set default values for properties in the constructor function.

```
function Person(first, last, age, eyecolor) {
  this.firstName = first;
  this.lastName = last;
```

```
    this.age = age;
    this.eyeColor = eyecolor;
    this.nationality = "English";
  }
```

JavaScript Object Methods

General Methods

- `Object.assign(target, source)`
- `Object.create(object)`
- `Object.entries(object)`
- `Object.fromEntries()`
- `Object.keys(object)`
- `Object.values(object)`
- `Object.groupBy(object, callback)`

Property Management Methods

- `Object.defineProperty(object, property, descriptor)`
- `Object.defineProperties(object, descriptors)`
- `Object.getOwnPropertyDescriptor(object, property)`
- `Object.getOwnPropertyDescriptors(object)`
- `Object.getOwnPropertyNames(object)`
- `Object.getPrototypeOf(object)`

Object Protection Methods

- `Object.preventExtensions(object)`
- `Object.isExtensible(object)`
- `Object.seal(object)`
- `Object.isSealed(object)`
- `Object.freeze(object)`
- `Object.isFrozen(object)`

Classes

- **Class Intro**
- **JavaScript Class Inheritance**
- **JavaScript Static Methods**

Class Intro

JavaScript classes, introduced in ECMAScript 2015 (ES6), provide a more structured and object-oriented way to create and manage objects. Here's a detailed explanation of how to work with JavaScript classes:

Creating a Class

- A class is defined using the `class` keyword followed by the class name. The constructor method initializes the object properties.

Syntax:

```
class ClassName {
  constructor(parameter1, parameter2) {
    this.property1 = parameter1;
    this.property2 = parameter2;
  }
}
```

Example:

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
}
```

Creating Objects from a Class

- Once a class is defined, you can create objects using the `new` keyword.

Example:

```
const myCar1 = new Car("Ford", 2014);
const myCar2 = new Car("Audi", 2019);

console.log(myCar1); // Output: Car { name: 'Ford', year: 2014 }
console.log(myCar2); // Output: Car { name: 'Audi', year: 2019 }
```

The Constructor Method

- The `constructor` method is a special method for initializing objects. It is automatically called when creating a new object instance.

Example:

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
}
```

Adding Methods to a Class

- Methods can be added to a class to perform actions on objects created from the class.

Syntax:

```
class ClassName {
  constructor(parameter1, parameter2) {
    this.property1 = parameter1;
    this.property2 = parameter2;
  }

  method1() {
    // code for method1
  }

  method2() {
    // code for method2
  }
}
```

Example:

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }

  age() {
    const date = new Date();
    return date.getFullYear() - this.year;
  }
}

const myCar = new Car("Ford", 2014);
console.log("My car is " + myCar.age() + " years old."); // Output: My car is 10
years old. (assuming the current year is 2024)
```

Sending Parameters to Class Methods

- You can pass parameters to class methods to make them more dynamic.

Example:

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
  age(currentYear) {
    return currentYear - this.year;
  }
}

const date = new Date();
let year = date.getFullYear();

const myCar = new Car("Ford", 2014);
console.log("My car is " + myCar.age(year) + " years old."); // Output: My car is 10 years old. (assuming the current year is 2024)
```

Full Example with Console Output

- Here's a complete example demonstrating the creation of a class, instantiating objects, and using methods:

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }

  age() {
    const date = new Date();
    return date.getFullYear() - this.year;
  }

  ageWithYear(currentYear) {
    return currentYear - this.year;
  }
}

// Create instances
const myCar1 = new Car("Ford", 2014);
const myCar2 = new Car("Audi", 2019);

// Get the current year
const date = new Date();
const currentYear = date.getFullYear();

// Use methods
console.log(`My car ${myCar1.name} is ${myCar1.age()} years old.`); // Using age() method
console.log(`My car ${myCar2.name} is ${myCar2.ageWithYear(currentYear)} years old.`); // Using ageWithYear() method with parameter
```

JavaScript Class Inheritance

Class inheritance in JavaScript allows you to create a new class based on an existing class. The new class (child class) inherits all the properties and methods from the existing class (parent class). This feature promotes code reuse and simplifies the process of extending existing functionalities.

Basic Syntax

- To create a class that inherits from another class, use the `extends` keyword.

Example:

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }

  present() {
    return 'I have a ' + this.carname;
  }
}

class Model extends Car {
  constructor(brand, mod) {
    super(brand); // Calls the parent class constructor with the brand parameter
    this.model = mod;
  }

  show() {
    return this.present() + ', it is a ' + this.model;
  }
}

let myCar = new Model("Ford", "Mustang");
console.log(myCar.show()); // Output: I have a Ford, it is a Mustang
```

In this example:

- `Model` class extends the `Car` class.
- The `super` keyword is used to call the parent class's constructor and initialize the inherited properties.

Getters and Setters

- Getters and setters allow you to define methods to access and update the properties of an object. They provide a way to control access to the properties of an object, which is especially useful when you need to perform some actions on the data being set or retrieved.

Example:

```
class Car {
  constructor(brand) {
    this._carname = brand;
  }

  get carname() {
    return this._carname;
  }

  set carname(name) {
    this._carname = name;
  }
}

const myCar = new Car("Ford");
console.log(myCar.carname); // Output: Ford

myCar.carname = "Volvo";
console.log(myCar.carname); // Output: Volvo
```

Using Getters and Setters

- **Getter:** The `get` keyword defines a getter method to access the property.
- **Setter:** The `set` keyword defines a setter method to set the property value.

Example with underscore convention:

```
class Car {
  constructor(brand) {
    this._carname = brand;
  }

  get carname() {
    return this._carname;
  }

  set carname(name) {
    this._carname = name;
  }
}

const myCar = new Car("Ford");
console.log(myCar.carname); // Output: Ford

myCar.carname = "Volvo";
console.log(myCar.carname); // Output: Volvo
```

Class Declarations and Hoisting

- Unlike functions and other JavaScript declarations, class declarations are not hoisted. This means you must declare a class before you can use it.

Example:

```
// This will raise an error
// const myCar = new Car("Ford");

class Car {
  constructor(brand) {
    this.carname = brand;
  }
}

// Now you can use the class
const myCar = new Car("Ford");
console.log(myCar.carname); // Output: Ford
```

Complete Example

- Combining all concepts: inheritance, getters, setters, and hoisting.

```
class Car {
  constructor(brand) {
    this._carname = brand;
  }

  get carname() {
    return this._carname;
  }

  set carname(name) {
    this._carname = name;
  }

  present() {
    return 'I have a ' + this.carname;
  }
}

class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }

  show() {
    return this.present() + ', it is a ' + this.model;
  }
}

const myCar = new Model("Ford", "Mustang");
console.log(myCar.show()); // Output: I have a Ford, it is a Mustang

myCar.carname = "Volvo";
console.log(myCar.show()); // Output: I have a Volvo, it is a Mustang
```

In this comprehensive example:

- The Model class inherits from the Car class.
- Getters and setters are used to manage the _carname property.
- The super method is used to call the parent class's constructor.

JavaScript Static Methods

Static methods in JavaScript are methods that belong to the class itself rather than to instances of the class. These methods can be called directly on the class without creating an instance of the class. Static methods are commonly used to create utility functions that pertain to the class as a whole.

Defining and Using Static Methods

- To define a static method, use the static keyword within the class definition.

Example:

```
class Car {
  constructor(name) {
    this.name = name;
  }

  static hello() {
    return "Hello!!";
  }
}

const myCar = new Car("Ford");

// Calling the static method on the class
console.log(Car.hello()); // Output: Hello!!

// Attempting to call the static method on an instance will raise an error
// console.log(myCar.hello()); // Error: myCar.hello is not a function
```

Using Static Methods with Object Parameters

- If you need to use the properties of an instance inside a static method, you can pass the instance as a parameter to the static method.

Example:

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello(carInstance) {
    return `Hello ${carInstance.name}`;
  }
}

const myCar = new Car("Ford");
console.log(Car.hello(myCar)); // Output: Hello Ford
```

Javascript Async

- **JavaScript Callbacks**
- **Asynchronous JavaScript**
- **JavaScript Promises**
- **JavaScript Async Await**

JavaScript Callbacks

A callback is a function that is passed as an argument to another function. This technique allows the second function to call the callback function at a specific point in time. Callbacks are essential for managing asynchronous operations, such as handling data from APIs or performing operations after a delay.

Function Sequence

- JavaScript functions execute in the order they are called, not in the order they are defined. Here are two examples demonstrating this:

Example 1: Executing Functions in Sequence

```
function myFirst() {
  console.log("Hello");
}

function mySecond() {
  console.log("Goodbye");
}

myFirst();  // Outputs: Hello
mySecond(); // Outputs: Goodbye
```

Example 2: Changing the Order of Function Calls

```
function myFirst() {
  console.log("Hello");
}

function mySecond() {
  console.log("Goodbye");
}
```

```
mySecond(); // Outputs: Goodbye
myFirst();  // Outputs: Hello
```

Sequence Control

- Sometimes, you want to have better control over the sequence in which functions execute. For example, you might want to perform a calculation and then display the result.

Without Callback: Separate Function Calls

```
function myDisplayer(some) {
  console.log(some);
}

function myCalculator(num1, num2) {
  let sum = num1 + num2;
  return sum;
}

let result = myCalculator(5, 5);
myDisplayer(result);
```

Without Callback: Direct Display in Calculator

```
function myDisplayer(some) {
  console.log(some);
}

function myCalculator(num1, num2) {
  let sum = num1 + num2;
  myDisplayer(sum);
}

myCalculator(5, 5);
```

Using Callbacks

- To solve the problems with the above approaches, you can use a callback function. This allows you to have more control over when the display function is called.

With Callback:

```
// Define the callback function that will display the result
function myDisplayer(some) {
    console.log(some);
  }
  // Define the calculator function that will perform the calculation and call the
callback
  function myCalculator(num1, num2, myCallback) {
    let sum = num1 + num2;
    myCallback(sum);
  }

  // Call the calculator function with the callback
  myCalculator(5, 5, myDisplayer); // Outputs: 10
```

In the example above, myDisplayer is passed as a callback to myCalculator. The myCalculator function calls myDisplayer after computing the sum.

When to Use Callbacks?

Callbacks are especially useful in asynchronous functions where you need to wait for an operation to complete before proceeding, such as:

- Fetching data from an API
- Reading or writing files
- Setting timers with setTimeout or setInterval
- Handling events in event-driven programming

Asynchronous JavaScript

Introduction to Asynchronous Functions

- Asynchronous functions run in parallel with other functions. A classic example is JavaScript's setTimeout(), which executes a function after a specified delay, allowing the main program to continue running in the meantime.

Simple Callback Example

- Callbacks are functions passed as arguments to other functions and executed after some operation has been completed.

```
function myDisplayer(something) {
  console.log(something);
}

function myCalculator(num1, num2, myCallback) {
  let sum = num1 + num2;
  myCallback(sum);
}

myCalculator(5, 5, myDisplayer);
```

Using setTimeout() for Asynchronous Execution

- The setTimeout() function can be used to delay the execution of a function.

```
setTimeout(myFunction, 3000);

function myFunction() {
  console.log("Good Morning");
}
```

Passing an Anonymous Function to setTimeout()

- Instead of passing the name of a function, you can pass an anonymous function directly.

```
setTimeout(function() {
  myFunction("Good Morning");
}, 3000);

function myFunction(value) {
  console.log(value);
}
```

Using setInterval() for Repeated Execution

- The setInterval() function executes a function repeatedly at specified intervals.

```
setInterval(myFunction, 1000);

function myFunction() {
  let d = new Date();
  console.log(
    d.getHours() + ":" +
    d.getMinutes() + ":" +
    d.getSeconds());
}
```

Callback Alternatives: Promises

- Asynchronous programming can be complex and hard to debug when using callbacks. Modern JavaScript uses Promises to handle asynchronous operations more elegantly.

Example of Using Promises

Here's how you might rewrite an asynchronous operation using Promises:

```
function myDisplayer(something) {
  console.log(something);
}
```

```
function myCalculator(num1, num2) {
  return new Promise((resolve, reject) => {
    let sum = num1 + num2;
    if (isNaN(sum)) {
      reject("Calculation error");
    } else {
      resolve(sum);
    }
  });
}

myCalculator(5, 5)
  .then(result => myDisplayer(result))
  .catch(error => console.error(error));
```

Example of Using setTimeout() with Promises

- You can also wrap setTimeout() in a Promise to make it easier to work with:

```
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

delay(3000).then(() => {
  console.log("Good Morning");
});
```

Example of Using setInterval() with Promises

- While setInterval() isn't typically used with Promises directly, you can create a function to handle repeated tasks with Promises.

```
function repeatEvery(func, interval) {
  return new Promise((resolve, reject) => {
    const intervalId = setInterval(() => {
      try {
        func();
      } catch (error) {
        clearInterval(intervalId);
        reject(error);
      }
    }, interval);
  });
}

repeatEvery(() => {
  let d = new Date();
  console.log(d.getHours() + ":" + d.getMinutes() + ":" + d.getSeconds());
}, 1000);
```

JavaScript Promises

Promises in JavaScript provide a more robust way to handle asynchronous operations compared to callbacks. They represent a value that might be available now, or in the future, or never.

Promise Syntax

- A Promise object is created using the `Promise` constructor, which takes a function (executor) with two arguments: `resolve` and `reject`.

```
let myPromise = new Promise(function(myResolve, myReject) {
  // "Producing Code" (May take some time)
  myResolve(); // when successful
  myReject();  // when error
});
```

Consuming a Promise

- You handle the results of a Promise using the `then` method, which takes two callbacks: one for when the Promise is fulfilled and one for when it's rejected.

```
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

Promise States

A Promise can be in one of three states:

1. **Pending**: Initial state, neither fulfilled nor rejected.
2. **Fulfilled**: The operation completed successfully.
3. **Rejected**: The operation failed.

```
let myPromise = new Promise(function(myResolve, myReject) {
  let x = 0;

  // The producing code (this may take some time)
  if (x == 0) {
    myResolve("OK");
  } else {
    myReject("Error");
  }
});
```

```
myPromise.then(
  function(value) { console.log(value); },  // "OK"
  function(error) { console.log(error); }  // "Error"
);
```

Example: Waiting for a Timeout

Using Callback

```
setTimeout(function() {
  myFunction("Good Morning");
}, 3000);

function myFunction(value) {
  console.log(value);
}
```

Using Promise

```
let myPromise = new Promise(function(myResolve, myReject) {
  setTimeout(function() {
    myResolve("Good Morning");
  }, 3000);
});

myPromise.then(function(value) {
  console.log(value);
});
```

Example: Waiting for a File

Using Callback

```
// note: npm install node-fetch
const http = require('http');

function getFile(myCallback) {
  http.get('http://example.com/mycar.html', (res) => {
    let data = '';

    // A chunk of data has been received.
    res.on('data', (chunk) => {
      data += chunk;
    });

    // The whole response has been received.
```

```
      res.on('end', () => {
        if (res.statusCode === 200) {
          myCallback(data);
        } else {
          myCallback(`Error: ${res.statusCode}`);
        }
      });
    }).on('error', (err) => {
      myCallback(`Error: ${err.message}`);
    });
  }

  getFile(function(response) {
    console.log(response);
  });
```

Using Promise

```
// note:npm install node-fetch
const fetch = require('node-fetch');

let myPromise = new Promise(function(myResolve, myReject) {
  fetch('http://example.com/mycar.html') // Replace with the actual URL
    .then(response => {
      if (response.ok) {
        return response.text();
      } else {
        throw new Error('File not Found');
      }
    })
    .then(text => myResolve(text))
    .catch(error => myReject(error));
});

myPromise.then(
  function(value) { console.log(value); },
  function(error) { console.log(error.message); }
);
```

Key Points

- **Creating Promises**: Use the `Promise` constructor with `resolve` and `reject` functions.
- **Handling Promises**: Use the `then` method for success and error handling.
- **States**: Understand the three states of a Promise - pending, fulfilled, and rejected.

