**Hashing:**

* **Purpose:** The main purpose of hashing is to create a fixed-size string of characters from input data.
* **Irreversibility:** Hashing is a one-way function, meaning it cannot be reversed to obtain the original input data.
* **Security:** Hashing is used to verify data integrity, meaning you can compare two hashes to check if the original data has changed.
* **Output:** Hash functions always produce a fixed-size output, regardless of the size of the input.
* **Common Use Cases:** Password storage (storing hashed passwords instead of plaintext), verifying data integrity (ensuring data hasn't been tampered with), creating checksums for files.

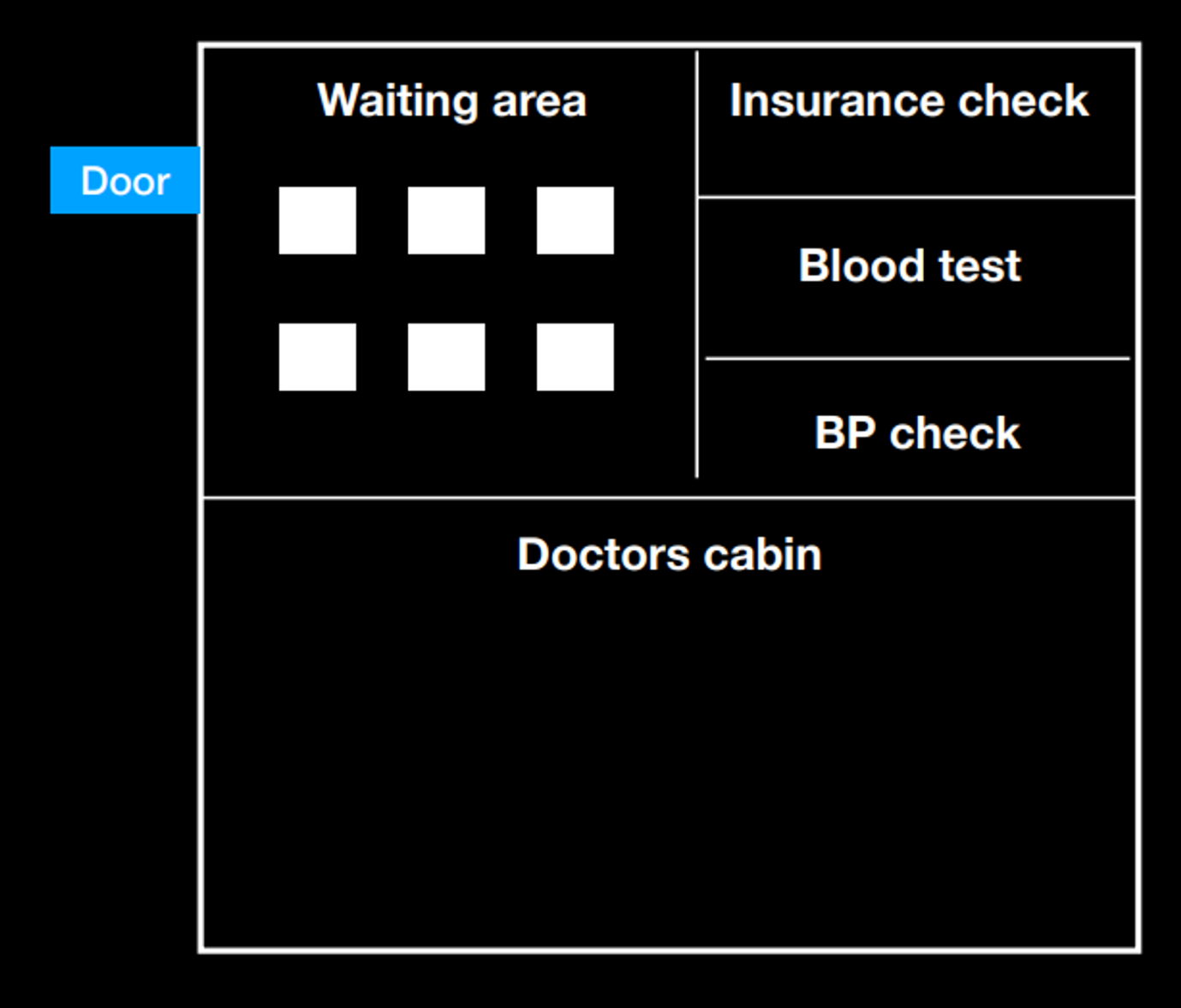
**Encryption:**

* **Purpose:** Encryption is used to protect data by transforming it into a ciphertext that can only be decrypted back to its original form with the correct key.
* **Reversibility:** Encryption is a two-way process - data is encrypted with a key to produce ciphertext, and then decrypted with the same key to get back the original data.
* **Security:** The security of encrypted data depends on the strength of the encryption algorithm and the secrecy of the key.
* **Output:** The output of encryption is typically variable in size, depending on the size of the input and the encryption algorithm used.
* **Common Use Cases:** Secure communication (like HTTPS for websites), protecting sensitive data (credit card information, personal information), securing files and emails.

**Middlewares, Global Catches & Zod**

**In this lecture, Harkirat dives deep into** M**iddlewares:** behind-the-scenes helpers that tidy up things before your main code does its thing. G**lobal catches:** safety nets for your code, they catch unexpected issues before they cause chaos. And finally, Zod: a library that ensures efficient input validation on your behalf.

