A black background with white text

Description automatically generated

**Introduction to DOM**

In this lecture, Harkirat lays a solid foundation for the upcoming Frontend Lectures, covering crucial topics such as ECMA script, DOM manipulation, methods for backend interaction, and the importance of debouncing. This session serves as a foundation for a deeper understanding of front-end development principles.

[Introduction to DOM](https://app.100xdevs.com/courses/3/44/160#e0acb196c0af45aaaa5ad2c7a5a29922)[ECMA Scripts](https://app.100xdevs.com/courses/3/44/160#a8390c963c3c474d80e0980b17bf0386)[Auxiliary APIs](https://app.100xdevs.com/courses/3/44/160#9406c8ce26594a49ae1a43f4cdc1d862)[Understanding Document](https://app.100xdevs.com/courses/3/44/160#30e01aeed4b34a868189bd394d3be5db)[Manipulating HTML with JS](https://app.100xdevs.com/courses/3/44/160#50d26babc8f94dfab4723b1d572829ca)[Understanding Rendering Via DOM](https://app.100xdevs.com/courses/3/44/160#49818fbdcd1e44b9ace297ba3b171cfe)[Classes vs IDs](https://app.100xdevs.com/courses/3/44/160#c9aef703aaa54b1696dd6a80d7e360bc)[Methods to Select Elements](https://app.100xdevs.com/courses/3/44/160#a5ca9e90748441b4bd819853e7805f54)[1. querySelector():](https://app.100xdevs.com/courses/3/44/160#320f2a8ab0cc4da195d3d212fd0fbc05)[2. getElementById():](https://app.100xdevs.com/courses/3/44/160#77e1125c0bbc47d288c82c696b719c8a)[3. getElementsByClassName():](https://app.100xdevs.com/courses/3/44/160#857d370e7873451da8f1901780034815)[Communicating to the Backend Server](https://app.100xdevs.com/courses/3/44/160#3af8f28dc8b2431484f7d67d48bdcc10)[The onInput() Function](https://app.100xdevs.com/courses/3/44/160#80e709a62e37491dbb64099af31a6124)[Understanding Debouncing](https://app.100xdevs.com/courses/3/44/160#3c9b8764445f47a29dbb4b50d75ed9b9)[Throttling vs Rate Limiting](https://app.100xdevs.com/courses/3/44/160#db6df98ca5ff4d72ac2e08edf59aca56)[Throttling:](https://app.100xdevs.com/courses/3/44/160#2b8ffc4c8e9549b298428ca9bf5dc10d)[Rate Limiting:](https://app.100xdevs.com/courses/3/44/160#69a7173feec64b9eb8265ba807a3d6f3)[Key Differences:](https://app.100xdevs.com/courses/3/44/160#a3c379089ed64ff98d953dfe4fb96fd2)

**ECMA Scripts**

ECMAScript, often abbreviated as ES, is a scripting language specification that serves as the standard upon which JavaScript is based. Basically, consider them as a set of rules and guidelines that make sure JavaScript behaves in a certain way. It's like a manual that tells developers what features JavaScript should have and how it should work.