JavaScript Async Await

- The `async` and `await` keywords in JavaScript provide a more readable and straightforward way to work with Promises, allowing asynchronous code to be written as if it were synchronous.

`async` Keyword

- The `async` keyword is used to define an asynchronous function. An async function returns a Promise.

Example

```
async function myFunction() {
  return "Hello";
}
```

This is equivalent to:

```
function myFunction() {
  return Promise.resolve("Hello");
}
```

You can consume the Promise returned by an async function using `then`:

```
myFunction().then(
  function(value) { console.log(value); }, // "Hello"
  function(error) { console.log(error); }
);
```

A simplified version, if you only expect a normal value:

```
myFunction().then(function(value) {
  console.log(value); // "Hello"
});
```

`await` Keyword

- The `await` keyword can only be used inside an `async` function. It pauses the execution of the async function and waits for the Promise to resolve.

Basic Syntax

```
async function myDisplay() {
  let myPromise = new Promise(function(resolve) {
    resolve("Good Morning");
  });
  console.log(await myPromise);
}

myDisplay();
```

In the example above, `myDisplay` is an async function that waits for `myPromise` to resolve and then sets the inner HTML of the element with id "demo" to the resolved value.

Example without `reject`

Often, you may not need to handle rejection explicitly.

```
async function myDisplay() {
  let myPromise = new Promise(function(resolve) {
    resolve("Good Morning");
  });
  console.log(await myPromise);
}

myDisplay();
```

Waiting for a Timeout

You can use `await` to wait for a timeout:

```
async function myDisplay() {
  let myPromise = new Promise(function(resolve) {
    setTimeout(function() {
      resolve("Good Morning");
    }, 3000);
  });
  console.log(await myPromise); // Output to the console instead of the DOM
}

myDisplay();
```

Waiting for a File

Here's an example of using `await` to wait for a file to load:

```
// note: npm install node-fetch
const fetch = require('node-fetch');
const fs = require('fs');

async function getFile() {
  try {
    const response = await fetch('http://example.com/mycar.html'); // Replace with the correct URL
    if (response.ok) {
      const data = await response.text();
      console.log(data); // For demonstration in Node.js, logging to console
    } else {
      console.log('File not Found');
    }
  } catch (error) {
    console.error('Error fetching file:', error);
  }
}

getFile();
```

In this example, `getFile` is an async function that waits for the response of an HTTP request to fetch a file. If the request is successful, it resolves with the response; otherwise, it resolves with "File not Found".

Key Points

- `async` **Function**: An async function always returns a Promise.
- `await` **Expression**: `await` pauses the execution of the async function and waits for the Promise to resolve.

Modules

The most apparent difference between CommonJS and ES Modules is the syntax used for importing and exporting modules.

CommonJS: Utilizes the require() function for importing modules and module.exports or exports for exporting. This syntax is straightforward and familiar to many JavaScript developers, especially those with a background in Node.js.

```
// Importing with CommonJS
const express = require('express');

// Exporting with CommonJS
module.exports = function() {
  // Some functionality
};
```

ES Modules: Uses the import statement for importing and the export statement for exporting. This syntax is more declarative and supports importing and exporting multiple values, as well as renaming imports and exports.

```
// Importing with ES Modules
import express from 'express';

// Exporting with ES Modules
export function myFunction() {
  // Some functionality
};
```

Global and Module Scope in JavaScript

In addition to block scopes, variables can be scoped to the global and module scope.

In a web browser, the global scope is at the top level of a script. It is the root of the scope tree that we described earlier, and it contains all other scopes. Thus, creating a variable in the global scope makes it accessible in every scope:

A simple Node.js script that demonstrates the use of ES modules (`import` and `export`).

Project Structure

Create a directory for your project. Inside this directory, create two files:

1. module.js
2. main.js

Your project structure should look like this:

```
your-project-directory/
│
├── module.js
├── main.js
└── package.json
```

1. Initialize an npm Project

```
cd your-project-directory
npm init -y
```

2. Update `package.json` to Use ES Modules

In package.json update type as module to use ES Modules

```
{
  "name": "module",
  "version": "1.0.0",
  "main": "main.js",
  "type": "module",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}
```

3. Create `module.js`

```
// module.js

export const foo = "foo";
```

4. Create `main.js`

```
// main.js

import { foo } from './module.js';

console.log(foo); // Should log "foo"

function bar() {
  if (true) {
    console.log(foo);
  }
}

bar(); // Should log "foo"
```

Running the Script

1. Navigate to Your Project Directory:

Open your terminal and navigate to the directory where your project files are located.

```
cd your-project-directory
```

2. Run the Script:

Use `Node.js` to run `main.js`.

```
node main.js
```

Expected Output:

When you run the script, you should see the following output:

```
foo
foo
```

Using require for Modules (CommonJS)

In Node.js, the module system is based on CommonJS. Variables declared within a module are not added to the global scope but to the module scope. You use module.exports to export variables or functions from a module and require to import them.

Simple Node.js script that demonstrates the use of require to import a module and module.exports to export a constant.

Project Structure

Create a directory for your project. Inside this directory, create two files:

```
your-project-directory/
│
├── module.js
└── main.js
```

1. Create `module.js`

This file will export a constant `foo`.

```
// module.js

const foo = "foo";
module.exports = foo;
```

2. Create `main.js`

This file will import the constant foo from `module.js` and use it.

```
// main.js

const foo = require('./module.js');

console.log(foo); // Should log "foo"

function bar() {
  if (true) {
    console.log(foo);
  }
}

bar(); // Should log "foo"
```

Running the Script

1. Navigate to Your Project Directory:

Open your terminal and navigate to the directory where your project files are located.

```
cd your-project-directory
```

2. Run the Script:

Use `Node.js` to run `main.js`.

```
node main.js
```

Expected Output:

When you run the script, you should see the following output:

```
foo
foo
```

REST Api

A Representational State Transfer (REST) API is a widely adopted architectural style for designing networked applications. RESTful APIs are known for their simplicity, scalability, and ability to leverage existing web technologies.

Key Concepts of RESTful APIs:

1. `Resources:`

- Central to REST is the concept of resources, which are the key abstractions. Resources can represent entities such as users, products, or articles.
- Resources are identified by Uniform Resource Identifiers (URIs) or URLs. Each resource has a unique URI.
- Resources can have multiple representations, such as JSON or XML, to support different data formats.

2. `HTTP Methods:`

- RESTful APIs use HTTP methods (also known as verbs) to define actions on resources.
- Common HTTP methods include:
  - GET: Retrieve resource data.
  - POST: Create a new resource.
  - PUT: Update an existing resource (or create if it doesn't exist).
  - DELETE: Remove a resource.
- HTTP methods are idempotent, meaning that repeated requests have the same effect as a single request (except for POST).

3. `Statelessness:`

- REST APIs are stateless, meaning that each request from a client to a server must contain all the information needed to understand and process the request.
- No session state is stored on the server between requests. Session state, if needed, should be managed on the client side.

4. `Representations:`

- Resources can have multiple representations, such as JSON, XML, or HTML.
- Clients can request specific representations by specifying the desired media type in the request headers (e.g., Accept: application/json).

Characteristics of RESTful APIs:

1. `Uniform Interface:`

- A RESTful API should have a uniform and consistent interface, which simplifies client and server interactions.
- It follows a small set of well-defined conventions, such as standard HTTP methods and status codes.

2. `Statelessness:`

- Each request from a client to a server must contain all the information needed to understand and process the request.
- This allows for easy scalability and load balancing since each request can be treated independently.

3. `Resource-Based:`

- Resources are the central abstractions in RESTful APIs, and they are represented by URIs.
- Resources are manipulated using standard HTTP methods.

4. `Representation:`

- Resources can have multiple representations (e.g., JSON, XML) to support different client needs.
- The client can specify the desired representation using the Accept header.

5. `Self-Descriptive Messages:`

- Responses from the server should include metadata (e.g., links to related resources) to help clients navigate the API.

6. `Stateless Communication:`

- Each request from a client to a server must contain all the necessary information. The server should not store client state between requests.

Status Codes:

HTTP status codes are used to indicate the outcome of a request. Common status codes in RESTful APIs include:

- `200 OK:` Successful request.
- `201 Created:` Resource successfully created.
- `204 No Content:` Successful request with no response body.

- `400 Bad Request:` Invalid request by the client.
- `401 Unauthorized:` Authentication required or failed.
- `403 Forbidden:` Authentication succeeded, but the client doesn't have permission.
- `404 Not Found:` Resource not found.
- `405 Method Not Allowed:` Requested HTTP method is not supported for the resource.
- `500 Internal Server Error:` Server error.

Best Practices:

1. `Use Plural Nouns for Resource Names:` Use plural nouns (e.g., /users, /products) for resource endpoints to indicate collections.

2. `Versioning:` Include API versioning in the URI (e.g., /v1/users) to allow for backward compatibility as the API evolves.

3. `Use HTTP Status Codes Correctly:` Return appropriate HTTP status codes to convey the result of a request.

4. `Pagination:` Implement pagination for resource collections to handle large datasets efficiently (e.g., /users?page=2&limit=10).

5. `Authentication and Authorization:` Implement secure authentication and authorization mechanisms.

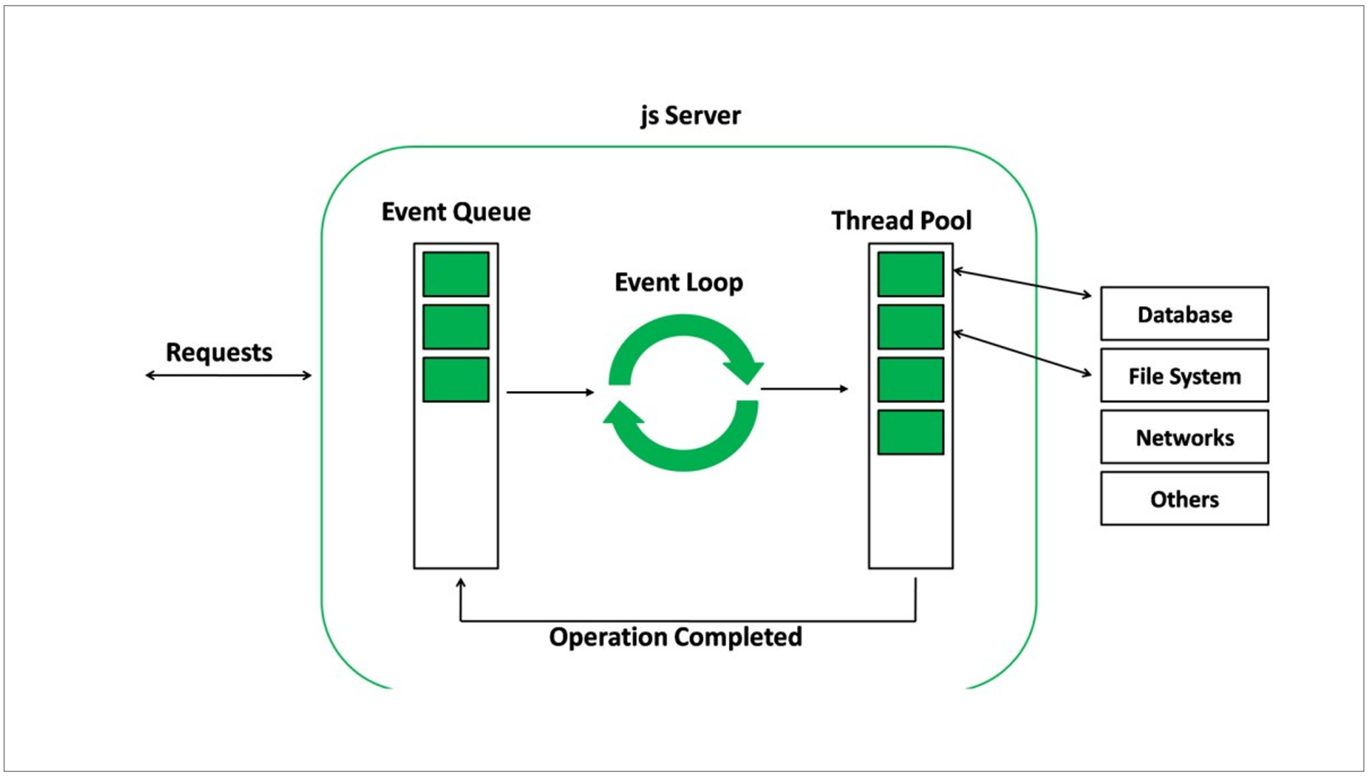
6. `Rate Limiting:` Consider implementing rate limiting to prevent abuse of the API.

7. `Documentation:` Provide comprehensive API documentation to help developers understand how to use your API.

8. `HATEOAS (Hypermedia as the Engine of Application State):` Optionally, use links in responses to allow clients to navigate the API dynamically.

Creating a server

In Node.js, you can create a streaming HTTP server using the built-in http module. Streaming HTTP servers allow you to efficiently handle large volumes of data by using streams to send and receive data in chunks, making them ideal for building real-time or data-intensive web applications.



```js
const http = require("http");

const PORT = 3000;
const HOST = "localhost";

const server = http.createServer((req, res) => {
  // Set response headers
  res.setHeader("Content-Type", "text/plain");

  // Write data to the response stream
  res.write("Hello, ");
  res.write("World!");

  // End the response stream
  res.end();
});

server.listen(PORT, HOST, () => {
  console.log(`Server is running at http://${HOST}:${PORT}`);
});
```

Output:

```
Server is running at http://localhost:3000
```

Import the http module. Create an HTTP server using the http.createServer() method. Specify the port and host to listen on, and start the server using the command `node index.js`(whichever the name of the file).

Http-Client

Http Client Request to consume the Http Request

```
const http = require('http');

const options = {
  hostname: 'localhost',
  port: 3000,
  path: '/',
  method: 'GET',
  headers: {
    'Content-Type': 'text/plain'
  }
};

const req = http.request(options, (res) => {
  let data = '';

  // Set the encoding of the response to UTF-8
  res.setEncoding('utf8');

  // Collect data chunks as they come in
  res.on('data', (chunk) => {
    data += chunk;
  });

  // When the whole response is received, print it out
  res.on('end', () => {
    console.log('Response from server:', data);
  });
});

// Handle request errors
req.on('error', (e) => {
  console.error(`Problem with request: ${e.message}`);
});

// End the request
req.end();
```

How HTTP Module Uses Streams for API Requests

The http module in Node.js uses streams internally to handle API requests and responses efficiently. When a client sends a request to your HTTP server, the server processes the request using streams. Here's a simplified overview of how it works:

1. `Request Stream:` When a client makes an API request (e.g., a GET request), the incoming data from the client is processed as a readable stream. This allows the server to read the request data in chunks, making it suitable for handling large request payloads.