[Middlewares, Global Catches & Zod](https://app.100xdevs.com/courses/3/23/158#43e25ea3ece54679a4d6c7e63b20c59b)[Understanding Middlewares:](https://app.100xdevs.com/courses/3/23/158#59e078a1b41d4f88af7d95f72ee07b77)[Middlewares in JS Context & Problem Statement:](https://app.100xdevs.com/courses/3/23/158#360a4f69b08b4a808556cc82d6683c17)[Solution: Middlewares](https://app.100xdevs.com/courses/3/23/158#abdd65f2a8e448f182f931ce1997efd8)[Some Associated Concepts:](https://app.100xdevs.com/courses/3/23/158#b48ef4278332459198a0ad9e5399a010)[1. next() Keyword:](https://app.100xdevs.com/courses/3/23/158#d5c7c6465c60495596c13eb83c97b528)[2. Difference between res.send and res.json:](https://app.100xdevs.com/courses/3/23/158#056f3a5278df4c7792949bebe4dfd344)[3. Importance of app.use(express.json()):](https://app.100xdevs.com/courses/3/23/158#bd48b88480d846348c3084ab11255b51)[4. Middleware and req.body:](https://app.100xdevs.com/courses/3/23/158#1a30eca4c8214868912fc073b113a834)[3 Ways of Sending Inputs to a Response:](https://app.100xdevs.com/courses/3/23/158#3b9becaf58884926ab7c4d3627a983a4)[1. Query Parameter:](https://app.100xdevs.com/courses/3/23/158#82ea70ce4a4844f6973b979337e369cc)[2. Body:](https://app.100xdevs.com/courses/3/23/158#7be5d6846dfc4224999559ff16a6be02)[3. Headers:](https://app.100xdevs.com/courses/3/23/158#cef023f07fba44858db51d79f57a6919)[Bottom Line:](https://app.100xdevs.com/courses/3/23/158#608f0f4e655b4acb9e9f3091f6d4771f)[Global Catches:](https://app.100xdevs.com/courses/3/23/158#e56a14f3766b452f81316ac840c4c8b7)[Importance of Global Error Handling:](https://app.100xdevs.com/courses/3/23/158#f3b69f9ca6bc41438d063e660c89aba3)[Input Validation:](https://app.100xdevs.com/courses/3/23/158#5e0cda5cb7a746f3b476dc69e2797787)[1. Naive Way - Multiple If-Else Statements:](https://app.100xdevs.com/courses/3/23/158#8dda6a1e015b4ad2826a7a5b9986e129)[2. Using zod Library for Schema Validation:](https://app.100xdevs.com/courses/3/23/158#d66ca50e06b7441483e219a9d6ca4d98)[Zod:](https://app.100xdevs.com/courses/3/23/158#1734a97d6c16434fa292f26f4e5cd1e1)[Zod Syntax Overview:](https://app.100xdevs.com/courses/3/23/158#7c852002394e45a69c0a1857f9156428)[Why Zod:](https://app.100xdevs.com/courses/3/23/158#eecf114c1eb743009414b34e4b483b68)

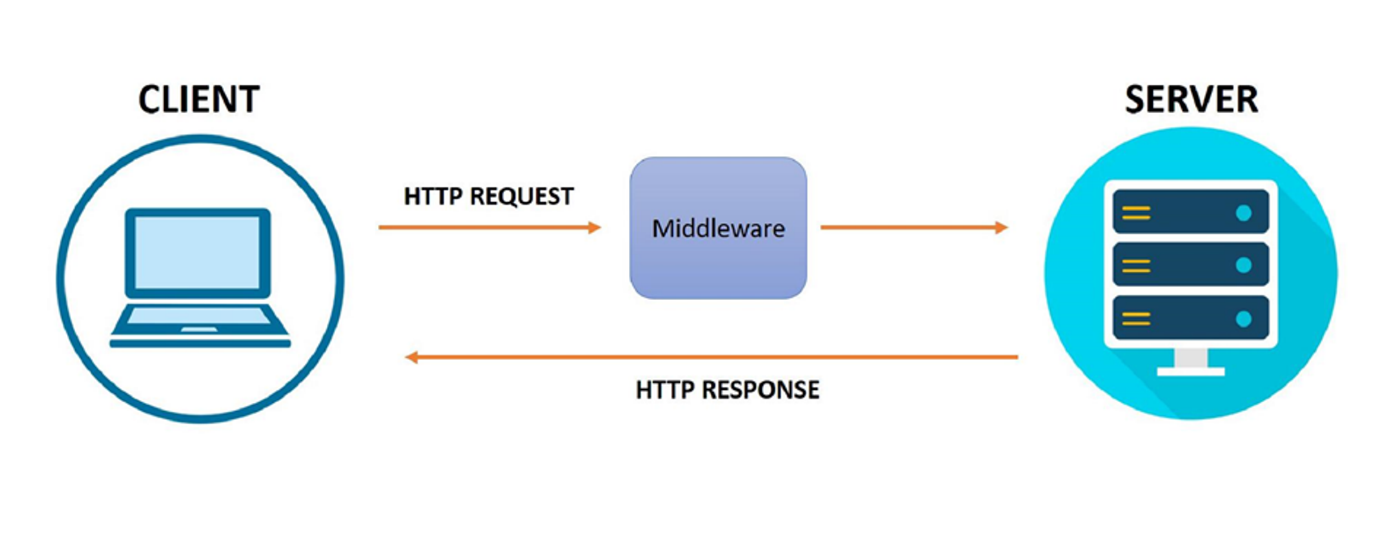


**Understanding Middlewares:**

**Imagine a Busy Hospital:**

Think of a hospital where there's a doctor, patients waiting in line, and a few helpful assistants making sure everything runs smoothly.