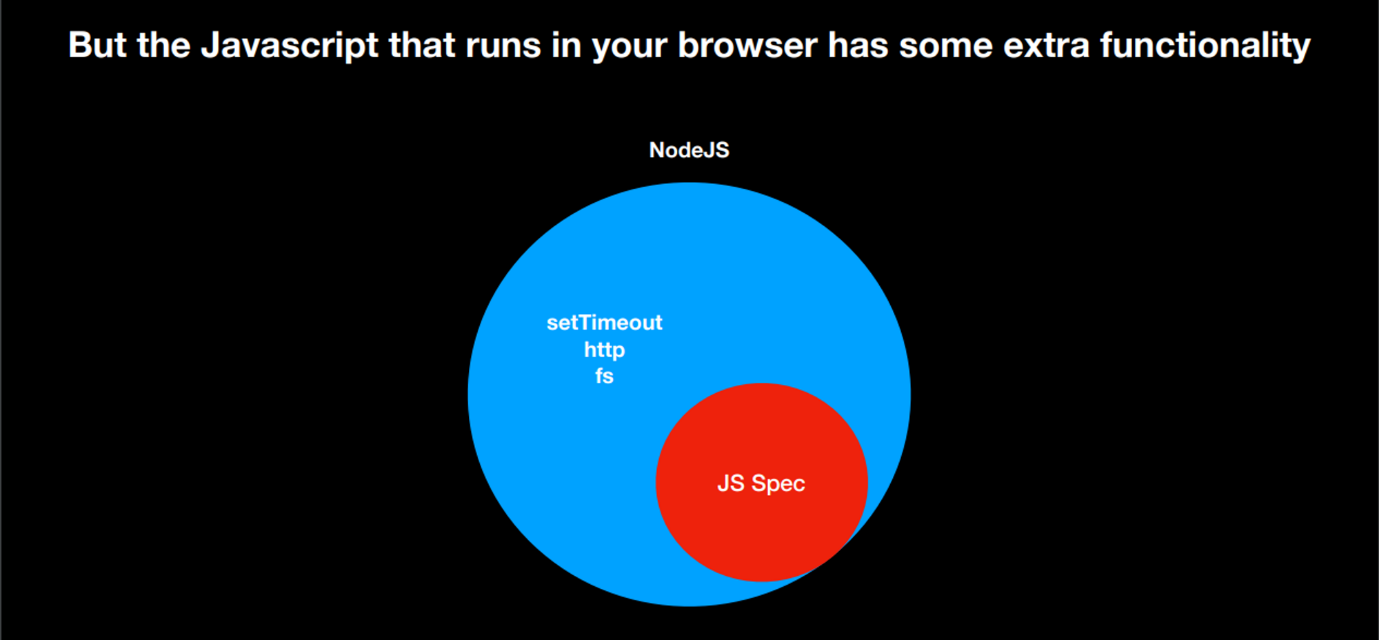
**Auxiliary APIs**

Auxiliary APIs, in the context of web development, refer to additional interfaces and functionalities provided by browsers or runtime environments beyond the core JavaScript language (as specified by ECMAScript). These APIs extend the capabilities of JavaScript, enabling developers to interact with various aspects of the browser environment or perform tasks that go beyond the language's basic features.

Here are some examples:

1. **Node.js APIs:**

* In the context of server-side JavaScript using Node.js, there are APIs specific to Node.js that provide access to the file system, networking, and other server-related functionalities.