2. `Response Stream:` When your server responds to the client, it uses a writable stream to send data back in chunks. This is especially useful for streaming data to the client as it becomes available, rather than waiting for the entire response to be generated.

3. `Efficiency:` Streams allow your server to efficiently process data without loading the entire request or response into memory at once. This is crucial for handling large files, streaming real-time data, or handling multiple concurrent requests.

Here's a simplified example of how the http module handles a request using streams:

```
const http = require("http");

const server = http.createServer((req, res) => {
  // Incoming data from the client is a readable stream (req)
  req.on("data", (chunk) => {
    // Process the request data in chunks
    console.log("Received data chunk:", chunk.toString());
  });

  // Outgoing data to the client is a writable stream (res)
  // Sending a response in chunks
  res.write("Hello, ");
  res.write("World!");
  res.end();
});

server.listen(3000, () => {
  console.log("Server is running at http://localhost:3000");
});
```

Output:

```
Hello, World!
```

In this example, the server processes incoming request data in chunks and responds to the client using writable streams. This approach ensures that the server can handle large requests and efficiently stream data back to clients.

Https Module

The https module in Node.js allows you to create secure HTTP servers and make secure client requests using the HTTPS protocol. It builds upon the functionality of the built-in http module while adding encryption and security features. HTTPS servers provide additional security by encrypting data that is being transmitted from a server to a client, plus it uses SSL/TLS certificates to verify the client. It is used widely on the internet, for secure communication over a network

HTTPS uses the TLS or Transport Layer Security encryption protocol, which was formerly known as Secure Sockets Layer or SSL. The encryption protocols of HTTPS give the ability to users on the web, to transfer or pass on sensitive information such as credit card numbers, banking information, login credentials, and more in a secure way.

Advantages of HTTPS Over HTTP:

HTTPS (Hypertext Transfer Protocol Secure) offers several advantages over HTTP:

1. `Data Encryption:` HTTPS encrypts the data transmitted between the client and server, making it difficult for attackers to intercept and read sensitive information.

2. `Data Integrity:` It ensures that the data remains unchanged during transmission. Any tampering with the data will result in an error.

3. `Authentication:` HTTPS provides authentication through digital certificates, verifying the identity of the server to the client, reducing the risk of man-in-the-middle attacks.

4. `SEO Benefits:` Search engines often favor HTTPS websites, resulting in improved search engine rankings

Certificates

There are two kinds certificates; those signed by a ‘Certified Author’, also known as CA, and those that are ‘Self-signed’.

A CA-signed certificate allows your users to trust the identity of your website and is recommended for the production environment. For testing or development purposes, a Self-signed certificate will do just fine.

Installing openssl

`Windows:`

1. Install Chocolatey from here.
2. Open powershell with admin access.
3. Run `Set-ExecutionPolicy AllSigned` command.
4. Run `Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::SecurityProtocol = [System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object System.Net.WebClient).DownloadString('https://community.chocolatey.org/install.ps1'))` to install Chocolatey package manager.
5. Run `choco` to verify the installation
6. Run `choco install openssl` to install openssl cli tool.
7. Open a new powersheel instance and Run `openssl version` to verify the installation.

`Mac:`

1. Run `brew uninstall openssl` to install openssl.

`Linux`

1. Run `sudo apt install openssl` to install openssl.

Generating self signed certificate

```
openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 -keyout privateKey.key
-out certificate.crt
```

This command generates a self-signed certificate (certificate.crt) and a private key (privateKey.key) valid for 365 days. You can use these files for your local HTTPS server.

Creating a https server

```
const https = require("https");
const fs = require("fs");

// Load the certificate and private key
const privateKey = fs.readFileSync("privateKey.key", "utf8");
const certificate = fs.readFileSync("certificate.crt", "utf8");

const credentials = { key: privateKey, cert: certificate };

const server = https.createServer(credentials, (req, res) => {
  res.writeHead(200, { "Content-Type": "text/plain" });
  res.end("Secure Hello, World!\n");
});

server.listen(443, () => {
  console.log("HTTPS Server is running on port 443");
});
```

In this example:

- We load the self-signed certificate and private key.
- We create an HTTPS server using https.createServer() and provide the certificate and private key as credentials.
- We define a request handler to respond with "Secure Hello, World!".
- The server listens on port 443, the default port for HTTPS.

If you use a self-signed certificate, your browser may display a security warning because the certificate is not trusted. To bypass this, you can add the certificate to your browser's trust store or use tools like Postman that allow you to accept self-signed certificates for testing purposes.

Make a call from postman to `https://localhost:443` to get the response `Secure Hello, World!`.

Requesting data with http https Module

Making GET Requests:

1. Import the required module (http for HTTP or https for HTTPS).

2. Specify the request options, including the host, path, and method.

3. Use the module's request() method to send the GET request.

4. Handle the response data using event listeners.

```
const http = require("http");

const options = {
  hostname: "jsonplaceholder.typicode.com",
  path: "/posts/1",
  method: "GET",
};

const req = http.request(options, (res) => {
  let data = "";

  res.on("data", (chunk) => {
    data += chunk;
  });

  res.on("end", () => {
    console.log("GET Response:", JSON.parse(data));
  });
});

req.on("error", (error) => {
  console.error("Request error:", error);
});

req.end();
```

To make a GET request using the https module, simply replace http with https in the code above.

Making POST Requests:

1. Import the required module (http for HTTP or https for HTTPS).
2. Specify the request options, including the host, path, method (POST), headers, and the data to be sent in the request body.
3. Use the module's request() method to send the POST request.
4. Handle the response data using event listeners.

```
const https = require("https");

const options = {
  hostname: "jsonplaceholder.typicode.com",
  path: "/posts",
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
};
```

```
const postData = JSON.stringify({
  title: "Sample Post",
  body: "This is a sample post request.",
  userId: 1,
});

const req = https.request(options, (res) => {
  let data = "";

  res.on("data", (chunk) => {
    data += chunk;
  });

  res.on("end", () => {
    console.log("POST Response:", JSON.parse(data));
  });
});

req.on("error", (error) => {
  console.error("Request error:", error);
});

req.write(postData);
req.end();
```

In this example, we send a POST request with JSON data. Adjust the postData variable to contain the data you want to send in your request.

Working with Routes

Routes are the endpoints in api. For example `/users`, `/home`, `/about`, `/login` etc. You can define routes and whenever a request is made to a particular endpoint, you can program the server to return a particular response.

```
const http = require("http");

const server = http.createServer((req, res) => {
  if (req.url === "/") {
    // Handle the root route (e.g., display a homepage)
    res.writeHead(200, { "Content-Type": "text/plain" });
    res.end("Welcome to the homepage!");
  } else if (req.url === "/about") {
    // Handle the "/about" route (e.g., display an about page)
    res.writeHead(200, { "Content-Type": "text/plain" });
    res.end("About us: We are an awesome company!");
  } else {
    // Handle 404 - Not Found for all other routes
    res.writeHead(404, { "Content-Type": "text/plain" });
    res.end("404 - Not Found");
  }
});

server.listen(3000, () => {
  console.log("Server is running on port 3000");
});
```

Output:

```
Welcome to the homepage!
```

Start the server and make request to `localhost:3000/` to get `Welcome to the homepage!` response. This is how you define a route and handle the response.

Parsing URLs and Query Strings

In Node.js, you can parse URLs and query strings using the built-in http module and the url module. Parsing URLs is essential when working with web applications, as it allows you to extract information from the request URL and query parameters.

We can use query string to send data to the server. We can return the appropriate data back to client based on the request data.

Parsing URLs and Query string with the url Module:

The url module in Node.js provides utilities for URL resolution and parsing. You can use it to parse URLs from incoming HTTP requests. Here's how to do it:

```
const http = require("http");
const url = require("url");

const server = http.createServer((req, res) => {
  // Parse the request URL
  const parsedUrl = url.parse(req.url, true);

  // Extract information from the parsed URL
  const pathname = parsedUrl.pathname; // URL path
  const query = parsedUrl.query; // Query parameters as an object

  res.writeHead(200, { "Content-Type": "text/plain" });

  // Respond with parsed information
  res.end(`Path: ${pathname}\nQuery: ${JSON.stringify(query)}`);
});

server.listen(3000, () => {
  console.log("Server is running on port 3000");
});
```

In this example:

1. We import the url module along with the http module.
2. Inside the request handler, we parse the request URL using url.parse().
3. We pass the req.url and true as arguments to url.parse() to also parse the query string into an object.
4. We extract the URL path and query parameters from the parsed URL and respond with them.

From the browser call the url `localhost:3000/getUser??name=John&age=30`

Output:

```
Path: /some/path
Query: {"name":"John","age":"30"}
```

CRUD app

This guide will help you understand how to create, run, and interact with a simple CRUD (Create, Read, Update, Delete) application for managing a list of todos. The application will use a JSON file (data.json) to store todos and will provide a REST API to interact with the todos.

Project Structure

```
my-crud-app/
├── data.json
└── index.js
```

```
{
  "todos": [
    { "id": 1, "task": "Buy groceries", "completed": false },
    { "id": 2, "task": "Walk the dog", "completed": true }
  ]
}
```

```
const http = require("http");
const fs = require("fs");

const PORT = 3000;
const dataFilePath = "./data.json";

const server = http.createServer((req, res) => {
  if (req.method === "GET" && req.url === "/todos") {
    // Read todos from data.json and send as JSON response
    fs.readFile(dataFilePath, "utf-8", (err, data) => {
      if (err) {
        console.error("Error reading file:", err);
```

```
          res.writeHead(500, { "Content-Type": "application/json" });
          res.end(JSON.stringify({ error: "Internal Server Error" }));
          return;
        }

        try {
          const todos = JSON.parse(data).todos;
          res.writeHead(200, { "Content-Type": "application/json" });
          res.end(JSON.stringify(todos));
        } catch (parseErr) {
          console.error("Error parsing JSON:", parseErr);
          res.writeHead(500, { "Content-Type": "application/json" });
          res.end(JSON.stringify({ error: "Internal Server Error" }));
        }
      });
    } else if (req.method === "POST" && req.url === "/todos") {
      // Create a new todo
      let requestBody = "";
      req.on("data", (chunk) => {
        requestBody += chunk.toString();
      });
      req.on("end", () => {
        try {
          const newTodo = JSON.parse(requestBody);

          fs.readFile(dataFilePath, "utf-8", (err, data) => {
            if (err) {
              console.error("Error reading file:", err);
              res.writeHead(500, { "Content-Type": "application/json" });
              res.end(JSON.stringify({ error: "Internal Server Error" }));
              return;
            }

            try {
              const todos = JSON.parse(data).todos;
              const id = todos.length + 1;
              newTodo.id = id;
              todos.push(newTodo);

              fs.writeFile(dataFilePath, JSON.stringify({ todos }), (writeErr) => {
                if (writeErr) {
                  console.error("Error writing file:", writeErr);
                  res.writeHead(500, { "Content-Type": "application/json" });
                  res.end(JSON.stringify({ error: "Internal Server Error" }));
                  return;
                }

                res.writeHead(201, { "Content-Type": "application/json" });
                res.end(JSON.stringify(newTodo));
              });
            } catch (parseErr) {
              console.error("Error parsing JSON:", parseErr);
              res.writeHead(500, { "Content-Type": "application/json" });
              res.end(JSON.stringify({ error: "Internal Server Error" }));
            }
          });
        } catch (parseErr) {
          console.error("Error parsing request body JSON:", parseErr);
          res.writeHead(400, { "Content-Type": "application/json" });
          res.end(JSON.stringify({ error: "Bad Request" }));
        }
      });
    } else if (req.method === "PUT" && req.url.startsWith("/todos/")) {
      // Update a todo by ID
      const todoId = parseInt(req.url.split("/")[2]);

      let requestBody = "";
      req.on("data", (chunk) => {
        requestBody += chunk.toString();
      });
      req.on("end", () => {
        try {
          const updatedTodo = JSON.parse(requestBody);

          fs.readFile(dataFilePath, "utf-8", (err, data) => {
            if (err) {
              console.error("Error reading file:", err);
              res.writeHead(500, { "Content-Type": "application/json" });
              res.end(JSON.stringify({ error: "Internal Server Error" }));
              return;
            }

            try {
              const todos = JSON.parse(data).todos;
              const existingTodo = todos.find((todo) => todo.id === todoId);

              if (!existingTodo) {
                res.writeHead(404, { "Content-Type": "application/json" });
                res.end(JSON.stringify({ error: "Todo not found" }));
                return;
              }

              Object.assign(existingTodo, updatedTodo);

              fs.writeFile(dataFilePath, JSON.stringify({ todos }), (writeErr) => {
                if (writeErr) {
                  console.error("Error writing file:", writeErr);
                  res.writeHead(500, { "Content-Type": "application/json" });
                  res.end(JSON.stringify({ error: "Internal Server Error" }));
                  return;
                }

                res.writeHead(200, { "Content-Type": "application/json" });
                res.end(JSON.stringify(existingTodo));
              });
            } catch (parseErr) {
              console.error("Error parsing JSON:", parseErr);
              res.writeHead(500, { "Content-Type": "application/json" });
              res.end(JSON.stringify({ error: "Internal Server Error" }));
            }
          });
        } catch (parseErr) {
          console.error("Error parsing request body JSON:", parseErr);
          res.writeHead(400, { "Content-Type": "application/json" });
          res.end(JSON.stringify({ error: "Bad Request" }));
        }
      });
    } else if (req.method === "DELETE" && req.url.startsWith("/todos/")) {
      // Delete a todo by ID
      const todoId = parseInt(req.url.split("/")[2]);

      fs.readFile(dataFilePath, "utf-8", (err, data) => {
        if (err) {
          console.error("Error reading file:", err);
          res.writeHead(500, { "Content-Type": "application/json" });
          res.end(JSON.stringify({ error: "Internal Server Error" }));
          return;
        }

        try {
          const todos = JSON.parse(data).todos;
          const indexToDelete = todos.findIndex((todo) => todo.id === todoId);

          if (indexToDelete === -1) {
            res.writeHead(404, { "Content-Type": "application/json" });
            res.end(JSON.stringify({ error: "Todo not found" }));
            return;
          }

          const deletedTodo = todos.splice(indexToDelete, 1)[0];

          fs.writeFile(dataFilePath, JSON.stringify({ todos }), (writeErr) => {
            if (writeErr) {
              console.error("Error writing file:", writeErr);
              res.writeHead(500, { "Content-Type": "application/json" });
              res.end(JSON.stringify({ error: "Internal Server Error" }));
              return;
            }

            res.writeHead(200, { "Content-Type": "application/json" });
            res.end(JSON.stringify(deletedTodo));
          });
        } catch (parseErr) {
          console.error("Error parsing JSON:", parseErr);
          res.writeHead(500, { "Content-Type": "application/json" });
          res.end(JSON.stringify({ error: "Internal Server Error" }));
        }
      });
    } else {
      // Handle other routes or methods
      res.writeHead(404, { "Content-Type": "application/json" });
      res.end(JSON.stringify({ error: "Not Found" }));
    }
  });

  server.listen(PORT, () => {
    console.log(`Server is running on port ${PORT}`);
  });
```

To run CRUD App

```
node index.js
```

```
http://localhost:3000/todos
```

We check the type of call using `req.method`. Based on that we perform CURD operation on todos. We use `fs` module to read and write data to `data.json` file. This way implementation is hard to read and maintain. Node.js ecosystem has many packages that simplifies this process and provides tons of features. One such package is `express.js`. Lets explore more in the next section.