1. **Doctor's Cabin (Application Logic) :** 
   * + The doctor is like the main brain of our hospital – ready to help patients with their problems.
2. **Waiting Room (Callback Queue) :**
   * + The waiting room is where patients hang out before seeing the doctor. Each patient has a unique situation.
3. Intermediates **(Middlewares) :**
   * + Before a patient sees the doctor, there are some helpers doing important tasks.
     + One helper checks if patients have the right paperwork . This is like ensuring everyone is who they say they are (Authentication)
     + Another helper does quick health checks – like making sure patients' blood pressure is okay. This is similar to checking if the information coming to the doctor is healthy and makes sense (Input Validation)

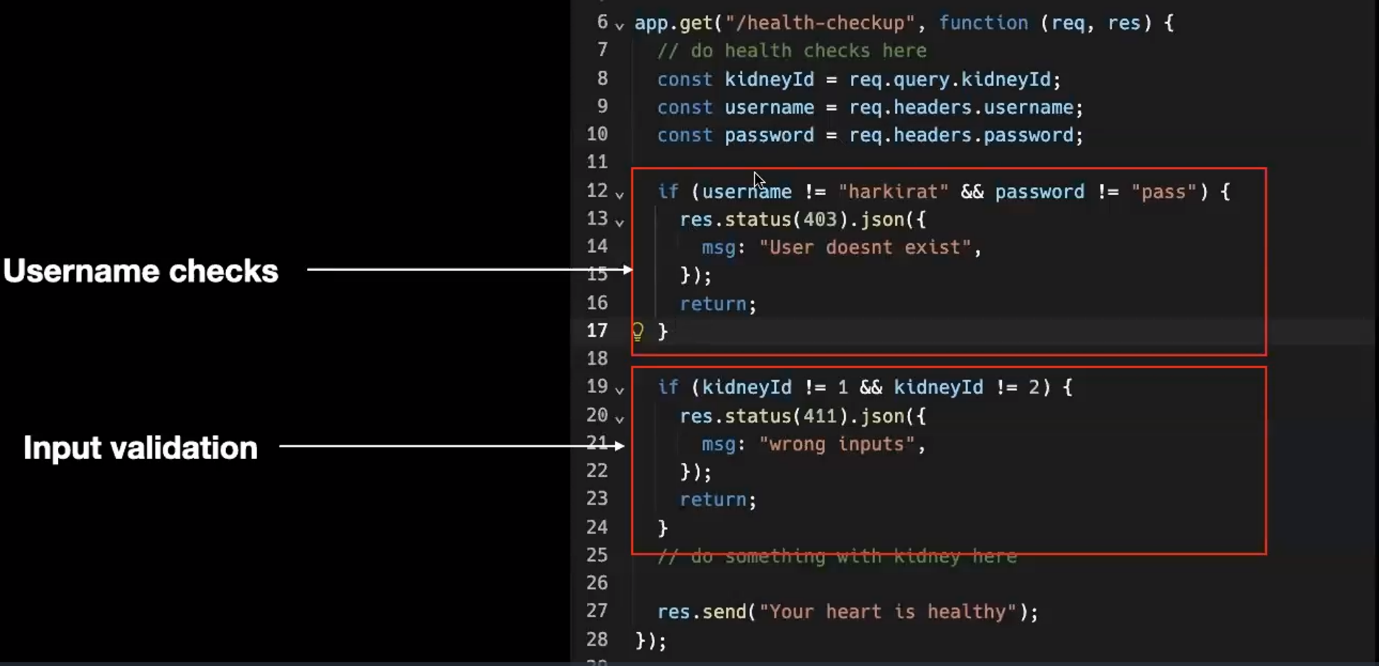


**Middlewares in JS Context & Problem Statement:**

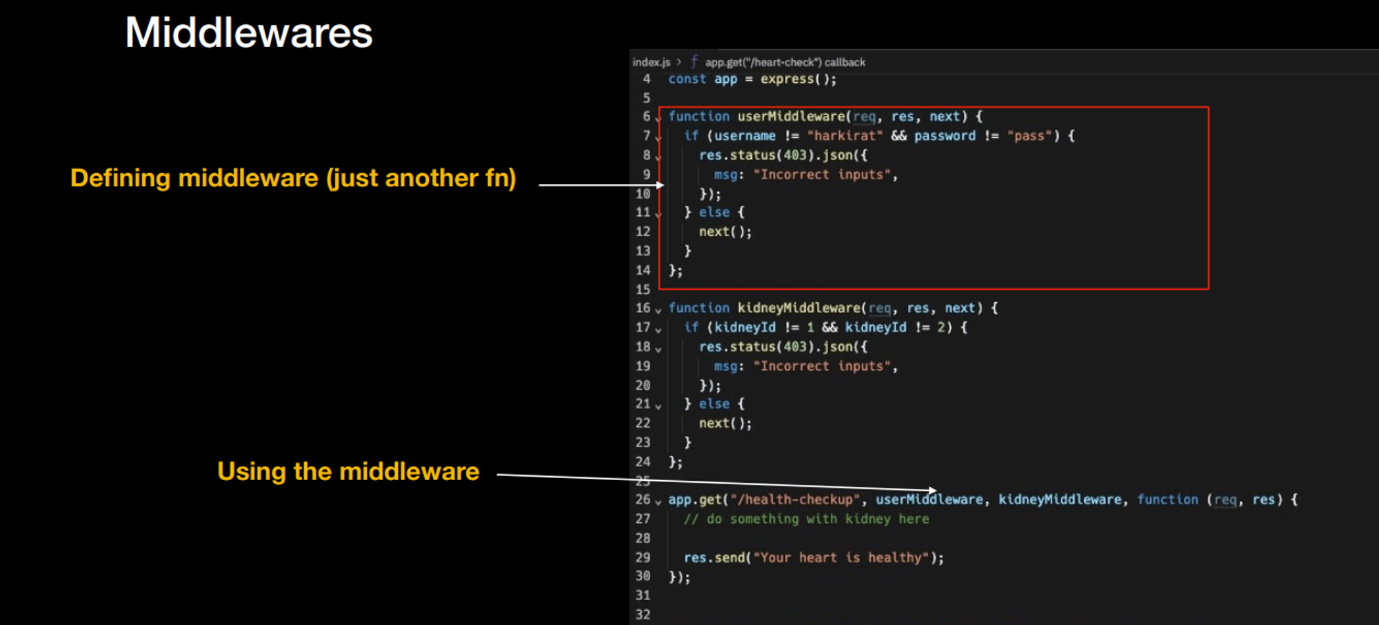
Earlier we used to organize all our prechecks followed by the application logic all in one route.