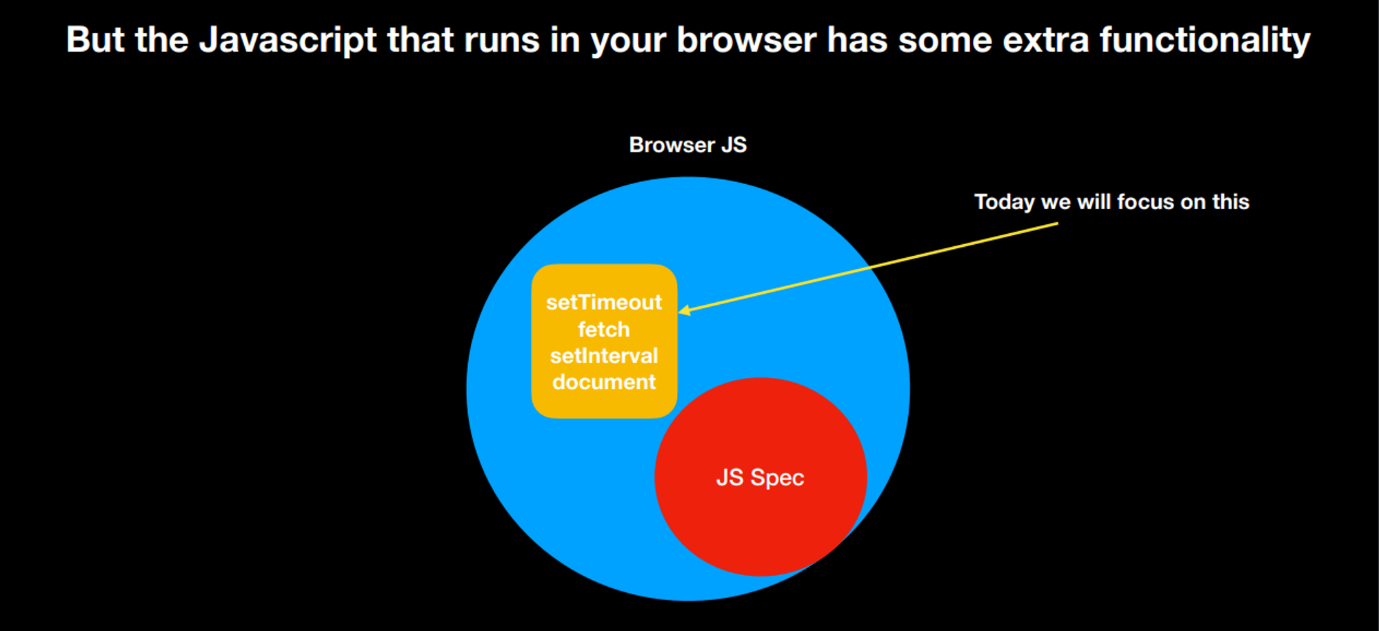


1. **Third-Party APIs:**

* APIs provided by external services or libraries that developers can use to enhance their applications. Examples include Google Maps API, Twitter API, or any other API that allows integration with external services.

1. **Web APIs:**
   * + These are browser-specific APIs that provide additional functionality to JavaScript for interacting with the browser environment. Examples include the DOM (Document Object Model), Fetch API for making network requests, and the Web Storage API for local storage.

These APIs are not part of the JavaScript language specification (ECMAScript) but are essential for building web applications, interacting with external services, and handling server-side operations. They extend the capabilities of JavaScript in specific environments.



Further, in these topics of interest, we have already covered seTimeout, fetch and setInterval. Thus, our major focus today will be on understanding the Document.

**Understanding Document**

In JavaScript, the Document refers to the root object of the DOM.

The DOM (Document Object Model) API is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as a tree of objects; each object represents a part of the page.

The **document** object provides an entry point to the DOM, and it represents the entire HTML or XML document. Developers can use methods and properties provided by the **document** object to interact with and manipulate the content of a web page dynamically.

**Manipulating HTML with JS**

The DOM (Document Object Model) allows JavaScript to manipulate the HTML of a web page.

Imagine the DOM as a tree-like structure that represents your HTML document. Each element in your HTML, like buttons, paragraphs, and images, is a part of this tree. JavaScript can interact with this tree, changing, adding, or removing elements. It's like giving JavaScript the power to update what you see on a webpage.

**Code Example:** Let's create a simple button that, when clicked, changes the content of a paragraph:

In this example:

* We have an HTML file with a button (simpleButton) and an empty paragraph (output).
* JavaScript code at the bottom gets references to these elements using document.getElementById.
* An event listener is added to the button. When clicked, the event listener function is triggered.
* Inside the function, we use textContent to change the content of the paragraph, making something appear on the page.

When you open this HTML file in a browser, clicking the button will make the paragraph magically display a message. This showcases how the DOM allows JavaScript to interact with and modify the content of a web page dynamically.

**Understanding Rendering Via DOM**

Let's create a simple HTML page with two input fields, a button, and JavaScript code to calculate the sum and render it when the button is clicked.

**Explanation:**

1. We have two input fields (num1 and num2), a button (calculateButton), and an empty paragraph (sumResult) in our HTML.
2. JavaScript code at the bottom gets references to these elements using document.getElementById.
3. An event listener is added to the button. When clicked, the event listener function is triggered.
4. Inside the function, it retrieves the values from the input fields, calculates the sum, and displays the result in the paragraph.

Now, when you enter numbers into the input fields and click the "Calculate Sum" button, it will compute the sum and display the result on the page.