Express.js

What is Express js & Why Express js

Express.js, often referred to as Express, is a widely used and highly popular web application framework for Node.js. It simplifies the process of building scalable and robust web applications by providing a range of powerful features and tools.

Express.js is a minimal and flexible Node.js web application framework designed for building web applications and APIs. It is built on top of the core HTTP module in Node.js, which means it leverages the low-level capabilities of Node.js while providing a more structured and feature-rich environment for developing web applications.

Key characteristics and features of Express.js include:

1. `Middleware:` Express.js is known for its middleware architecture. Middleware functions can be thought of as a chain of functions that process incoming HTTP requests before they reach the application's routes. Middleware can perform tasks like authentication, logging, data parsing, and more.

2. `Routing:` Express provides a simple and powerful routing system that allows developers to define routes based on HTTP methods and URL patterns. This makes it easy to create RESTful APIs and define routes for handling various HTTP requests.

3. `Template Engines:` Although not bundled with Express itself, developers can easily integrate template engines like EJS, Handlebars, or Pug (formerly known as Jade) to generate dynamic HTML content. This is useful for rendering views on the server side.

4. `HTTP Utility Methods:` Express simplifies working with HTTP methods and status codes. It provides methods for common HTTP operations like GET, POST, PUT, DELETE, and more. This helps in writing concise and readable code.

5. `Error Handling:` Express includes error handling mechanisms to gracefully handle errors that may occur during the request-response cycle. This ensures that applications remain stable and provide meaningful error messages.

6. `Session Management:` Although not included in the core package, Express can be extended with middleware like express-session to manage user sessions and handle session-based authentication.

7. `Static File Serving:` Express can serve static files such as CSS, JavaScript, and images with ease, making it suitable for both API development and serving client-side applications.

Why Express.js?

Developers choose Express.js for various reasons, making it one of the most popular choices for building web applications and APIs:

1. `Simplicity:` Express.js follows a minimalistic and unopinionated approach, allowing developers to choose the libraries and tools they prefer. It provides essential features while giving developers the freedom to structure their applications as they see fit.

2. `Middleware Ecosystem:` Express's middleware ecosystem is extensive, offering a wide range of pre-built middleware packages that simplify common tasks. Developers can also create custom middleware to suit their specific needs.

3. `Community and Documentation:` Express has a large and active community, which means there are numerous resources, tutorials, and libraries available to assist developers. The official documentation is comprehensive and well-maintained.

4. `Flexibility:` Express.js can be used to build a variety of web applications, including APIs, single-page applications (SPAs), and traditional server-rendered web applications. Its flexibility allows it to adapt to different project requirements.

5. `Performance:` Express.js is known for its high performance and low overhead. It is designed to handle a large number of concurrent connections efficiently, making it suitable for high-traffic applications.

6. `Compatibility:` Express can be used in conjunction with other Node.js libraries and frameworks. This allows developers to integrate additional functionality seamlessly.
7. `Scalability:` Express.js provides a foundation for building scalable applications. It doesn't impose rigid application structures, giving developers the freedom to design their applications to scale as needed.
8. `Mature and Stable:` Express has been around for years and has a proven track record in the industry. Many well-known companies and projects use Express for their web applications.

Setting up express app

1. Initialize a Node.js project:

```
mkdir my-express-app
cd my-express-app
npm init -y
```

Creates a folder by name `my-express-app`. `npm init -y` creates `package.json` file and instantiates a project.

2. Install Express:

```
npm install express
```

3. Create an `index.js` file.

```
import express from "express";
const app = express(); // application object

// Define a route
app.get("/", (req, res) => {
  res.send("Hello, Express!");
});

// Start the server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

The app object in Express.js is an instance of the Express application. It provides methods to configure routes, handle HTTP requests and responses, set up middleware, and more.

Let's break down the key parts of the code:

1. `Import Express:` First, we import the Express module.

2. `Create an Express App:` We create an instance of the Express application and assign it to the app variable.

3. `Define a Route:` Using the app.get() method, we define a route for handling HTTP GET requests to the root path '/'. When a GET request is made to the root URL, the provided callback function is executed, sending the response 'Hello, Express!'.

4. `Start the Server:` We specify a port number (in this case, 3000) and use the app.listen() method to start the Express server. The server will listen on the specified port, and a message is logged to the console when the server starts.

Routes

This guide provides a step-by-step approach to setting up a CRUD API for "todos" using Express and documenting it with Swagger.

```
my-express-app/
├── node_modules/
├── todos.js
├── swagger.js
└── package.json
```

Step 1: Initialize the Project

Initialize your project and install the required dependencies:

```
npm init -y
npm install express swagger-jsdoc swagger-ui-express
```

Step 2: Configure `package.json`

Ensure your `package.json` has the following structure to support ES modules:

```
{
  "name": "my-express-app",
  "version": "1.0.0",
  "type": "module",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.19.2",
    "node-fetch": "^3.3.2",
    "swagger-jsdoc": "^6.2.8",
    "swagger-ui-express": "^5.0.0"
  }
}
```

Step 3: Define Routes (todos.js)

Create a separate file for todos routes.

```
import express, { json } from "express";
import { serve, setup } from "swagger-ui-express";
import swaggerSpecs from './swagger.js';
const app = express();
const PORT = 3000;

// Sample data - stored in a local variable (for simplicity)
let todos = [
  { id: 1, task: "Buy groceries", completed: false },
  { id: 2, task: "Walk the dog", completed: true },
  // Add more todos here
];

app.use(json());

// Swagger UI setup
app.use("/api-docs", serve, setup(swaggerSpecs));

/**
 * @swagger
 * components:
 *   schemas:
 *     Todo:
 *       type: object
 *       required:
 *         - task
 *         - completed
 *       properties:
 *         id:
 *           type: integer
 *           description: The auto-generated id of the todo item
 *         task:
 *           type: string
 *           description: The task description
 *         completed:
 *           type: boolean
 *           description: The status of the task
 *       example:
 *         id: 1
 *         task: Buy groceries
 *         completed: false
 */
```

```
/**
 * @swagger
 * tags:
 *   name: Todos
 *   description: The todos managing API
 */

/**
 * @swagger
 * /todos:
 *   get:
 *     summary: Returns the list of all the todos
 *     tags: [Todos]
 *     responses:
 *       200:
 *         description: The list of the todos
 *         content:
 *           application/json:
 *             schema:
 *               type: array
 *               items:
 *                 $ref: '#/components/schemas/Todo'
 */
app.get("/todos", (req, res) => {
  res.json(todos);
});

/**
 * @swagger
 * /todos/{id}:
 *   get:
 *     summary: Get the todo by id
 *     tags: [Todos]
 *     parameters:
 *       - in: path
 *         name: id
 *         schema:
 *           type: integer
 *         required: true
 *         description: The todo id
 *     responses:
 *       200:
 *         description: The todo description by id
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/Todo'
 *       404:
 *         description: The todo was not found
 */
app.get("/todos/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const todo = todos.find((t) => t.id === id);
  if (!todo) return res.status(404).json({ error: "Todo not found" });
  res.json(todo);
});

/**
 * @swagger
 * /todos:
 *   post:
 *     summary: Create a new todo
 *     tags: [Todos]
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             $ref: '#/components/schemas/Todo'
 *     responses:
 *       201:
 *         description: The todo was successfully created
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/Todo'
 *       500:
 *         description: Some server error
 */
app.post("/todos", (req, res) => {
  const newTodo = req.body;
  newTodo.id = todos.length + 1;
  todos.push(newTodo);
  res.status(201).json(newTodo);
});

/**
 * @swagger
 * /todos/{id}:
 *   put:
 *     summary: Update the todo by the id
 *     tags: [Todos]
 *     parameters:
 *       - in: path
 *         name: id
 *         schema:
 *           type: integer
 *         required: true
 *         description: The todo id
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             $ref: '#/components/schemas/Todo'
 *     responses:
 *       200:
 *         description: The todo was updated
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/Todo'
 *       404:
 *         description: The todo was not found
 *       500:
 *         description: Some error happened
 */
app.put("/todos/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const updatedTodo = req.body;
  const todoIndex = todos.findIndex((t) => t.id === id);
  if (todoIndex === -1)
    return res.status(404).json({ error: "Todo not found" });
  todos[todoIndex] = updatedTodo;
  res.json(updatedTodo);
});

/**
 * @swagger
 * /todos/{id}:
 *   delete:
 *     summary: Remove the todo by id
 *     tags: [Todos]
 *     parameters:
 *       - in: path
 *         name: id
 *         schema:
 *           type: integer
 *         required: true
 *         description: The todo id
 *     responses:
 *       200:
 *         description: The todo was deleted
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/Todo'
 *       404:
 *         description: The todo was not found
 */
app.delete("/todos/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const todoIndex = todos.findIndex((t) => t.id === id);
  if (todoIndex === -1)
    return res.status(404).json({ error: "Todo not found" });
  const deletedTodo = todos.splice(todoIndex, 1)[0];
  res.json(deletedTodo);
});

app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

Step 4: Configure Swagger (swagger.js)

Create a separate file for Swagger configuration.

```
import swaggerJSDoc from 'swagger-jsdoc';

const options = {
  definition: {
    openapi: "3.0.0",
    info: {
      title: "Todo API",
      version: "1.0.0",
      description: "A simple Express Todo API"
    },
    servers: [
      {
        url: "http://localhost:3000",
      },
    ],
  },
  apis: ["./todo.js"], // Path to the API docs
};

const specs = swaggerJSDoc(options);

export default specs;
```

Step 5: Run the Application

Start your Express.js server:

```
node todo.js
```

- Navigate to http://localhost:3000/api-docs in your browser to see the Swagger UI and interact with your API endpoints.

With core http module we have to add if else checks to check for method type and route. Express provides simple methods to do the same. We are storing the todos in a local variable. All the routes are refering that same variable. Adding all the roues within the index file is not manageable. Also in an actual application we will have more resources, complex business logic and Database connection. Lets see how modularization makes the code better.

Modularization

Modularization involves breaking down your Express application into separate modules or files, each handling specific functionality. It offers several advantages, including improved code organization, maintainability, and scalability. Here's how you can modularize the above code:

Project Structure

```
my-express-app/
├── node_modules/
├── todosRouter.js
├── swagger.js
├── package.json
└── index.js
```

Step-by-Step Guide

1. Initialize the Project

Initialize your project and install the required dependencies:

```
npm init -y
npm install express swagger-jsdoc swagger-ui-express
```

2. Configure `package.json`

Ensure your `package.json` has the following structure to support ES modules:

```
{
  "name": "my-express-app",
  "version": "1.0.0",
  "type": "module",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.19.2",
    "swagger-jsdoc": "^6.2.8",
    "swagger-ui-express": "^5.0.0"
  }
}
```

3. Create the Entry Point (index.js)

Set up the Express server and configure Swagger in index.js.

```
import express from 'express';
import swaggerUi from 'swagger-ui-express';
import bodyParser from 'body-parser';
import swaggerSpecs from './swagger.js'; // Now using ES module syntax
import todosRouter from './todosRouter.js';

const app = express();
const PORT = 3000;

app.use(bodyParser.json());

// Swagger UI setup
app.use("/api-docs", swaggerUi.serve, swaggerUi.setup(swaggerSpecs));

app.use("/todos", todosRouter);

app.use((req, res) => {
  res.status(404).json({ error: "Not Found" });
});

app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

4. Define Todos Routes (todosRouter.js)

Create a separate file for todos routes.

```
import express from 'express';

const router = express.Router();

let todos = [
  { id: 1, task: "Buy groceries", completed: false },
  { id: 2, task: "Walk the dog", completed: true },
  // Add more todos here
];

/**
 * @swagger
 * components:
 *   schemas:
 *     Todo:
 *       type: object
 *       required:
 *         - task
 *         - completed
 *       properties:
 *         id:
 *           type: integer
 *           description: The auto-generated id of the todo item
 *         task:
 *           type: string
 *           description: The task description
 *         completed:
 *           type: boolean
 *           description: The status of the task
 *       example:
 *         id: 1
 *         task: Buy groceries
 *         completed: false
 */

/**
 * @swagger
 * tags:
 *   name: Todos
 *   description: The todos managing API
 */

/**
 * @swagger
 * /todos:
 *   get:
 *     summary: Returns the list of all the todos
 *     tags: [Todos]
 *     responses:
 *       200:
 *         description: The list of the todos
 *         content:
 *           application/json:
 *             schema:
 *               type: array
 *               items:
 *                 $ref: '#/components/schemas/Todo'
 */
router.get("/", (req, res) => {
  res.json(todos);
});

/**
 * @swagger
 * /todos/{id}:
 *   get:
 *     summary: Get the todo by id
 *     tags: [Todos]
 *     parameters:
 *       - in: path
 *         name: id
 *         schema:
 *           type: integer
 *         required: true
 *         description: The todo id
 *     responses:
 *       200:
```

```
 *         description: The todo description by id
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/Todo'
 *       404:
 *         description: The todo was not found
 */
router.get("/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const todo = todos.find((t) => t.id === id);
  if (!todo) return res.status(404).json({ error: "Todo not found" });
  res.json(todo);
});

/**
 * @swagger
 * /todos:
 *   post:
 *     summary: Create a new todo
 *     tags: [Todos]
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             $ref: '#/components/schemas/Todo'
 *     responses:
 *       201:
 *         description: The todo was successfully created
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/Todo'
 *       500:
 *         description: Some server error
 */
router.post("/", (req, res) => {
  const newTodo = req.body;
  newTodo.id = todos.length + 1;
  todos.push(newTodo);
  res.status(201).json(newTodo);
});

/**
 * @swagger
 * /todos/{id}:
 *   put:
 *     summary: Update the todo by the id
 *     tags: [Todos]
 *     parameters:
 *       - in: path
 *         name: id
 *         schema:
```

```
 *             type: integer
 *           required: true
 *           description: The todo id
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             $ref: '#/components/schemas/Todo'
 *     responses:
 *       200:
 *         description: The todo was updated
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/Todo'
 *       404:
 *         description: The todo was not found
 *       500:
 *         description: Some error happened
 */
router.put("/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const updatedTodo = req.body;
  const todoIndex = todos.findIndex((t) => t.id === id);
  if (todoIndex === -1)
    return res.status(404).json({ error: "Todo not found" });
  todos[todoIndex] = updatedTodo;
  res.json(updatedTodo);
});

/**
 * @swagger
 * /todos/{id}:
 *   delete:
 *     summary: Remove the todo by id
 *     tags: [Todos]
 *     parameters:
 *       - in: path
 *         name: id
 *         schema:
 *           type: integer
 *         required: true
 *         description: The todo id
 *     responses:
 *       200:
 *         description: The todo was deleted
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/Todo'
 *       404:
 *         description: The todo was not found
 */
```

```
router.delete("/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const todoIndex = todos.findIndex((t) => t.id === id);
  if (todoIndex === -1)
    return res.status(404).json({ error: "Todo not found" });
  const deletedTodo = todos.splice(todoIndex, 1)[0];
  res.json(deletedTodo);
});

export default router;
```

5. Configure Swagger (swagger.js)

Create a separate file for Swagger configuration.