Middlewares emerged as a solution to enhance code organization by extracting prechecks from the core application logic. The motivation behind their introduction lies in our commitment to the "Don't Repeat Yourself" (DRY) principle.

By isolating these preliminary checks into distinct functions or code blocks known as middlewares, we achieve a more modular and maintainable codebase. This separation not only streamlines the primary application logic but also promotes code reuse, making it easier to manage, understand, and scale our software architecture.



**Solution: Middlewares**



Furthermore, with middleware, we can easily include as many precheck functions as needed. This means we have the freedom to add various checks or operations to our application without making the main code complex. It's like having building blocks that we can mix and match to create a customized process for our application, making it more adaptable and easier to manage. Here userMiddleware and kidneyMiddleware

**Some Associated Concepts:**

**1. next() Keyword:**

In middleware functions in Express, next is a callback function that is used to pass control to the next middleware function in the stack. When you call next(), it tells Express to move to the next middleware in line. If next() is not called within a middleware function, the request-response cycle stops, and the client receives no response.

Example:

**2. Difference between res.send and res.json:**

* res.send: Sends a response of various types (string, Buffer, object, etc.). Express tries to guess the content type based on the data provided.
* res.json: Sends a JSON response. It automatically sets the Content-Type header to application/json.

**3. Importance of app.use(express.json()):**

app.use(express.json()) is middleware that parses incoming JSON payloads in the request body. It is crucial when dealing with JSON data sent in the request body, typically in POST or PUT requests. Without this middleware, you might receive the JSON data as a raw string, and you'd need to manually parse it.

Example:

**4. Middleware and req.body:**

* req.query and req.headers don't require middleware because they represent the query parameters and headers of the incoming request, respectively. Express automatically parses them.
* req.body requires middleware like express.json() to parse the request body, especially when the body contains JSON data. Other middleware, like express.urlencoded(), is used for parsing form data in the request body.

Middleware helps in processing the request at different stages and is essential for tasks like parsing, logging, authentication, and more in a modular and organized way.

**3 Ways of Sending Inputs to a Response:**

**1. Query Parameter:**

* **What it is:** Like giving specific instructions in the web address.
* **Example:** In **www.example.com/search?topic=animals**, the query parameter is **topic** with the value **animals**.
* **Use Case:** Good for simple stuff you want everyone to see, like search terms in a URL.

**2. Body:**

* **What it is:** Imagine it as the hidden part of a request, carrying more detailed information.
* **Example:** When you fill out a form on a website, the details you enter (name, email) go in the body of the request.
* **Use Case:** Great for sending lots of information, especially when you're submitting something like a form.

**3. Headers:**

* **What it is:** Extra information attached to the request, kind of like details about a letter.
* **Example:** Headers could include things like your identity or the type of data you're sending.
* **Use Case:** Perfect for passing along special information that doesn't fit neatly in the URL or body, like who you are or how to handle the data.

**Bottom Line:**

* **Query Parameters:** Simple instructions visible in the web address.
* **Body:** Hidden part of the request for more detailed info, great for forms.
* **Headers:** Extra details about the request, useful for special information.

**Global Catches:**

It essentially help us the developers give a better error message to the user.

Global Catch or Error-Handling Middleware is a special type of middleware function in Express that has four arguments instead of three ((err, req, res, next)). Express recognizes it as an error-handling middleware because of these four arguments.

**Importance of Global Error Handling:**

1. **Centralized Handling:**
   * + Global catch blocks allow you to centrally manage and handle errors that occur anywhere in your application. Instead of handling errors at each specific location, you can capture and process them in a centralized location.
2. **Consistent Error Handling:**
   * + Using a global catch mechanism ensures a consistent approach to error handling throughout the application. You can define how errors are logged, reported, or displayed in one place, making it easier to maintain a uniform user experience.
3. **Fallback Mechanism:**
   * + Global catches often serve as a fallback mechanism. If an unexpected error occurs and is not handled locally, the global catch can capture it, preventing the application from crashing and providing an opportunity to log the error for further analysis.

**Input Validation:**

Input validation is a crucial aspect of securing your application. It helps ensure that the data received by your server is in the expected format and meets certain criteria. Take for instance a login schema, now instead of passing a username and password in the body, the user can pass in any gibberish and may try to crash the server. Thus, it is our responsibility to ensure that our application logic handles all these input vulnerabilities. Let's explore two approaches to input validation: the naive way with multiple if-else statements and using the **zod** library for schema validation.

**1. Naive Way - Multiple If-Else Statements:**

In the naive approach, you manually check each input parameter to ensure it meets your criteria. Here's an example using Express.js:

In this example, we manually check the username and password fields for their existence, data type, and minimum length. This approach can become cumbersome as the number of input parameters increases, and it may lead to code duplication.

**2. Using zod Library for Schema Validation:**

zod is a TypeScript-first schema declaration and validation library. It provides a concise way to define schemas and validate input data. Here's an example using zod for the same login scenario:

In this example, we define a loginSchema using zod that specifies the expected structure and constraints for the input data. The parse method is then used to validate the input against the schema. If the input is invalid, zod throws an error, and we can handle it appropriately. This approach is more concise and less error-prone compared to the manual if-else checks.

**Zod:**

Zod is a TypeScript-first schema declaration and validation library. It provides a simple and expressive way to define the structure and constraints of your data, allowing you to easily validate and parse input against those specifications. Here's a brief explanation of Zod and its syntax:

**Zod Syntax Overview:**

1. **Basic Types:**
   * + Zod provides basic types such as string, number, boolean, null, undefined, etc.
2. **Object Schema:**
   * + You can define the structure of an object using the object method and specify the shape of its properties.
3. **Nested Schemas:**
   * + You can nest schemas within each other to create more complex structures.
4. **Array Schema:**
   * + You can define the schema for arrays using the array method.
5. **Union and Intersection Types:**
   * + Zod supports union and intersection types for more flexibility.
6. **Optional and Nullable:**
   * + You can make properties optional or nullable using optional and nullable methods.
7. **Custom Validators:**
   * + Zod allows you to define custom validation logic using the refine method.
8. **Parsing and Validation:**
   * + To validate and parse data, use the parse method. If the data is invalid, it throws an error with details about the validation failure.