**Classes vs IDs**

**Classes:**

* **Definition:** Used to group multiple HTML elements together.
* **Syntax (HTML):** <element class="class-name">Content</element>
* **Syntax (CSS):** .class-name { /\* styles \*/ }
* **Usage:** Can be shared by multiple elements; an element can have multiple classes.

**IDs:**

* **Definition:** Used to uniquely identify a specific HTML element.
* **Syntax (HTML):** <element id="unique-id">Content</element>
* **Syntax (CSS):** #unique-id { /\* styles \*/ }
* **Usage:** Must be unique within a page; often used for styling or JavaScript interaction.

**Differences:**

* **Uniqueness:**
  + Classes can be shared; IDs must be unique within a page.
* **Application:**
  + Classes are for styling multiple elements.
  + IDs are for styling a specific element or targeting with JavaScript.

In short, classes group elements, allowing shared styles, while IDs uniquely identify elements, often for specific styling or JavaScript interactions.

**Methods to Select Elements**

**1. querySelector():**

* **Definition:** querySelector() is a method that selects the first element that matches a specified CSS selector.
* **Syntax:** document.querySelector('selector')
* **Example:**
  + In this example, querySelector('#example') selects the element with the ID "example," and the text color is changed to blue.

**2. getElementById():**

* **Definition:** getElementById() is a method that selects a single element by its ID attribute.
* **Syntax:** document.getElementById('id')
* **Example:**
  + In this example, getElementById('example') selects the element with the ID "example," and the text color is changed to red.

**3. getElementsByClassName():**

* **Definition:** getElementsByClassName() is a method that selects all elements with a specific class name.
* **Syntax:** document.getElementsByClassName('class')
* **Example:**
  + In this example, getElementsByClassName('highlight') selects all elements with the class "highlight," and their font weight is changed to bold.

**Relationship:**

* All three methods are used to select and manipulate HTML elements.
* getElementById() is specifically for selecting by ID.
* getElementsByClassName() selects by class name, but it returns a collection of elements.
* querySelector() is more flexible as it can select by any valid CSS selector and returns the first matching element.

**Example Using All Three:**

In this example, all three methods are used to select elements by ID, class name, and a general CSS selector, respectively. Each selected element is then styled accordingly.

**Communicating to the Backend Server**

Let's try to understand the process of connecting the frontend and backend through a small example. In this scenario, the backend is already hosted on the internet. This implies that in the frontend, we only need to make an API call to the provided link to interact with the backend.

**Explanation:**

1. **Input Fields:** Two input fields (firstNumber and secondNumber) are provided for users to enter numeric values.
2. **Button:** The "Calculate sum" button is associated with the populateDiv() function, which will be triggered when the button is clicked.
3. **JavaScript Function (populateDiv):**
   * + Retrieves the values entered by the user from the input fields.
     + Uses the fetch API to make a GET request to the specified backend server (https://sum-server.100xdevs.com/sum) with the provided parameters a and b.
     + Processes the response:
       - Converts the response to text.
       - Updates the content of the finalSum div with the calculated sum.
4. **Display Area:** The result is displayed in the finalSum div.
5. **Async Function (populateDiv2):**

* The **async** keyword is added before the function declaration, indicating that the function will handle asynchronous operations using **await**.
* **await** is used with the **fetch** call, allowing the code to pause and wait for the promise to resolve before moving on to the next line of code.
* The response is obtained from the fetch call using **await**, and then the response text is retrieved similarly.

By using **async/await**, the code becomes more concise and easier to read, making it a preferable alternative for handling promises. Both **populateDiv()** and **populateDiv2()** perform the same functionality, but the latter takes advantage of modern JavaScript syntax for better code organization.

**The onInput() Function**

The **onInput** function is an event handler in JavaScript that gets executed when the value of an input field is changed by the user. This event is triggered dynamically as the user types or modifies the content within the input field. The **onInput** event is commonly used to perform actions in real-time as the user interacts with the input element.

**Understanding Debouncing**

Debouncing is a programming practice used to ensure that time-consuming tasks do not fire so often, making them more efficient. In the context of onInput events, debouncing is often applied to delay the execution of certain actions (e.g., sending requests) until after a user has stopped typing for a specific duration.