```
import swaggerJSDoc from 'swagger-jsdoc';

const options = {
  definition: {
    openapi: "3.0.0",
    info: {
      title: "Todo API",
      version: "1.0.0",
      description: "A simple Express Todo API"
    },
    servers: [
      {
        url: "http://localhost:3000",
      },
    ],
  },
  apis: ["./todosRouter.js"], // Path to the API docs
};

const specs = swaggerJSDoc(options);

export default specs;
```

6. Run the Application

Start your Express.js server:

```
node index.js
```

JSON text to a JSON object

To convert the JSON text type to JSON object type in your POST request handler for the todos endpoint, you need to ensure that the data you are receiving is parsed correctly. This is typically done using body-parser middleware in Express, which you've already included in your setup. However, if the JSON data is being sent as a string, you may need to explicitly parse it within your handler.

To incorporate the specific handling of JSON text (a JSON string) into your project structure and edit the POST method accordingly, follow these instructions:

Project Structure

```
my-express-app/
├── node_modules/
├── todosRouter.js
├── swagger.js
├── package.json
└── index.js
```

Here is the Updated post Method

```
router.post('/', (req, res) => {
  let newTodo = req.body;
  console.log("Before Parse : ",newTodo);

  // If the body is a JSON string, parse it
  if (typeof newTodo === 'string') {
    try {
      newTodo = JSON.parse(newTodo);
    } catch (error) {
      return res.status(400).json({ error: 'Invalid JSON' });
    }
  }

  newTodo.id = todos.length + 1;
  console.log("After Parse : ",newTodo);
  todos.push(newTodo);
  res.status(201).json(newTodo);
});
```

Here's the updated `todosRouter.j`s for the POST method:

```
import express from 'express';

const router = express.Router();

let todos = [
  { id: 1, task: "Buy groceries", completed: false },
  { id: 2, task: "Walk the dog", completed: true },
  // Add more todos here
];
```

```
/**
 * @swagger
 * components:
 *   schemas:
 *     Todo:
 *       type: object
 *       required:
 *         - task
 *         - completed
 *       properties:
 *         id:
 *           type: integer
 *           description: The auto-generated id of the todo item
 *         task:
 *           type: string
 *           description: The task description
 *         completed:
 *           type: boolean
 *           description: The status of the task
 *       example:
 *         id: 1
 *         task: Buy groceries
 *         completed: false
 */

/**
 * @swagger
 * tags:
 *   name: Todos
 *   description: The todos managing API
 */

/**
 * @swagger
 * /todos:
 *   get:
 *     summary: Returns the list of all the todos
 *     tags: [Todos]
 *     responses:
 *       200:
 *         description: The list of the todos
 *         content:
 *           application/json:
 *             schema:
 *               type: array
 *               items:
 *                 $ref: '#/components/schemas/Todo'
 */
router.get("/", (req, res) => {
  res.json(todos);
});

/**
 * @swagger
 * /todos/{id}:
 *   get:
 *     summary: Get the todo by id
 *     tags: [Todos]
 *     parameters:
 *       - in: path
 *         name: id
 *         schema:
 *           type: integer
 *         required: true
 *         description: The todo id
 *     responses:
 *       200:
 *         description: The todo description by id
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/Todo'
 *       404:
 *         description: The todo was not found
 */
router.get("/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const todo = todos.find((t) => t.id === id);
  if (!todo) return res.status(404).json({ error: "Todo not found" });
  res.json(todo);
});

/**
 * @swagger
 * /todos:
 *   post:
 *     summary: Create a new todo
 *     tags: [Todos]
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             $ref: '#/components/schemas/Todo'
 *     responses:
 *       201:
 *         description: The todo was successfully created
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/Todo'
 *       500:
 *         description: Some server error
 */
router.post("/", (req, res) => {
  const newTodo = req.body;
  newTodo.id = todos.length + 1;
  todos.push(newTodo);
  res.status(201).json(newTodo);
});

/**
 * @swagger
 * /todos/{id}:
 *   put:
 *     summary: Update the todo by the id
 *     tags: [Todos]
 *     parameters:
 *       - in: path
 *         name: id
 *         schema:
 *           type: integer
 *         required: true
 *         description: The todo id
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             $ref: '#/components/schemas/Todo'
 *     responses:
 *       200:
 *         description: The todo was updated
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/Todo'
 *       404:
 *         description: The todo was not found
 *       500:
 *         description: Some error happened
 */
router.put("/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const updatedTodo = req.body;
  const todoIndex = todos.findIndex((t) => t.id === id);
  if (todoIndex === -1)
    return res.status(404).json({ error: "Todo not found" });
  todos[todoIndex] = updatedTodo;
  res.json(updatedTodo);
});

/**
 * @swagger
 * /todos/{id}:
 *   delete:
 *     summary: Remove the todo by id
 *     tags: [Todos]
 *     parameters:
 *       - in: path
 *         name: id
 *         schema:
```

```
 *             type: integer
 *           required: true
 *           description: The todo id
 *     responses:
 *       200:
 *         description: The todo was deleted
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/Todo'
 *       404:
 *         description: The todo was not found
 */
router.delete("/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const todoIndex = todos.findIndex((t) => t.id === id);
  if (todoIndex === -1)
    return res.status(404).json({ error: "Todo not found" });
  const deletedTodo = todos.splice(todoIndex, 1)[0];
  res.json(deletedTodo);
});

export default router;
```

Content Type Negotiation

To support content type negotiation with JSON and XML in your Express.js application, you need to handle different content types for both incoming requests and outgoing responses. Here's how you can modify your code to achieve this:

To incorporate the content type negotiation into your project structure and edit the POST method accordingly and Middlewares to parse XML Body and Helper function to send response based on Accept header, follow these instructions:

Project Structure

```
my-express-app/
├── node_modules/
├── todosRouter.js
├── swagger.js
├── package.json
└── index.js
```

Install the xml-js package: This package will help in converting XML to JSON and vice versa.

```
npm install xml-js
```

Update Middlewares to parse XML Body and Helper function to send response based on Accept header in your `todosRouter1.js`

```
// Middleware to parse XML body
router.use((req, res, next) => {
    if (req.is('xml')) {
      let data = '';
      req.setEncoding('utf8');
      req.on('data', chunk => { data += chunk });
      req.on('end', () => {
        req.body = xml2js(data, { compact: true, ignoreComment: true,
alwaysChildren: true });
        next();
      });
    } else {
      next();
    }
  });

  // Helper function to send response based on Accept header
  const sendResponse = (req, res, data) => {
    const accept = req.headers.accept || '';
    if (accept.includes('xml')) {
      const xml = js2xml(data, { compact: true, ignoreComment: true, spaces: 4 });
      res.type('application/xml').send(xml);
    } else {
      res.json(data);
    }
  };
```

Here is an Updated Post Method

```
router.post('/', (req, res) => {
    let newTodo = req.body;

    if (req.is('xml')) {
      newTodo = {
        task: newTodo.todo.task._text,
        completed: newTodo.todo.completed._text === 'true'
      };
    } else if (typeof newTodo === 'string') {
      try {
        newTodo = JSON.parse(newTodo);
      } catch (error) {
        return res.status(400).json({ error: 'Invalid JSON' });
      }
    }

    newTodo.id = todos.length + 1;
    todos.push(newTodo);
    res.status(201);
    sendResponse(req, res, { newTodo });
  });
```

Here's the updated `todosRouter.js` for the POST method:

```
import express from 'express';

const router = express.Router();

let todos = [
  { id: 1, task: "Buy groceries", completed: false },
  { id: 2, task: "Walk the dog", completed: true },
  // Add more todos here
];

/**
 * @swagger
 * components:
 *   schemas:
 *     Todo:
 *       type: object
 *       required:
 *         - task
 *         - completed
 *       properties:
 *         id:
 *           type: integer
 *           description: The auto-generated id of the todo item
 *         task:
 *           type: string
 *           description: The task description
 *         completed:
 *           type: boolean
 *           description: The status of the task
 *       example:
 *         id: 1
 *         task: Buy groceries
 *         completed: false
 */

/**
 * @swagger
 * tags:
 *   name: Todos
 *   description: The todos managing API
 */

/**
 * @swagger
 * /todos:
 *   get:
```

```
 *     summary: Returns the list of all the todos
 *     tags: [Todos]
 *     responses:
 *       200:
 *         description: The list of the todos
 *         content:
 *           application/json:
 *             schema:
 *               type: array
 *               items:
 *                 $ref: '#/components/schemas/Todo'
 */
router.get("/", (req, res) => {
  res.json(todos);
});

/**
 * @swagger
 * /todos/{id}:
 *   get:
 *     summary: Get the todo by id
 *     tags: [Todos]
 *     parameters:
 *       - in: path
 *         name: id
 *         schema:
 *           type: integer
 *         required: true
 *         description: The todo id
 *     responses:
 *       200:
 *         description: The todo description by id
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/Todo'
 *       404:
 *         description: The todo was not found
 */
router.get("/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const todo = todos.find((t) => t.id === id);
  if (!todo) return res.status(404).json({ error: "Todo not found" });
  res.json(todo);
});

/**
 * @swagger
 * /todos:
 *   post:
 *     summary: Create a new todo
 *     tags: [Todos]
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             $ref: '#/components/schemas/Todo'
 *     responses:
 *       201:
 *         description: The todo was successfully created
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/Todo'
 *       500:
 *         description: Some server error
 */
router.post("/", (req, res) => {
  const newTodo = req.body;
  newTodo.id = todos.length + 1;
  todos.push(newTodo);
  res.status(201).json(newTodo);
});

/**
 * @swagger
 * /todos/{id}:
 *   put:
 *     summary: Update the todo by the id
 *     tags: [Todos]
 *     parameters:
 *       - in: path
 *         name: id
 *         schema:
 *           type: integer
 *         required: true
 *         description: The todo id
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             $ref: '#/components/schemas/Todo'
 *     responses:
 *       200:
 *         description: The todo was updated
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/Todo'
 *       404:
 *         description: The todo was not found
 *       500:
 *         description: Some error happened
 */
router.put("/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const updatedTodo = req.body;
  const todoIndex = todos.findIndex((t) => t.id === id);
  if (todoIndex === -1)
    return res.status(404).json({ error: "Todo not found" });
  todos[todoIndex] = updatedTodo;
  res.json(updatedTodo);
});

/**
 * @swagger
 * /todos/{id}:
 *   delete:
 *     summary: Remove the todo by id
 *     tags: [Todos]
 *     parameters:
 *       - in: path
 *         name: id
 *         schema:
 *           type: integer
 *         required: true
 *         description: The todo id
 *     responses:
 *       200:
 *         description: The todo was deleted
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/Todo'
 *       404:
 *         description: The todo was not found
 */
router.delete("/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const todoIndex = todos.findIndex((t) => t.id === id);
  if (todoIndex === -1)
    return res.status(404).json({ error: "Todo not found" });
  const deletedTodo = todos.splice(todoIndex, 1)[0];
  res.json(deletedTodo);
});

export default router;
```

Consuming the REST API

1. Initialize a Node.js project:

```
mkdir my-express-client
cd my-express-client
npm init -y
```

Creates a folder by name `my-express-client`. `npm init -y` creates `package.json` file and instantiates a project.

2. Install Express:

```
npm install express
```

3. Project Structure

```
my-express-client/
├── public/
│   └── index.html
├── node_modules/
├── app.js
└── package.json
```

4. Create an `app.js` file.

```
const express = require('express');
const app = express();

// Serve static files from the 'public' directory
app.use(express.static('public'));

// Start the server
const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

4. Create a public folder and create `index.html` which consumes todo Rest API.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Todo API Client</title>
  <style>
    table {
      width: 100%;
      border-collapse: collapse;
    }
    table, th, td {
      border: 1px solid black;
    }
```

```
      th, td {
        padding: 8px;
        text-align: left;
      }
      th {
        background-color: #f2f2f2;
      }
      form {
        margin-bottom: 20px;
      }
    </style>
  </head>
  <body>
    <h1>Todo List</h1>
    <table id="todo-table">
      <thead>
        <tr>
          <th>ID</th>
          <th>Task</th>
          <th>Completed</th>
          <th>Actions</th>
        </tr>
      </thead>
      <tbody>
        <!-- Todos will be dynamically populated here -->
      </tbody>
    </table>

    <h2>Add/Edit Todo</h2>
    <form id="add-todo-form">
      <label for="todo-id">Todo ID:</label>
      <select id="todo-id">
        <option value="">Select Todo ID</option>
        <!-- Options will be populated dynamically -->
      </select>
      <br>
      <label for="task">Task:</label>
      <input type="text" id="task" placeholder="Task" required>
      <br>
      <label for="completed">Completed:</label>
      <input type="checkbox" id="completed">
      <br>
      <label for="content-type">Content Type:</label>
      <select id="content-type">
        <option value="application/json">JSON</option>
        <option value="application/xml">XML</option>
      </select>
      <br>
      <button type="submit">Save Todo</button>
    </form>

    <script src="https://cdn.jsdelivr.net/npm/xml-js@1.6.11/dist/xml-js.min.js">
  </script>
    <script>
      const API_URL = 'http://192.168.42.12:3000/todos';

      // Fetch and display all todos
      async function fetchTodos() {
        try {
          const response = await fetch(API_URL);
          const todos = await response.json();
          const todoTableBody = document.querySelector('#todo-table tbody');
          const todoIdSelect = document.getElementById('todo-id');

          // Clear current contents
          todoTableBody.innerHTML = '';
          todoIdSelect.innerHTML = '<option value="">Select Todo ID</option>';

          todos.forEach(todo => {
            // Populate table
            const row = document.createElement('tr');
            row.innerHTML = `
              <td>${todo.id}</td>
              <td>${todo.task}</td>
              <td>${todo.completed ? 'Completed' : 'Not Completed'}</td>
              <td>
                <button onclick="populateForm(${todo.id})">Edit</button>
                <button onclick="deleteTodo(${todo.id})">Delete</button>
              </td>
            `;
            todoTableBody.appendChild(row);

            // Populate dropdown
            const option = document.createElement('option');
            option.value = todo.id;
            option.textContent = todo.id;
            todoIdSelect.appendChild(option);
          });
        } catch (error) {
          console.error('Error fetching todos:', error);
        }
      }

      function getSelectedContentType() {
        return document.getElementById('content-type').value;
      }

      // Add a new todo
      async function addTodo() {
        const taskInput = document.getElementById('task');
        const completedInput = document.getElementById('completed');
        const contentType = getSelectedContentType();

        const newTodo = {
          task: taskInput.value,
          completed: completedInput.checked
        };
```

```
        let body;
        let headers = { 'Content-Type': contentType };

        if (contentType === 'application/xml') {
          body = js2xml({ todo: { task: { _text: newTodo.task }, completed: { _text:
newTodo.completed.toString() } } }, { compact: true });
          console.log(body);
        } else {
          body = JSON.stringify(newTodo);
        }

        try {
          const response = await fetch(API_URL, {
            method: 'POST',
            headers: headers,
            body: body
          });

          if (response.ok) {
            taskInput.value = '';
            completedInput.checked = false;
            fetchTodos(); // Refresh the todo list
          } else {
            console.error('Error adding todo:', response.statusText);
          }
        } catch (error) {
          console.error('Error adding todo:', error);
        }
      }