**Why Zod:**

* **TypeScript-First Approach:** Zod is designed with TypeScript in mind, providing strong type-checking and autocompletion for your schemas.
* **Concise and Expressive Syntax:** Zod's syntax is concise and expressive, making it easy to define complex data structures with minimal code.
* **Validation and Parsing:** Zod not only validates data but also automatically parses it into the expected TypeScript types.
* **Rich Set of Features:** Zod includes a variety of features, such as custom validation, optional and nullable types, union and intersection types, making it a powerful tool for data validation in your applications.

Overall, Zod simplifies the process of declaring and validating data structures, reducing the likelihood of runtime errors and improving the overall robustness of your code.

A black background with white text

Description automatically generated  
The JWT is like a bank check that all can see the signature but only the bank can verify it  
The jwt has 2 main functions:

Jwt.sign(object,password) =>The object is a JSON that we want to encrypt  
Jwt.verify(token,password) =>To verify the token with the same password as used for sign we will be returned with

A black background with white text

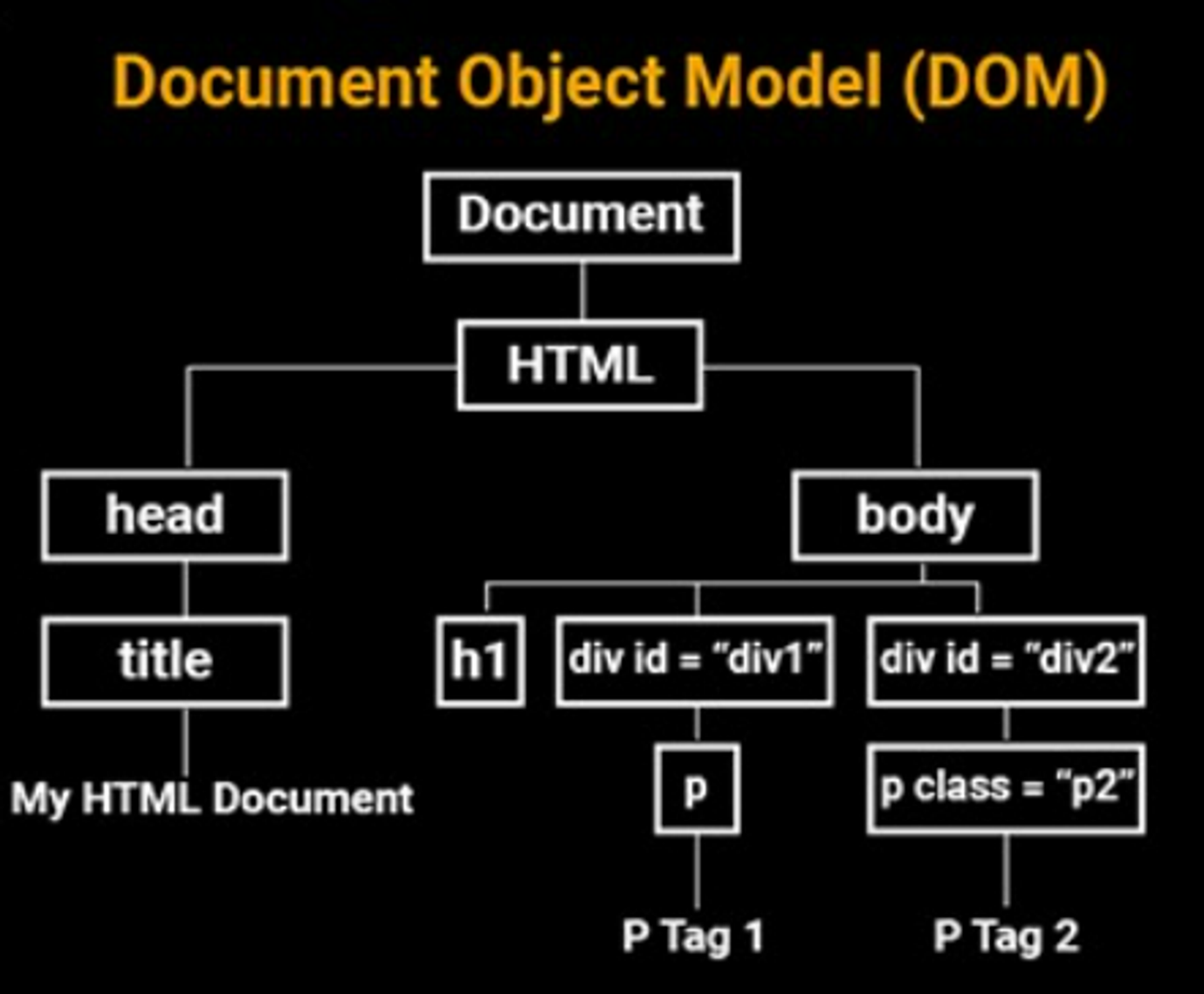
Description automatically generated

A black background with white text

Description automatically generated

**DOM**

You know how a remote control brings a television to life by letting you change channels and do cool things? Well, in the web world, JavaScript is like that remote control for your HTML page, making it active and dynamic. And the secret sauce behind this magic is the DOM – the Document Object Model.



**What is DOM?**

So, what's this DOM thing? DOM stands for Document Object Model. It's like the behind-the-scenes framework that JavaScript uses to talk to your browser. Imagine it as the language that JavaScript speaks with your web browser to make things happen on your HTML page.