**Implementation:** The following example demonstrates debouncing in the onInput event to delay the execution of a function that sends a request based on user input.

**Explanation:**

* The debounce function is a generic debounce implementation that takes a function (func) and a delay time (delay).
* Inside the debounce function, a timeout is set to delay the execution of the provided function (func) by the specified delay time (delay).
* The handleInput function is the actual function to be executed when the onInput event occurs. It simulates sending a request (e.g., an AJAX call) based on user input.

**How it works:**

* When a user types in the input field, the onInput event triggers the debounce function.
* The debounce function sets a timeout, and if the user continues typing within the specified delay time, the previous timeout is cleared, and a new one is set.
* After the user stops typing for the specified delay, the handleInput function is executed.

This ensures that the function associated with the onInput event is not called on every keystroke but rather after the user has stopped typing for a brief moment, reducing unnecessary and potentially resource-intensive calls, such as sending requests.

**Throttling vs Rate Limiting**

**Throttling:**

* **Definition:** Controls the rate at which a specific action is performed.
* **Purpose:** Ensures a smooth user experience, preventing rapid consecutive actions.
* **Implementation:** Limits the frequency of a particular function within a specified time frame.

**Rate Limiting:**

* **Definition:** Controls the number of requests a client can make within a specific time period.
* **Purpose:** Protects server resources, avoids abuse, and maintains fair usage.
* **Implementation:** Typically applied at the server/API level, limiting requests per second or minute.

**Key Differences:**

* Throttling focuses on action frequency; rate limiting focuses on request count.
* Throttling can be applied to various actions; rate limiting is often used at the API level.
* Throttling aims for a smooth user experience; rate limiting protects server resources and enforces fairness.

**Reconcilers & Intro to React**

In this lecture, Harkirat addresses the challenges encountered in vanilla JavaScript while building a Todo application. Focusing on the limitations of manual DOM manipulation and the lack of a centralized statethe discussion sets the context for transitioning to React. The session highlights the pain points faced during development and introduces React's declarative and component-based approach as a solution for more efficient and scalable web development.

[Reconcilers & Intro to React](https://app.100xdevs.com/courses/3/44/161#b9a336e12ca3489e9e1b66f8c1c2c578)[Why React?](https://app.100xdevs.com/courses/3/44/161#aea267e2d41e4161809c480fc1607827)[Todo Application Frontend](https://app.100xdevs.com/courses/3/44/161#b7404fe59efe4abd87c3af46edcc1f76)[The Current Approach faces Several Challenges:](https://app.100xdevs.com/courses/3/44/161#e41df24b65e94473b89e70a1f4d3dcd1)[What do we mean by State?](https://app.100xdevs.com/courses/3/44/161#62678a543a7e419280aaa0e2b7141ced)[Solution With a Blackbox Fn](https://app.100xdevs.com/courses/3/44/161#69bdc232169e45189f4afa4ef1658583)[The updateState() function](https://app.100xdevs.com/courses/3/44/161#650af09b1c484618979cac71af9fe2ad)[Virtual DOM](https://app.100xdevs.com/courses/3/44/161#e85dffd564a74bb8952d5999bba221b9)[Complete Solution of the Todo Application](https://app.100xdevs.com/courses/3/44/161#f9d653617aa54091b959ccb1d6d1781c)[Conclusions](https://app.100xdevs.com/courses/3/44/161#5b15a4e0ddc04e1b952d0f3e8e11d604)[Challenges Faced Throughout](https://app.100xdevs.com/courses/3/44/161#6de11e8102ba4725afea659d3fea7362)[Starting with React](https://app.100xdevs.com/courses/3/44/161#c07be35a394b49c0a1517d5e2fab14b2)[Building our First Application in React](https://app.100xdevs.com/courses/3/44/161#3e977d78122441a4b617c5cd8cf30f87)

**Why React?**

DOM manipulation, in its raw form, poses significant challenges for developers. Constructing dynamic websites using the basic primitives offered by the DOM—such as **document.createElement**, **document.appendChild**, **element.setAttribute**, and **element.children**—can be a complex and labor-intensive process. The inherent difficulty lies in orchestrating intricate interactions and updates within the document structure using these primitive tools.