      // Populate form with selected todo
      async function populateForm(id) {
        try {
          const response = await fetch(`${API_URL}/${id}`);
          const todo = await response.json();
          if (response.ok) {
            const idSelect = document.getElementById('todo-id');
            const taskInput = document.getElementById('task');
            const completedInput = document.getElementById('completed');

            idSelect.value = todo.id;
            taskInput.value = todo.task;
            completedInput.checked = todo.completed;

            document.getElementById('add-todo-form').onsubmit = function(event) {
              event.preventDefault();
              updateTodo();
            };
          } else {
            console.error('Error fetching todo:', response.statusText);
          }
        } catch (error) {
          console.error('Error fetching todo:', error);
        }
```

```
    }

    // Update a todo
    async function updateTodo() {
      const idSelect = document.getElementById('todo-id');
      const taskInput = document.getElementById('task');
      const completedInput = document.getElementById('completed');

      const updatedTodo = {
        id: parseInt(idSelect.value),
        task: taskInput.value,
        completed: completedInput.checked
      };

      try {
        const response = await fetch(`${API_URL}/${updatedTodo.id}`, {
          method: 'PUT',
          headers: {
            'Content-Type': 'application/json'
          },
          body: JSON.stringify(updatedTodo)
        });

        if (response.ok) {
          idSelect.value = '';
          taskInput.value = '';
          completedInput.checked = false;
          document.getElementById('add-todo-form').onsubmit = addTodo; // Reset
form submission to addTodo
          fetchTodos(); // Refresh the todo list
        } else {
          console.error('Error updating todo:', response.statusText);
        }
      } catch (error) {
        console.error('Error updating todo:', error);
      }
    }

    // Delete a todo
    async function deleteTodo(id) {
      try {
        const response = await fetch(`${API_URL}/${id}`, {
          method: 'DELETE'
        });

        if (response.ok) {
          fetchTodos(); // Refresh the todo list
        } else {
          console.error('Error deleting todo:', response.statusText);
        }
      } catch (error) {
        console.error('Error deleting todo:', error);
      }
    }
    document.getElementById('add-todo-form').addEventListener('submit',
function(event) {
      event.preventDefault();
      if (document.getElementById('todo-id').value) {
        updateTodo();
      } else {
        addTodo();
      }
    });

    // Initial fetch
    fetchTodos();
  </script>
</body>
</html>
```

- The page displays a table with columns for ID, Task, Completed status, and Actions (Edit and Delete buttons).
- Initially, this table is empty, but it gets populated with todos fetched from the REST API when the page loads.
- Users can choose between JSON and XML formats for sending data to the server by selecting the desired option from the Content Type dropdown.
- Depending on the selected format, the data submitted in the form will be sent to the server as either JSON or XML.

5. Ensure your index.html file is placed in the public directory. Then, run your application using the command:

```
node app.js
```

Enabling CORS

Enabling CORS in Your Express Application

To enable CORS (Cross-Origin Resource Sharing) in your Express application, follow these steps:

1. Install the CORS Package

First, you need to install the cors package, which will allow you to enable CORS in your Express application:

Install CORS package:

```
npm install cors
```

2. Project Structure

```
my-express-app/
├── node_modules/
├── corsOptions.js
├── todosRouter.js
├── swagger.js
├── package.json
└── index.js
```

3. Configure CORS Options

Create a new file named `corsOptions.js` to configure your CORS options. This file will specify allowed origins, methods, headers, etc.

```
// corsOptions.js
const corsOptions = {
    origin: 'http://localhost:5000',
    methods: 'GET,POST',
    allowedHeaders: 'Content-Type,Authorization',
};

export default corsOptions;
```

- `origin`: Specifies which origins are allowed to access the resources. You can set it to a specific origin (e.g., '`http://localhost:5000`') or use a wildcard ('`*`') to allow access from any origin.
- `methods`: Defines which HTTP methods are allowed for cross-origin requests. Common methods include `GET, POST, PUT, DELETE`, etc. You can specify multiple methods separated by commas.
- `allowedHeaders`: Specifies which headers can be included in cross-origin requests. This is useful for controlling which headers are allowed to be sent with the request.
- `credentials`: Indicates whether credentials such as cookies or HTTP authentication should be included in cross-origin requests. By default, browsers do not include credentials in CORS requests. If you set this option to true, you must also configure the server to allow credentials.
- `exposedHeaders`: Lists the headers that the server allows to be accessed by the client in response to the CORS request.
- `maxAge`: Specifies how long the results of a preflight request can be cached by the client, in seconds.

4. Update Your Main Application File (index.js)

Modify your index.js file to enable CORS using the configured options:

```
import express from 'express';
import cors from 'cors';
import swaggerUi from 'swagger-ui-express';
import bodyParser from 'body-parser';
import swaggerSpecs from './swagger.js'; // Now using ES module syntax
import todosRouter from './todosRouter1.js';
import corsOptions from './corsOptions.js';

const app = express();
const PORT = 3000;

// Enable CORS
app.use(cors(corsOptions));

app.use(bodyParser.json());

// Swagger UI setup
app.use("/api-docs", swaggerUi.serve, swaggerUi.setup(swaggerSpecs));

app.use("/todos", todosRouter);

app.use((req, res) => {
  res.status(404).json({ error: "Not Found" });
});

app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

5. Running the Application

Run the server using the following command:

```
node index.js
```

Navigate to `http://localhost:3000/api-docs` in your browser to see the Swagger UI and interact with your API endpoints.

Middlewares

Middleware plays a crucial role in Express.js, allowing you to add functionality to the request-response pipeline. Middleware is a series of functions that are executed in the order they are defined when an HTTP request is made to an Express.js application. These functions have access to the request (req) and response (res) objects and can perform various tasks, such as logging, data validation, authentication, and more.

Types of Middleware:

1. `Application-level Middleware:` These are middleware functions that are applied to the entire application. They are defined using app.use() and are executed for every incoming request.
2. `Router-level Middleware:` These are middleware functions applied to a specific router using router.use(). They only affect routes defined within that router.

3. `Error-handling Middleware:` These middleware functions handle errors. They have four parameters (err, req, res, next) and are defined with an extra parameter.

Built-in Middlewares in Express.js:

Express.js provides several built-in middleware functions to handle common tasks. Here are some of them:

1. `express.json():` Parses incoming JSON data from the request body.

```
import express, { json } from "express";
const app = express();

app.use(json());

app.post("/api/post", (req, res) => {
  console.log(req.body); // Access the JSON data
  res.json({ message: "Data received successfully" });
});

app.listen(3000, () => {
  console.log("Server is running on port 3000");
});
```

2. `express.urlencoded():` Parses incoming URL-encoded data (e.g., form submissions) from the request body.

```
import express, { urlencoded } from "express";
const app = express();

app.use(urlencoded({ extended: true }));

app.post("/api/form", (req, res) => {
  console.log(req.body); // Access URL-encoded data
  res.json({ message: "Form data received successfully" });
});

app.listen(3000, () => {
  console.log("Server is running on port 3000");
});
```

3. `express.static():` Serves static files (e.g., HTML, CSS, JavaScript, images) from a specified directory.

```
import express from "express";

const app = express();

app.use(express.static("public")); // Serve files from the 'public' directory
app.listen(3000, () => {
  console.log("Server is running on port 3000");
});
```

4. `Custom Middleware:` You can create custom middleware functions to perform specific tasks. Here's an example of a custom middleware that logs the request method and URL

```
import express from "express";
const app = express();

// // Custom middleware
app.use((req, res, next) => {
  console.log(`Request Method: ${req.method}, URL: ${req.url}`);
  next(); // Call the next middleware
});

app.get("/", (req, res) => {
  res.send("Home Page");
});

app.get("/about", (req, res) => {
  res.send("About Page");
});

app.listen(3000, () => {
  console.log("Server is running on port 3000");
});
```

In this example, the custom middleware logs information about each incoming request before passing control to the next middleware or route handler.

Serving static content

Static content are nothing but files which are served by server. For example html, css, js, images etc. Express has a middleware to do this. `express.static()` allows us to server all the content we want from a configured folder.

Step-by-Step Guide

1. Create Project Directory:

Create a new directory for your project and navigate into it:

```
mkdir static-content-server
cd static-content-server
```

2. Initialize npm:

Initialize a new Node.js project:

```
npm init -y
```

3. Install Express:

Install Express as a dependency:

```
npm install express
```

4. Create Directory Structure:

Create the necessary directories and files:

```
mkdir public
mkdir public/css
mkdir public/js
touch public/index.html
touch public/css/styles.css
touch public/js/script.js
touch server.js
```

5. Create HTML, CSS, and JavaScript Files:

public/index.html:

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Hello World Page</title>
    <link rel="stylesheet" href="/css/styles.css" />
    <script src="/js/script.js"></script>
  </head>
  <body>
    <h1 onclick="showAlert()">Hello World</h1>
  </body>
</html>
```

public/css/styles.css:

```css
body {
  margin: 0;
  padding: 0;
  display: flex;
  justify-content: center;
  align-items: center;
  min-height: 100vh;
  background-color: lightblue;
}

h1 {
  cursor: pointer;
}

h1:hover {
  text-decoration: underline;
}
```

public/js/script.js:

```
function showAlert() {
  alert("Hello World");
}
```

6. Create Express Server:

server.js:

```
const express = require("express");
const path = require("path");

const app = express();
const PORT = 3000;

app.use(express.static(path.join(__dirname, "public"))); // Configure public
folder as servable

// Define a route for the homepage
app.get("/", (req, res) => {
  res.send("Welcome to my static content app!");
});

app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

7. Run the Server:

Start your server by running the following command in your project directory:

```
node server.js
```

We pass the complete path of the `public` folder to the `express.static()` middleware. public folder has `index.html` file at its root. Now any request made to the server at `localhost:3000` will serve `index.html` file. Express takes care of the underlying logic for us. Within the html we import the css and js files. Both css and js files should be placed inside public folder for express to serve them. If not express will block the download of those files which browser attempts to.

When the `index.html` file is loaded by the browser. Browser also downloads all the css and js files linked within the html file. We can see the styles and event listeners added to the html element by css and js.

Project NPM packages

dotenv

`Installation: npm install dotenv`

`Purpose:` Injects environment variables into application code. Enables us to read configrations from `.env` file. Based on the environment(dev, testing or production) we are in we can load environment specific cofigs like port number, database connection url or db credentials.

`Usage:` import the package using `require("dotenv/config")`. Dont have to initialise a variable to store the module. Just requiring is enough. Place all out environment variables into a `.env` file in the root folder of out project.

```
PORT=8080

JWT_SECRET = nodejstraining
USER_TOKEN_EXPIRY=3600
ADMIN_EMAIL=admin@nodejs.com

DATABASE_URL=mongodb://0.0.0.0:27017/todoapp
```

This is just a text file. No need to wrap strings like db connection url into quotes. Now we read the value within the application code using `process.env.PORT`. process is exposed by node.js. We can access all the environment variables from .env file using this.

bcrypt

`Installation: npm install bcrypt`.

`Purpose:` bcrypt helps in hashing. We should not store user passwords into database as plane strings. Its best practise to always hash the password and save. Even developers wont be able to see what the password is. We hash the password before saving to the db. We hash the password sent by the user while loggin in and compare the already hashed password saved in the db to check if the login credentials are right.

`Usage:`

```
const bcrypt = require("bcrypt");
const salt = await bcrypt.genSalt(10); // greater the number greater security
const hashedPassword = await bcrypt.hash("password", salt);
```

Here `password` is the string we are hashing. We generate the salt and use that to hash the string `password`.

jsonwebtoken

`Installation: npm install jsonwebtoken`.

`Purpose:` Used to generate, verify and decode jwt tokens. Jwt tokens helps us to build secure systems.

`Usage:`

Creating the token:

```
const token = jwt.sign(
  {
    id: existingUser.id,
    email: existingUser.email,
    role: existingUser.role,
  },
  "Jwt secret",
  { expiresIn: parseInt(3600) }
);
```

as 1st argument, we pass what we want to embed into the token, 2nd argument is the secret used to generate the token. This secret is requird to decode the token aswell.3rd agrument can be options. For example duration of validity of the token in seconds.

cors

`Installation: npm install cors`

`Purpose:` It is used to enable or restrict cross-origin HTTP requests in web applications. When you make an HTTP request from one domain (origin) to another domain, the browser's same-origin policy typically blocks the request for security reasons. The "cors" middleware helps you control and configure how your server responds to such requests from different origins.

`Usage:` just pass the cors as middleware to express application.

```
const express = require("express");
const cors = require("cors");

const app = express();

// Enable CORS for all routes
app.use(cors());
```

```
// Your routes and other middleware

app.listen(3000, () => {
  console.log("Server is running on port 3000");
});
```

By adding app.use(cors()) to your Express application, you enable CORS for all routes. You can customize CORS behavior by passing options to the cors function, such as specifying allowed origins and HTTP methods.

JS HTML DOM

DOM Intro

The Document Object Model (DOM) in JavaScript is a programming interface that represents the structure of an HTML or XML document as a tree-like structure, where each node represents a part of the document, such as elements, attributes, and text. This model allows JavaScript to interact with and manipulate the content, structure, and style of a webpage.

Here are some key concepts related to the JavaScript DOM:

1. **Document Object**: The top-level object representing the entire HTML document. It provides methods and properties to access and manipulate the document's content.

2. **Elements**: HTML elements are represented as nodes in the DOM tree. You can access elements using methods like `document.getElementById()`, `document.querySelector()`, or by navigating the DOM tree using properties like `parentNode`, `childNodes`, `firstChild`, and `lastChild`.

3. **Attributes**: Elements may have attributes like `id`, `class`, `src`, `href`, etc. You can access and modify element attributes using methods like `getAttribute()`, `setAttribute()`, and properties like `element.id`, `element.className`, etc.

4. **Events**: DOM events are actions or occurrences that happen in the DOM, such as a user clicking a button or the browser finishing loading a document. You can attach event listeners to elements using methods like `addEventListener()` to respond to these events.

5. **Manipulating Content**: JavaScript can dynamically modify the content of a webpage by creating, removing, or modifying elements and their attributes. For example, you can create new elements using `document.createElement()`, add them to the DOM using methods like `appendChild()` or `insertBefore()`, and remove elements using `removeChild()`.

6. **Styling**: You can also manipulate the style of elements using the `style` property or by adding/removing CSS classes with the `classList` property.

DOM manipulation is a fundamental part of web development, allowing you to create interactive and dynamic web experiences. If you have specific questions or need more details on any aspect of the DOM, feel free to ask!

Document Object

The Document Object in JavaScript represents the entire HTML document. It's the top-level object in the DOM hierarchy and provides access to various properties and methods for interacting with the document's structure and content. Here are some key points about the Document Object:

1.Accessing the Document Object

You can access the Document Object using the global `document` variable in JavaScript. For example:

```
console.log(document); // Outputs the Document Object
```

2.Properties of the Document Object:

- `document.documentElement`: Represents the root element of the document (usually `<html>`).
- `document.body`: Represents the `<body>` element of the document.
- `document.head`: Represents the `<head>` element of the document.
- `document.title`: Gets or sets the title of the document.
- `document.URL`: Gets the full URL of the document.
- `document.domain`: Gets or sets the domain of the document.

3.Methods of the Document Object:

- `document.getElementById(id)`: Returns the element with the specified ID.
- `document.querySelector(selector)`: Returns the first element that matches the specified CSS selector.
- `document.createElement(tagName)`: Creates a new element with the specified tag name.
- `document.createTextNode(text)`: Creates a new text node with the specified text.
- `document.querySelectorAll(selector)`: Returns a NodeList of elements that match the specified CSS selector.
- `document.createElementNS(namespaceURI, qualifiedName)`: Creates a new element with the specified namespace URI and qualified name.