**Communication with the Browser**

JavaScript and the browser communicate through a set of tools in this magical interface known as the DOM. These tools include properties, methods, and events. It's like having a language to tell your browser what to do and when to do it.

**Accessing the DOM**

Okay, how do we get our hands on this DOM magic? Well, accessing the DOM is like reaching for that remote control. In JavaScript, you use commands to grab elements from your HTML page, change their content, or even create new elements. It's like giving instructions to your browser using JavaScript.

**Possibilities of DOM**

Now, let's talk about the possibilities the DOM opens up. With JavaScript and the DOM, you can:

* Change the content of your webpage dynamically.
* Update styles and layout on the fly.
* Respond to user interactions, like clicks or keyboard inputs.
* Add or remove elements, making your page super interactive.

**Independence and Consistency**

DOM doesn't pick sides. It's independent of any particular programming language. This means whether you're using JavaScript, Python, or any other language, the DOM provides a consistent way to interact with your document. It's like a universal remote that works with any TV.

**In a Nutshell:** So, the DOM is your backstage pass to making HTML pages come alive with JavaScript. It's a set of rules and tools that allow you to control, change, and interact with your webpage dynamically. It's like giving your webpage a personality and making it respond to your JavaScript commands.

**DOM Tree**

The DOM tree, or The Document Object Model tree, is a hierarchical representation of the structure of a web document in the context of web development. It's essentially a way to organize and navigate the elements of an HTML or XML document. Here's a breakdown:

* **Document Object:** At the top of the tree is the Document Object, representing the entire web document.
* **HTML Element:** The HTML element comes next, serving as the container for the entire document.
* **Head and Body Elements:** Within the HTML element, there are two main sections: the Head and the Body. The Head typically contains meta-information, styles, and links to external resources, while the Body holds the primary content visible on the webpage.
* **Further Nesting:** Each of these main sections may contain further nested elements. For instance, the Head could include elements like title, meta, or link, while the Body could include paragraphs, images, buttons, and other content-related elements.

The DOM tree essentially forms a family tree-like structure, where elements are organized in a hierarchy based on their relationships with each other. Understanding the DOM tree is crucial for web developers because it provides a structured way to interact with and manipulate the content of a webpage using programming languages like JavaScript.

**Implementing innerHTML**

The below implementation consists of an input field that allows users to enter their name. Accompanying this, a button is present, equipped with an onclick attribute that invokes the displayGreeting function. Within this function, the entered name is acquired by accessing the value of the input field through document.getElementById("nameInput").value. Subsequently, the function utilizes innerHTML to dynamically alter the content of the <p> element identified by the id "greetingMessage," facilitating the display of a personalized greeting message.

**Difference Between HTMLCollection and NodeList:**

Both HTMLCollections and NodeLists are collections of nodes in the Document Object Model (DOM) provided by JavaScript, but they have some key differences:

**HTMLCollection:**

1. **Live Collection:**
   * + **Live:** An HTMLCollection is live, meaning it is automatically updated when the underlying document changes. If elements are added or removed, the HTMLCollection is automatically updated to reflect these changes.
2. **Accessing Elements:**
   * + **By Index:** Elements in an HTMLCollection can be accessed using numerical indices, similar to an array.
3. **Methods:**
   * + **Limited Methods:** HTMLCollections have a more limited set of methods compared to NodeLists.
4. **Specific to Elements:**
   * + **Element-Specific:** HTMLCollections are typically used for collections of HTML elements, such as those returned by getElementsByTagName or getElementsByClassName.

**NodeList:**

1. **Live or Static:**
   * + **Live or Static:** A NodeList can be live or static. If it's obtained using querySelectorAll, it's static and won't automatically update. If it's obtained by other means, like childNodes, it might be live.
2. **Accessing Elements:**
   * + **By Index or forEach:** Like HTMLCollection, you can access elements by index. Additionally, NodeList supports the forEach method for iteration.
3. **Methods:**
   * + **Richer Set of Methods:** NodeLists typically have a broader set of methods compared to HTMLCollections.
4. **Not Limited to Elements:**
   * + **Node-Oriented:** NodeLists can include various types of nodes, not just HTML elements. They might include text nodes, comment nodes, etc.

**Practical Considerations:**

* **Common Methods:**
  + For general purpose, when using methods like querySelectorAll, you will get a NodeList.
* **Live vs. Static:**
  + If you need a live collection that automatically updates, an HTMLCollection might be suitable.
  + If you want a static collection that won't change, or if you need a broader range of methods, a NodeList might be preferable.
* **Usage:**
  + HTMLCollections are often associated with specific methods like getElementsByClassName or getElementsByTagName.
  + NodeLists are often the result of more generic methods like querySelectorAll or properties like childNodes.

In summary, the choice between HTMLCollection and NodeList depends on your specific needs, especially regarding the liveliness of the collection and the methods you require for manipulation.

**HTMLCollection**

**Finding HTML Elements**

**By ID:**

To find an HTML element by its ID, you can use the getElementById method.

**By Tag Name:**

To find HTML elements by their tag name, you can use the getElementsByTagName method.

**By Class Name:**

To find HTML elements by their class name, you can use the getElementsByClassName method.

**By CSS Selector:**

To find HTML elements using CSS selectors, you can use the querySelector or querySelectorAll methods.

**By HTML Object Collections:**

To find HTML elements using HTML collections, you can use methods like getElementsByName or getElementsByName in specific cases.

These methods provide different ways to locate and interact with HTML elements in a document using JavaScript. Choose the appropriate method based on your specific needs and the structure of your HTML document.