Recognizing the intricacies involved, React, a JavaScript library, emerged as a powerful solution. React abstracts away much of the manual DOM manipulation complexity, providing developers with a declarative and component-based approach to building user interfaces. This abstraction not only enhances code readability and maintainability but also simplifies the creation of dynamic and interactive web applications.

**However, before jumping directly into React, let's us first try building a simple todo application using vanilla JavaScript. This exercise will serve a dual purpose—it will not only reinforce your JavaScript concepts but also allow you to grasp the problem statement, setting the stage to appreciate the elegance of React even more.**

**Todo Application Frontend**

In today's session, our focus shifts to constructing a straightforward todo application. Unlike the previous day's example where we were overwriting the **innerHTML** to display results of calculateSum, today's task involves a different approach. We aim to append elements instead of overwriting them, creating a more seamless user experience. The goal is to have all entered todos consistently displayed on the screen, eliminating the abrupt resetting of content and ensuring a continuous and fluid representation of the user's input.

First, let's take a comprehensive look at the entire code for building the frontend of a Todo Application in vanilla JavaScript. Later, we will delve deep into each line and component of the code to gain a thorough understanding of its workings.

Now, Let's break down the JavaScript code step by step:

1. **Global Variables:**
   * + globalId is a variable initialized to 1. It is used to assign a unique identifier to each todo item.

1. **markAsDone Function:**

* + - markAsDone is a function that marks a todo as done.
    - It takes the todoId as a parameter, which corresponds to the unique identifier of the todo item.
    - It finds the todo item using document.getElementById(todoId) and then accesses its third child element (index 2) to update its HTML content to "Done!".

1. **createChild Function:**

* + - createChild is a function that generates a new todo item.
    - It takes title, description, and id as parameters.
    - It creates three child elements (firstGrandParent, secondGrandParent, and thirdGrandParent) representing the title, description, and "Mark as done" button, respectively.
    - These child elements are appended to the child div, and the unique identifier (id) is set as the id attribute for the child div.
    - The function returns the generated child div.

1. **addTodo Function:**
   * + addTodo is a function that adds a new todo item to the list.
     + It retrieves the values of the title and description from the corresponding input fields.
     + It then gets the parent container (todos) and appends a new todo item using the createChild function.
     + The globalId is passed as the unique identifier, and it is then incremented for the next todo.

This setup allows users to add todo items with a title, description, and a "Mark as done" button that, when clicked, updates the todo's status to "Done!".

**The Current Approach faces Several Challenges:**

1. **Difficulty in Adding and Removing Elements:** The process of adding and removing elements becomes intricate with the existing setup. Direct manipulation of the DOM for such operations can lead to complex and error-prone code.
2. **Lack of Central State:** The absence of a centralized state management system poses issues. With each function managing its own state, maintaining consistency across the application becomes challenging.
3. **Integration with a Server:** If there is a server where these todos are stored, the current structure lacks a mechanism to seamlessly integrate with it. Handling data from an external server requires a more organized approach.
4. **Mobile App Updates:** Imagine updating a TODO item from a mobile app. When the updated list of TODOs arrives on the frontend, there is no provision for efficiently updating the DOM. The current structure lacks functions for updating or removing existing TODOs.

In essence, while there's an addTodo function, the absence of updateTodo and removeTodo functions limits the flexibility and robustness of the application, especially in scenarios involving dynamic changes and external data sources.