4.Working with Document Content:

- Adding elements to the document:

```
const newElement = document.createElement('div');
newElement.textContent = 'Hello, World!';
document.body.appendChild(newElement);
```

- Modifying element attributes:

```
const myElement = document.getElementById('myElement');
myElement.setAttribute('class', 'newClass');
```

- Manipulating styles:

```
const myElement = document.getElementById('myElement');
myElement.style.color = 'red';
```

5.Events:

- `DOMContentLoaded`: Fires when the initial HTML document has been completely loaded and parsed.
- `load`: Fires when the entire page (including images and other resources) has finished loading.
- `click`, `mouseover`, `submit`, etc.: Various events that can be attached to elements in the document.

The Document Object provides a powerful API for working with HTML documents in JavaScript, allowing you to create, modify, and interact with the content dynamically.

DOM Model

The Document Object Model (DOM) is a programming interface for web documents. It represents the structure of HTML and XML documents as a tree-like model, where each node in the tree corresponds to a part of the document, such as elements, attributes, and text.

Here's a breakdown of the DOM model:

1. **Node Types**:
    - **Element Node**: Represents an HTML element, like `<div>`, `<p>`, `<a>`, etc.
    - **Text Node**: Represents the text within an element, such as "Hello, World!" inside a `<p>` tag.
    - **Attribute Node**: Represents an attribute of an element, like `id="myElement"`.
    - **Comment Node**: Represents a comment within the HTML code, .
    - **Document Node**: Represents the entire HTML document.
    - **Document Type Node**: Represents the document type declaration, like `<!DOCTYPE html>`.
2. **Hierarchy**:
    - The DOM tree starts with the Document Node at the top.
    - The Document Node has child nodes representing the `<html>` element, which in turn has child nodes like `<head>` and `<body>`.

  - Elements can have child nodes, which can be other elements, text nodes, or even comment nodes.
  - Nodes can have sibling nodes, which are nodes that share the same parent node.

3. **Accessing Nodes**:

   - **Traversal**: You can navigate the DOM tree using properties like `parentNode`, `childNodes`, `firstChild`, `lastChild`, `nextSibling`, and `previousSibling`.
   - **Selectors**: Methods like `getElementById()`, `querySelector()`, `getElementsByClassName()`, and `getElementsByTagName()` allow you to select specific elements based on IDs, classes, tags, or CSS selectors.
   - **Creating Nodes**: You can create new elements and nodes using `document.createElement()`, `document.createTextNode()`, and other methods.

4. **Manipulating Nodes**:

   - **Adding Nodes**: Use methods like `appendChild()`, `insertBefore()`, and `insertAdjacentHTML()` to add new nodes to the DOM.
   - **Modifying Nodes**: Change element attributes using `setAttribute()`, update text content with `textContent` or `innerHTML`, and modify styles with the `style` property.
   - **Removing Nodes**: Remove nodes using methods like `removeChild()` or `parentNode.removeChild()`.

5. **Events**:

   - DOM events allow JavaScript to respond to user actions and document changes.
   - Common events include `click`, `mouseover`, `keydown`, `submit`, `load`, `DOMContentLoaded`, etc.
   - You can attach event listeners using methods like `addEventListener()`.

Understanding the DOM model is crucial for web development as it enables dynamic interaction with HTML documents, making it possible to create interactive and responsive web applications.

Here's a simple example demonstrating the DOM model in action. This example creates a new paragraph element, sets its text content, and appends it to the body of the HTML document:

HTML:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>DOM Model Example</title>
</head>
<body>
    <h1>DOM Model Example</h1>
    <div id="content">
        <!-- New paragraph will be added here -->
    </div>
    <script src="script.js"></script>
```

```
</body>
</html>
```

JavaScript (script.js):

```
// Create a new paragraph element
const paragraph = document.createElement('p');

// Set the text content of the paragraph
paragraph.textContent = 'This is a new paragraph created using the DOM model.';

// Get the content div where we want to append the paragraph
const contentDiv = document.getElementById('content');

// Append the paragraph to the content div
contentDiv.appendChild(paragraph);
```

In this example:

1. We start with an HTML document that includes a `<div>` element with the ID "content," where we'll add the new paragraph.
2. The JavaScript code creates a new `<p>` element using `document.createElement('p')`.
3. We set the text content of the paragraph using `paragraph.textContent`.
4. The `getElementById()` method is used to get the content `<div>` where we want to append the paragraph.
5. Finally, we append the paragraph to the content `<div>` using `contentDiv.appendChild(paragraph)`.

When you open this HTML file in a browser and inspect the DOM using the browser's developer tools, you'll see the newly created paragraph element added inside the "content" `<div>`.

DOM other Object

Apart from the Document Object, there are several other important objects and concepts in the DOM that are commonly used in JavaScript:

1. **Element Object**:
    - Represents an HTML element in the DOM tree.
    - Provides properties and methods to manipulate the attributes, styles, and content of the element.
2. **Node Object**:
    - Represents a node in the DOM tree.
    - Both Element and non-Element nodes (such as Text nodes, Comment nodes, etc.) are instances of the Node object.
    - Provides common properties and methods for all types of nodes, such as `parentNode`, `childNodes`, `firstChild`, `lastChild`, `nextSibling`, `previousSibling`, etc.

3. **Event Object**:
   - Represents an event that occurs in the DOM, such as a mouse click, keypress, form submission, etc.
   - Contains information about the event, such as the target element, event type, mouse coordinates, key codes, etc.
   - Passed as an argument to event handler functions when an event is triggered.
4. **Window Object**:
   - Represents the browser window that contains the DOM document.
   - Provides properties and methods for interacting with the browser window, such as `window.location`, `window.history`, `window.alert()`, `window.setTimeout()`, etc.
   - Acts as the global object in client-side JavaScript.
5. **DocumentFragment Object**:
   - Represents a lightweight Document object that can hold a collection of nodes.
   - Useful for creating and manipulating a group of nodes before appending them to the main document, improving performance in DOM manipulation.
6. **HTMLCollection and NodeList Objects**:
   - Both represent collections of nodes (usually elements) in the DOM.
   - HTMLCollection is live, meaning it automatically updates as the DOM changes, such as when elements are added or removed.
   - NodeList is static and represents a snapshot of the DOM at the time it was queried.
7. **Location Object**:
   - Represents the current URL of the document and provides properties like `href`, `hostname`, `pathname`, `search`, etc.
   - Accessible via `window.location` or `document.location`.

These objects and concepts work together to provide a powerful and flexible environment for manipulating and interacting with web documents using JavaScript.

Certainly! Let's explore some other important objects and methods in the DOM along with examples.

1.Document Object:

- Represents the entire HTML document.
- Provides methods to access and manipulate the document's content.

Example:

```
// Access the document title
const title = document.title;
console.log('Document Title:', title);

// Change the document title
```

```
document.title = 'New Title';
console.log('New Document Title:', document.title);
```

2.Element Object:

- Represents an HTML element.
- Provides methods and properties to interact with elements.

Example:

```
// Get an element by ID
const myElement = document.getElementById('myElement');
console.log('Element by ID:', myElement);

// Add a CSS class to the element
myElement.classList.add('highlight');
```

3.NodeList Object:

- Represents a collection of nodes (e.g., returned by methods like `querySelectorAll()`).
- Allows iteration over nodes using methods like `forEach()`.

Example:

```
// Get all paragraphs in the document
const paragraphs = document.querySelectorAll('p');

// Iterate over paragraphs and log their text content
paragraphs.forEach(paragraph => {
    console.log(paragraph.textContent);
});
```

4.Event Object:

- Represents an event that occurs in the DOM.
- Contains information about the event (e.g., target element, event type).

Example:

```
// Add a click event listener to a button
const button = document.getElementById('myButton');
button.addEventListener('click', function(event) {
    console.log('Button clicked!');
    console.log('Target Element:', event.target);
});
```

5.Window Object:

- Represents the browser window or tab.
- Provides methods and properties for interacting with the window.

Example:

```
// Open a new window
const newWindow = window.open('https://www.example.com', '_blank');
console.log('New Window:', newWindow);

// Close the current window
window.close();
```

These examples demonstrate how to work with different objects in the DOM, such as accessing elements, handling events, and interacting with the browser window. Each object plays a crucial role in web development for building dynamic and interactive web applications.

DOM Get Method

In the DOM, the `getElementById` method is used to retrieve an element from the document based on its unique ID. This method is part of the `document` object and is commonly used in JavaScript to access specific elements for manipulation or interaction. Here's an example of how to use the `getElementById` method:

HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>DOM Get Method Example</title>
</head>
<body>
    <h1 id="mainHeading">Welcome to my Website</h1>
    <div id="content">
        <p>This is a paragraph.</p>
    </div>
    <script>
      // Get the element with ID "mainHeading"
      const heading = document.getElementById('mainHeading');

      // Change the text content and style of the heading
      heading.textContent = 'Hello World!';
      heading.style.color = 'blue';
    </script>
</body>
</html>
```

In this example:

- We have an HTML document with an `<h1>` element that has the ID "mainHeading" and a `<div>` element with the ID "content."
- In the JavaScript code, we use `document.getElementById('mainHeading')` to get the element with the ID "mainHeading" from the document.
- We then modify the text content of the heading using `heading.textContent` and change its color using `heading.style.color`.

After running this code, the text of the `<h1>` element with the ID "mainHeading" will be changed to "Hello World!" and its color will be set to blue.

DOM styling

Styling elements in the DOM is done using the `style` property of an element. This property allows you to directly modify the CSS styles of an element using JavaScript. Here's an example of how to style elements in the DOM:

HTML:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>DOM Styling Example</title>
    <style>
        .highlight {
            background-color: yellow;
            color: red;
            font-weight: bold;
        }
    </style>
</head>
<body>
    <h1>Welcome to my Website</h1>
    <p id="paragraph">This is a paragraph.</p>
    <button onclick="highlightText()">Highlight Paragraph</button>
    <script>
      function highlightText() {
        // Get the paragraph element
        const paragraph = document.getElementById('paragraph');

        // Add the 'highlight' class to apply styles
        paragraph.classList.add('highlight');
    }
    </script>
</body>
</html>
```

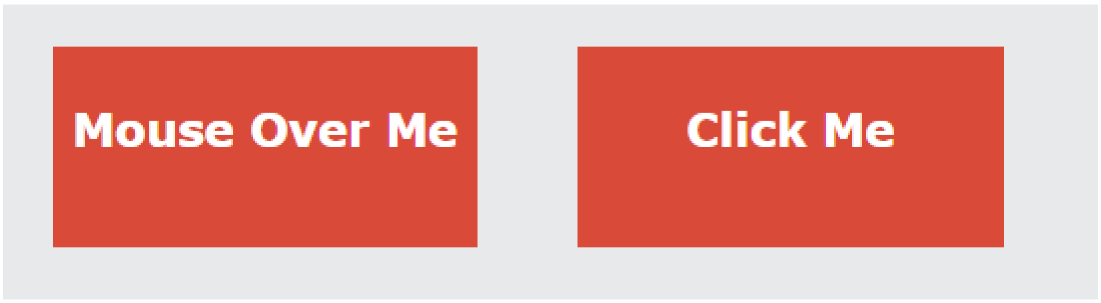
In this example:

- We have an HTML document with a `<p>` element that has the ID "paragraph" and a CSS style for the "highlight" class.
- The JavaScript code defines a function `highlightText()` that is triggered when a button is clicked.
- Inside the function, we use `document.getElementById('paragraph')` to get the paragraph element with the ID "paragraph."
- We then use `paragraph.classList.add('highlight')` to add the "highlight" class to the paragraph element, which applies the specified styles from the CSS.

When you click the "Highlight Paragraph" button, the background color of the paragraph will change to yellow, the text color will change to red, and the font weight will become bold, as defined in the CSS for the "highlight" class. This demonstrates how to dynamically apply styles to elements in the DOM using JavaScript.

Events

HTML DOM allows JavaScript to react to HTML events:



Reacting to Events

- A JavaScript can be executed when an event occurs, like when a user clicks on an HTML element.

- To execute code when a user clicks on an element, add JavaScript code to an HTML event attribute:

  ```
  onclick=JavaScript
  ```

Examples of HTML events:

- When a user clicks the mouse
- When a web page has loaded
- When an image has been loaded
- When the mouse moves over an element
- When an input field is changed
- When an HTML form is submitted
- When a user strokes a key

```
<!DOCTYPE html>
<html>
<body>
```

```
<h1 onclick="this.innerHTML = 'Ooops!'">Click on this text!</h1>

</body>
</html>
```

```
<!DOCTYPE html>
<html>
<body>

<h1 onclick="changeText(this)">Click on this text!</h1>

<script>
function changeText(id) {
  id.innerHTML = "Ooops!";
}
</script>

</body>
</html>
```

HTML Event Attributes

To assign events to HTML elements you can use event attributes.

Example

- Assign an onclick event to a button element:

```
<button onclick="displayDate()">Try it</button>
```

- In the example above, a function named displayDate will be executed when the button is clicked.

Assign Events Using the HTML DOM

The HTML DOM allows you to assign events to HTML elements using JavaScript:

Example

- Assign an onclick event to a button element:

```
<script>
document.getElementById("myBtn").onclick = displayDate;
</script>
```

- In the example above, a function named displayDate is assigned to an HTML element with the id="myBtn".

- The function will be executed when the button is clicked.

The onload and onunload Events

- The onload and onunload events are triggered when the user enters or leaves the page.
- The onload event can be used to check the visitor's browser type and browser version, and load the proper version of the web page based on the information.
- The onload and onunload events can be used to deal with cookies.

Example

```
<body onload="checkCookies()">
```

The oninput Event

- The oninput event is often to some action while the user input data.
- Below is an example of how to use the oninput to change the content of an input field.

Example

```
<input type="text" id="fname" oninput="upperCase()">
```

The onchange Event

- The onchange event is often used in combination with validation of input fields.
- Below is an example of how to use the onchange. The upperCase() function will be called when a user changes the content of an input field.

Example

```
<input type="text" id="fname" onchange="upperCase()">
```

AddEventListener

The `addEventListener` method in JavaScript is used to attach an event handler to an element. It allows you to listen for a specific event on the element, such as a click, mouseover, keypress, etc., and execute a function (event handler) when that event occurs. Here's an example of how to use `addEventListener`:

HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>addEventListener Example</title>
</head>
<body>
    <button id="myButton">Click Me</button>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js):

```
// Get the button element
const button = document.getElementById('myButton');

// Add a click event listener to the button
button.addEventListener('click', function() {
    alert('Button clicked!');
});
```

In this example:

- We have an HTML document with a `<button>` element that has the ID "myButton."
- The JavaScript code uses `document.getElementById('myButton')` to get the button element.
- We then use `addEventListener` to attach a click event listener to the button.
- When the button is clicked, the function inside `addEventListener` (`function() { alert('Button clicked!'); }`) is executed, showing an alert message.

You can attach event listeners for various events like click, mouseover, keypress, etc., and perform different actions based on those events. The `addEventListener` method is commonly used in web development to handle user interactions and create dynamic behavior on web pages.

DOM Forms

JavaScript Form Validation

- HTML form validation can be done by JavaScript.

If a form field (fname) is empty, this function alerts a message, and returns false, to prevent the form from being submitted:

```
function validateForm() {
  let x = document.forms["myForm"]["fname"].value;
  if (x == "") {
    alert("Name must be filled out");
    return false;
  }
}
```

The function can be called when the form is submitted:

```
<form name="myForm" action="/action_page.php" onsubmit="return validateForm()"
method="post">
Name: <input type="text" name="fname">
<input type="submit" value="Submit">
</form>
```

Automatic HTML Form Validation

- HTML form validation can be performed automatically by the browser:

If a form field (fname) is empty, the required attribute prevents this form from being submitted:

```
<form action="/action_page.php" method="post">
  <input type="text" name="fname" required>
  <input type="submit" value="Submit">
</form>
```

Data Validation

Data validation is the process of ensuring that user input is clean, correct, and useful.

Typical validation tasks are:

- has the user filled in all required fields?
- has the user entered a valid date?
- has the user entered text in a numeric field?
- Most often, the purpose of data validation is to ensure correct user input.

Validation can be defined by many different methods, and deployed in many different ways.

Server side validation is performed by a web server, after input has been sent to the server.

Client side validation is performed by a web browser, before input is sent to a web server.

HTML Constraint Validation

HTML5 introduced a new HTML validation concept called constraint validation.

HTML constraint validation is based on:

- Constraint validation HTML Input Attributes
- Constraint validation CSS Pseudo Selectors
- Constraint validation DOM Properties and Methods

Constraint Validation HTML Input Attributes

Attribute	Description

Attribute	Description
disabled	Specifies that the input element should be disabled
max	Specifies the maximum value of an input element
min	Specifies the minimum value of an input element
pattern	Specifies the value pattern of an input element
required	Specifies that the input field requires an element
type	Specifies the type of an input element

Constraint Validation CSS Pseudo Selectors

Selector	Description
:disabled	Selects input elements with the "disabled" attribute specified
:invalid	Selects input elements with invalid values
:optional	Selects input elements with no "required" attribute specified
:required	Selects input elements with the "required" attribute specified
:valid	Selects input elements with valid values
type	Specifies the type of an input element

DOM Animation

A Basic Web Page

To demonstrate how to create HTML animations with JavaScript, we will use a simple web page:

```
<!DOCTYPE html>
<html>
<body>

<h1>My First JavaScript Animation</h1>

<div id="animation">My animation will go here</div>

</body>
</html>
```

Create an Animation Container

All animations should be relative to a container element.