**Changing HTML Elements**

Changing HTML elements dynamically is a fundamental aspect of web development, and JavaScript provides several methods to achieve this. Here are some commonly used methods for changing HTML elements:

1. **innerHTML:**
   * + **Purpose:** Changes the HTML content (including tags) of an element.
     + **Example:**
2. **textContent:**
   * + **Purpose:** Changes the text content of an element, excluding HTML tags.
     + **Example:**
3. **setAttribute:**
   * + **Purpose:** Sets the value of an attribute on an element.
     + **Example:**
4. **style:**
   * + **Purpose:** Modifies the inline styles of an element.
     + **Example:**
5. **classList:**
   * + **Purpose:** Provides methods to add, remove, or toggle CSS classes on an element.
     + **Examples:**
6. **appendChild:**
   * + **Purpose:** Adds a new child element to an existing element.
     + **Example:**
7. **removeChild:**
   * + **Purpose:** Removes a child element from its parent.
     + **Example:**
8. **setAttribute:**
   * + **Purpose:** Sets or changes the value of an attribute on an HTML element.
     + **Example:**

These methods provide a diverse set of tools for us —developers to manipulate HTML elements dynamically, whether it's updating content, changing styles, or modifying attributes. The choice of method depends on the specific requirement and the nature of the change you want to apply.

**Example - using setAttribute to change an input field to a button:**

In this example, an input field with the id "myInput" is initially present, alongside a button labeled "Change to Button." Clicking this button triggers the **changeToButton** function, wherein a new button is dynamically created using **createElement**. Key attributes (type and onclick) are set via **setAttribute**, and the input field is promptly replaced by this newly fashioned button using **replaceChild**. The outcome is a dynamic transformation, demonstrating the capability to swap an input field for a button upon clicking "Change to Button," complete with an onclick attribute for interactive functionality.

**Adding HTML Elements:**

1. **createElement Method:**
   * + **Purpose:** Creates a new HTML element.
     + **Example:**
2. **appendChild Method:**
   * + **Purpose:** Appends a new child element to an existing element.
     + **Example:**
3. **insertBefore Method:**
   * + **Purpose:** Inserts a new element before a specified existing element.
     + **Example:**
4. **innerHTML Property:**
   * + **Purpose:** Sets or gets the HTML content inside an element.
     + **Example:**
5. **insertAdjacentHTML Method:**
   * + **Purpose:** Inserts HTML into a specified position relative to the element.
     + **Example:**

**Deleting HTML Elements:**

1. **removeChild Method:**
   * + **Purpose:** Removes a child element from its parent.
     + **Example:**
2. **remove Method (Modern Browsers):**
   * + **Purpose:** Removes the element itself.
     + **Example:**
3. **replaceChild Method:**
   * + **Purpose:** Replaces a child element with a new element.
     + **Example:**
4. **innerHTML Property (Setting to an Empty String):**
   * + **Purpose:** Sets the HTML content inside an element to an empty string, effectively removing its content.
     + **Example:**
5. **outerHTML Property:**
   * + **Purpose:** Replaces an element with its HTML content.
     + **Example:**

**Query Selectors**

Query Selectors allows developers to select and manipulate HTML elements in a document using CSS-like syntax. They provide a powerful and flexible way to target specific elements based on various criteria, such as element type, class, ID, or attribute.

Here are some common examples of using Query Selectors:

* **Selecting by Element Type:**
* **Selecting by Class Name:**
* **Selecting by ID:**
* **Selecting by Attribute:**
* **Combining Selectors:**

Query Selectors return either a NodeList (for querySelectorAll) or a single element (for querySelector). NodeList is a collection of nodes, which can be iterated through using methods like forEach.

In summary, Query Selectors provide a concise and versatile way to interact with HTML elements in a document, making it easier for developers to manipulate the content and structure of a webpage dynamically.

**DOM Node & Methods**

The DOM (Document Object Model) is a programming interface that represents the structure of a document as a tree of objects, where each object corresponds to a part of the document. A DOM Node is a fundamental interface in the DOM hierarchy, representing a generic node in the tree structure. All elements, attributes, and text content in an HTML or XML document are nodes.

Here are some key points about DOM Nodes and their methods:

**Key Points:**

1. **Node Types:**
   * + Nodes can have different types, such as elements, text nodes, attributes, comments, etc.
     + The nodeType property is used to determine the type of a node.
2. **Hierarchy:**
   * + Nodes are organized in a hierarchical structure, forming a tree.
     + The parentNode property allows you to access the parent node of a given node.
     + The childNodes property provides a NodeList of child nodes.
3. **Traversal:**
   * + The nextSibling and previousSibling properties allow traversal to adjacent nodes.
     + The firstChild and lastChild properties give access to the first and last child nodes.

**Types Of Nodes**

In the DOM (Document Object Model), nodes represent different parts of an HTML or XML document, forming a tree structure. There are various types of nodes, each serving a specific purpose. Here are the common types of nodes in the DOM:

1. **Element Nodes:**
   * + **Description:** Represent HTML or XML elements.
     + **Access:** Accessed using methods like getElementById, getElementsByTagName, or querySelector.
     + **Example:** The <div> element is an example of an element node.
2. **Attribute Nodes:**
   * + **Description:** Represent attributes of an HTML or XML element.
     + **Access:** Attributes can be accessed through the attributes property of an element node.
     + **Example:** In this example, src and alt are attribute nodes of the <img> element.
3. **Text Nodes:**
   * + **Description:** Contain the text content within an HTML or XML element.
     + **Access:** Accessed through the textContent or innerText property of an element node.
     + **Example:** The text "This is a text node" is a text node within the <p> element.
4. **Comment Nodes:**
   * + **Description:** Represent comments within the HTML or XML document.
     + **Access:** Accessed through the comment property of a comment node.
     + **Example:** The content within <!-- and --> is a comment node.
5. **Document Node:**
   * + **Description:** Represents the entire document.
     + **Access:** The document node is the entry point for accessing the DOM tree.
     + **Example:** The <html> element serves as the document node in this example.
6. **Document Type Node:**
   * + **Description:** Represents the document type declaration.
     + **Access:** Accessed through the doctype property of the document node.
     + **Example:** The <!DOCTYPE html> declaration is a document type node.