```
<div id ="container">
  <div id ="animate">My animation will go here</div>
```

```
</div>
```

Style the Elements

- The container element should be created with style = "position: relative".
- The animation element should be created with style = "position: absolute".

```
#container {
  width: 400px;
  height: 400px;
  position: relative;
  background: yellow;
}
#animate {
  width: 50px;
  height: 50px;
  position: absolute;
  background: red;
}
```

Animation Code

- JavaScript animations are done by programming gradual changes in an element's style.
- The changes are called by a timer. When the timer interval is small, the animation looks continuous.

```
id = setInterval(frame, 5);

function frame() {
  if (/* test for finished */) {
    clearInterval(id);
  } else {
    /* code to change the element style */
  }
}
```

Create the Full Animation Using JavaScript

```
function myMove() {
  let id = null;
  const elem = document.getElementById("animate");
  let pos = 0;
  clearInterval(id);
  id = setInterval(frame, 5);
  function frame() {
    if (pos == 350) {
      clearInterval(id);
    } else {
      pos++;
      elem.style.top = pos + 'px';
      elem.style.left = pos + 'px';
    }
  }
}
```

DOM Navigation

DOM Nodes

In the W3C HTML DOM standard, everything in an HTML document is considered a node:

- Document node: Represents the entire document.
- Element node: Represents HTML elements.
- Text node: Represents text content inside HTML elements.
- Attribute node: Represents HTML attributes (deprecated).
- Comment node: Represents comments within HTML.

DOM HTML Tree

The HTML DOM represents the structure of an HTML document as a tree. Each node in the tree can be accessed and manipulated using JavaScript.

Example:

```html
<html>
  <head>
    <title>DOM Tutorial</title>
  </head>
  <body>
    <h1>DOM Lesson one</h1>
    <p>Hello world!</p>
  </body>
</html>
```

Node Tree:

- `<html>` is the root node.
- `<head>` and `<body>` are children of `<html>`.
- `<head>` has one child: `<title>`.
- `<body>` has two children: `<h1>` and `<p>`.
- `<h1>` and `<p>` are siblings.

Node Relationships

Nodes in the DOM tree have hierarchical relationships:

- Parent: Every node (except the root) has one parent.
- Child: A node can have multiple children.

- Siblings: Nodes with the same parent.

Navigating Between Nodes

JavaScript provides several properties to navigate between nodes:

- `parentNode`
- `childNodes[nodenumber]`
- `firstChild`
- `lastChild`
- `nextSibling`
- `previousSibling`

Child Nodes and Node Values

Elements in the DOM may contain child nodes representing their content, which can be accessed using properties like `innerHTML`, `nodeValue`, or by navigating through child nodes.

Example:

`<title id="demo">DOM Tutorial</title>`

Accessing the text content:

```
var myTitle = document.getElementById("demo").innerHTML; // Using innerHTML
var myTitle = document.getElementById("demo").firstChild.nodeValue; // Accessing
firstChild
var myTitle = document.getElementById("demo").childNodes[0].nodeValue; //
Accessing childNodes
```

InnerHTML

`innerHTML` property allows retrieving or setting the HTML content of an element.

Example:

`<script> document.getElementById("id02").innerHTML = document.getElementById("id01").innerHTML; </script>`

DOM Root Nodes

Special properties allow access to the entire document:

- `document.body`: Represents the body of the document.
- `document.documentElement`: Represents the full document.

Example:

`<script> document.getElementById("demo").innerHTML = document.body.innerHTML; </script>`

The `nodeName` Property

`nodeName` specifies the name of a node:

- For an element node, it's the tag name.
- For an attribute node, it's the attribute name.
- For a text node, it's always `#text`.
- For the document node, it's always `#document`.

Example:

`<script> document.getElementById("id02").innerHTML = document.getElementById("id01").nodeName; </script>`

The `nodeValue` Property

`nodeValue` specifies the value of a node:

- For an element node, it's `null`.
- For a text node, it's the text content.
- For an attribute node, it's the attribute value.

The `nodeType` Property

`nodeType` returns the type of a node:

- `ELEMENT_NODE (1)`: Represents an HTML element.
- `ATTRIBUTE_NODE (2)`: Represents an HTML attribute (deprecated).
- `TEXT_NODE (3)`: Represents text content.
- `COMMENT_NODE (8)`: Represents comments.
- `DOCUMENT_NODE (9)`: Represents the entire HTML document.

Example:

`<script> document.getElementById("id02").innerHTML = document.getElementById("id01").nodeType; </script>`

DOM Node

Creating New HTML Elements (Nodes)

To add a new element to the HTML DOM, you first create the element (element node) and then append it to an existing element.

Example:

```
<div id="div1">
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
</div>

<script> const para = document.createElement("p"); // Create a new <p> element
const node = document.createTextNode("This is new."); // Create a text node
para.appendChild(node); // Append the text node to the <p> element
```

```
const element = document.getElementById("div1"); // Find an existing element
element.appendChild(para); // Append the new element to the existing element
</script>
```

Creating New HTML Elements - insertBefore()

If you want to specify the position of the new element, you can use the `insertBefore()` method instead of `appendChild()`.

Example:

```
<div id="div1">
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
</div>

<script> const para = document.createElement("p");
const node = document.createTextNode("This is new.");
para.appendChild(node);

const element = document.getElementById("div1");
const child = document.getElementById("p1");
element.insertBefore(para, child); // Insert the new element before an existing
element </script>
```

Removing Existing HTML Elements

To remove an HTML element, you can use the `remove()` method or `removeChild()` method for older browsers.

Example:

```
<div>
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
</div>

<script> const elmnt = document.getElementById("p1");
elmnt.remove(); // Remove the specified element </script>
```

Removing a Child Node

For browsers that don't support the `remove()` method, you can use the `removeChild()` method by first finding the parent node.

Example:

```
<div id="div1">
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
</div>

<script> const parent = document.getElementById("div1");
const child = document.getElementById("p1");
parent.removeChild(child); // Remove the specified child element from its parent
</script>
```

Replacing HTML Elements

To replace an element in the HTML DOM, you can use the `replaceChild()` method.

Example:

```
<div id="div1">
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
</div>

<script> const para = document.createElement("p");
const node = document.createTextNode("This is new.");
para.appendChild(node);

const parent = document.getElementById("div1");
const child = document.getElementById("p1");
parent.replaceChild(para, child); // Replace the specified child element with a
new element </script>
```

These examples demonstrate how to manipulate the DOM by adding, removing, and replacing HTML elements dynamically using JavaScript.

DOM Collections

The HTMLCollection Object

- The getElementsByTagName() method returns an HTMLCollection object.
- An HTMLCollection object is an array-like list (collection) of HTML elements.

The following code selects all `<p>` elements in a document:

```
const myCollection = document.getElementsByTagName("p");
```

The elements in the collection can be accessed by an index number.

To access the second `<p>` element you can write:

```
myCollection[1]
```

HTML HTMLCollection Length

The length property defines the number of elements in an `HTMLCollection`:

```
myCollection.length
```

The `length` property is useful when you want to loop through the elements in a collection:

```
const myCollection = document.getElementsByTagName("p");
for (let i = 0; i < myCollection.length; i++) {
  myCollection[i].style.color = "red";
}
```

DOM Node Lists

The HTML DOM NodeList Object

- A NodeList object is a list (collection) of nodes extracted from a document.
- A NodeList object is almost the same as an HTMLCollection object.
- Some (older) browsers return a NodeList object instead of an HTMLCollection for methods like getElementsByClassName().
- All browsers return a NodeList object for the property childNodes.
- Most browsers return a NodeList object for the method querySelectorAll().

The following code selects all `<p>` nodes in a document:

```
const myNodeList = document.querySelectorAll("p");
```

- The elements in the NodeList can be accessed by an index number.
- To access the second `<p>` node you can write:

```
myNodeList[1]
```

HTML DOM Node List Length

The length property defines the number of nodes in a node list:

```
myNodelist.length
```

The length property is useful when you want to loop through the nodes in a node list:

```
const myNodelist = document.querySelectorAll("p");
for (let i = 0; i < myNodelist.length; i++) {
  myNodelist[i].style.color = "red";
}
```

JavaScript Errors

Throw, and Try...Catch...Finally

- The `try` statement defines a code block to run (to try).
- The `catch` statement defines a code block to handle any error.
- The `finally` statement defines a code block to run regardless of the result.
- The `throw` statement defines a custom error.

Errors Will Happen!

- When executing JavaScript code, different errors can occur.
- Errors can be coding errors made by the programmer, errors due to wrong input, and other unforeseeable things.

In this example we misspelled "alert" as "adddlert" to deliberately produce an error:

```
<p id="demo"></p>

<script>
try {
  adddlert("Welcome guest!");
}
catch(err) {
  document.getElementById("demo").innerHTML = err.message;
}
</script>
```

JavaScript try and catch

- The try statement allows you to define a block of code to be tested for errors while it is being executed.
- The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.
- The JavaScript statements try and catch come in pairs:

```
try {
  Block of code to try
}
catch(err) {
  Block of code to handle errors
}
```

JavaScript Throws Errors

- When an error occurs, JavaScript will normally stop and generate an error message.
- The technical term for this is: JavaScript will throw an exception (throw an error).

The throw Statement

- The throw statement allows you to create a custom error.
- Technically you can throw an exception (throw an error).
- The exception can be a JavaScript `String`, a `Number`, a `Boolean` or an `Object`:

```
throw "Too big";    // throw a text
throw 500;          // throw a number
```

If you use throw together with try and catch, you can control program flow and generate custom error messages.

Input Validation Example

- This example examines input. If the value is wrong, an exception (err) is thrown.
- The exception (err) is caught by the catch statement and a custom error message is displayed:

```html
<!DOCTYPE html>
<html>
<body>

<p>Please input a number between 5 and 10:</p>

<input id="demo" type="text">
<button type="button" onclick="myFunction()">Test Input</button>
<p id="p01"></p>

<script>
function myFunction() {
  const message = document.getElementById("p01");
  message.innerHTML = "";
  let x = document.getElementById("demo").value;
  try {
    if(x.trim() == "") throw "empty";
    if(isNaN(x)) throw "not a number";
    x = Number(x);
    if(x < 5) throw "too low";
    if(x > 10) throw "too high";
  }
  catch(err) {
    message.innerHTML = "Input is " + err;
  }
}
</script>

</body>
</html>
```

HTML Validation

The code above is just an example.

Modern browsers will often use a combination of JavaScript and built-in HTML validation, using predefined validation rules defined in HTML attributes:

```
<input id="demo" type="number" min="5" max="10" step="1">
```

The finally Statement

The `finally` statement lets you execute code, after try and catch, regardless of the result:

```
try {
  Block of code to try
}
catch(err) {
  Block of code to handle errors
}
finally {
  Block of code to be executed regardless of the try / catch result
}
```

Example

```
function myFunction() {
  const message = document.getElementById("p01");
  message.innerHTML = "";
  let x = document.getElementById("demo").value;
  try {
    if(x.trim() == "") throw "is empty";
    if(isNaN(x)) throw "is not a number";
    x = Number(x);
    if(x > 10) throw "is too high";
    if(x < 5) throw "is too low";
  }
  catch(err) {
    message.innerHTML = "Error: " + err + ".";
  }
  finally {
    document.getElementById("demo").value = "";
  }
}
```

The Error Object

- JavaScript has a built in error object that provides error information when an error occurs.
- The error object provides two useful properties: name and message.

Error Object Properties

Property	Description
name	Sets or returns an error name
message	Sets or returns an error message (a string)

Error Name Values

Error Name	Description
EvalError	An error has occurred in the eval() function
RangeError	A number "out of range" has occurred
ReferenceError	An illegal reference has occurred
SyntaxError	A syntax error has occurred
TypeError	A type error has occurred
URIError	An error in encodeURI() has occurred

Range Error

- A RangeError is thrown if you use a number that is outside the range of legal values.
- For example: You cannot set the number of significant digits of a number to 500.

```
let num = 1;
try {
  num.toPrecision(500);   // A number cannot have 500 significant digits
}
catch(err) {
  document.getElementById("demo").innerHTML = err.name;
}
```

Reference Error

A ReferenceError is thrown if you use (reference) a variable that has not been declared:

```
let x = 5;
try {
  x = y + 1;   // y cannot be used (referenced)
}
catch(err) {
  document.getElementById("demo").innerHTML = err.name;
}
```

Syntax Error

A SyntaxError is thrown if you try to evaluate code with a syntax error.

```
try {
  eval("alert('Hello)");   // Missing ' will produce an error
}
catch(err) {
  document.getElementById("demo").innerHTML = err.name;
}
```

Type Error

A TypeError is thrown if an operand or argument is incompatible with the type expected by an operator or function.

```
let num = 1;
try {
  num.toUpperCase();   // You cannot convert a number to upper case
}
catch(err) {
  document.getElementById("demo").innerHTML = err.name;
}
```

URI (Uniform Resource Identifier) Error

A `URIError` is thrown if you use illegal characters in a URI function:

```
try {
  decodeURI("%%%");   // You cannot URI decode percent signs
}
catch(err) {
  document.getElementById("demo").innerHTML = err.name;
}
```