**DOM Events**

DOM events are interactions or occurrences that take place in a web page, such as a user clicking a button, pressing a key, resizing the browser window, or the content of an input field changing. The HTML DOM (Document Object Model) allows JavaScript to respond to these events, enabling developers to create interactive and dynamic web applications. Here's an overview of DOM events and how JavaScript can react to them:

**Key Concepts:**

1. **Event Types:**
   * + Events can be triggered by various actions, such as mouse clicks (click), keyboard presses (keydown, keyup), form submissions (submit), document loading (load), and more.
2. **Event Targets:**
   * + Events are associated with specific HTML elements, known as event targets. For example, a click event might be associated with a button, and a change event might be associated with a form input.
3. **Event Handlers:**
   * + JavaScript can respond to events by using event handlers. Event handlers are functions that get executed when a specific event occurs.

**Reacting to Events:**

1. **Inline Event Handlers:**
   * + You can define event handlers directly within HTML elements using inline attributes like onclick, onmouseover, etc.
2. **DOM Level 0 Event Handling:**
   * + You can assign event handlers directly to JavaScript properties of DOM elements.
3. **DOM Level 2 Event Handling:**
   * + The addEventListener method is used to attach event handlers to elements. This method provides more flexibility and allows multiple handlers for the same event.
4. **Event Object:**
   * + Event handlers typically receive an event object that provides information about the event, such as the target element, mouse coordinates, key codes, etc.

**Common Events:**

1. **Click Event:**
   * + Triggered when a mouse button is clicked.
2. **Keydown and Keyup Events:**
   * + Fired when a key on the keyboard is pressed or released.
3. **Submit Event:**
   * + Triggered when a form is submitted.
4. **Change Event:**
   * + Fired when the value of an input field changes.
5. **Load Event:**
   * + Occurs when a resource (like an image or script) and the entire page have finished loading.

**Example:**

In this example, a click event handler is attached to a button using the addEventListener method. When the button is clicked, an alert is displayed.

Understanding DOM events and how to handle them is crucial for creating interactive and responsive web applications. Developers use events to capture user actions and trigger appropriate JavaScript functionality in response.

**The onload and onunload functions:**

The onload and onunload events are part of the HTML DOM (Document Object Model) and are used to execute JavaScript code when a document or a page finishes loading (onload) or unloading (onunload). These events are commonly used to perform actions when a user enters or leaves a webpage.

**onload Event:**

The onload event is triggered when a document or a webpage has finished loading. This event is often used to ensure that all resources, such as images and scripts, have been fully loaded before executing specific JavaScript code.

Example:

In this example, the onload event is used to display an alert when the page has finished loading.

**onunload Event:**

The onunload event is triggered just before a document or a webpage is about to be unloaded, such as when the user navigates away from the page or closes the browser tab. This event is often used to perform cleanup tasks or prompt the user for confirmation before leaving the page.

Example:

In this example, the onunload event is used to display an alert just before the page is unloaded.

These events play a crucial role in managing the lifecycle of a web page and allow developers to execute code at specific points during the page's existence.

**DOM Event Listeners**

DOM Event Listeners provide a more flexible and powerful way to handle events compared to traditional event attributes (e.g., onclick). Event Listeners allow you to attach multiple event handlers to a single event, making your code more modular and easier to maintain.

**Using addEventListener:**

The addEventListener method is used to attach an event listener to an HTML element. It takes three parameters: the event type, the function to be executed when the event occurs, and an optional third parameter indicating whether the event should be captured during the event propagation phase.

**Syntax:**

* **eventType**: A string representing the type of event (e.g., "click", "keydown", "change").
* **eventHandler**: A function that will be called when the event occurs.
* **useCapture**: (Optional) A boolean value indicating whether to use the capturing phase (true) or the bubbling phase (false, default).

**Example of Multiple Event Listeners:**

Here's a code snippet demonstrating the use of multiple event listeners on a button. In this example, we have a button that changes its color and displays a message when clicked, and it resets to its default state when the mouse leaves it:

In this example:

* Clicking the button changes its color to green and triggers an alert.
* Hovering over the button changes its color to yellow.
* Moving the mouse away from the button resets its color to the default state.

Using multiple event listeners allows you to handle different aspects of user interaction separately, promoting cleaner and more organized code.

**Event Bubbling & Event Capturing**

Event bubbling and event capturing are two phases of event propagation in the HTML DOM. When an event occurs on an HTML element, it goes through these two phases:

1. **Event Capturing (Capture Phase):**
   * + In this phase, the event travels from the root of the DOM tree to the target element.
     + Event handlers attached with useCapture set to true are triggered during this phase.
2. **Event Bubbling (Bubbling Phase):**
   * + In this phase, the event travels from the target element back up to the root of the DOM tree.
     + Event handlers attached without specifying useCapture or with useCapture set to false are triggered during this phase.

**Example of Event Capturing:**

In the following example, we have a nested set of div elements, and we attach event listeners to the document capturing phase (useCapture set to true). When you click on the innermost div, you'll see that the event handlers for the capturing phase are triggered from the root to the target:

When you click on the "Inner" div, you'll see in the console that the capturing phase event handlers are triggered in the order: Outer Capturing, Middle Capturing, Inner Capturing.

**Example of Event Bubbling:**

In this example, event listeners are attached without specifying useCapture or with useCapture set to false. When you click on the innermost div, the event handlers are triggered in the bubbling phase from the target back up to the root:

When you click on the "Inner" div, you'll see in the console that the bubbling phase event handlers are triggered in the order: Inner Bubbling, Middle Bubbling, Outer Bubbling.

In practice, event bubbling is more commonly used, and the useCapture parameter is often omitted or set to false when attaching event listeners. Event capturing is less commonly used and is mainly applicable in specific scenarios where capturing is explicitly needed.