**What do we mean by State?**

When we refer to state in the context of our todo application, we are essentially talking about the current representation of the todo data within the application. In a more structured format, the state might look something like this:

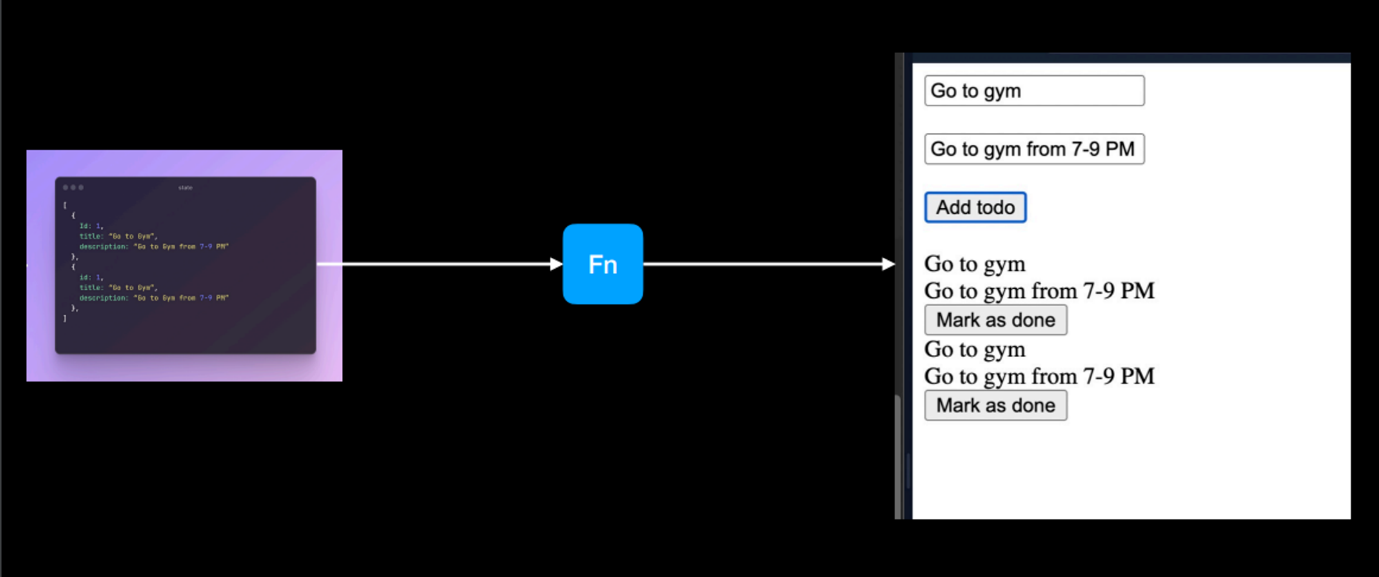
Here, the state is a collection of todo items, each represented as an object with properties such as 'id', 'title', and 'description'. This structured format allows us to organize and manage the current state of the application's data.

In a real-world scenario, as the user interacts with the application—adding, updating, or removing todos—the state dynamically changes to reflect the most recent data. This concept of state management becomes crucial for maintaining a consistent and up-to-date representation of the todo list within the application.

**Solution With a Blackbox Fn**

Now, if we can write a function, that takes this state as an input and creates the output, that would be much more powerful than our original approach.

When we talk about a function that takes a state as input and creates the desired output, we are referring to a function that can dynamically generate the representation of our todo list based on the current state.



Now for a moment let us consider this function Fn — updateState() to be a black box. This boils down our todo application code to:

Explanation:

1. The addTodo function is responsible for adding a new todo to the state.
2. It initializes an empty todoState array to store todos.
3. It retrieves the values of the todo title and description from the corresponding input fields.
4. A new todo object is created with a unique identifier (id) using the globalId variable.
5. The new todo object is pushed to the todoState array.
6. The updateState function or Fn (which is not provided in this code) is called to update the state and render the todos on the screen.
7. The HTML structure consists of input fields for the todo title and description, a button to add a new todo, and a container (div) to display the list of todos.

**The updateState() function**

Now Let us focus on this blackbox Fn — the updateState() function

A Dumb way to constructing the **updateState()** function would be:

1. **Clear the Parent Element:**

Remove all child elements from the parent element.

1. **Repopulate the DOM:**

Call **addTodo()** for each element in the state, effectively re-rendering all todos.

However, a more intelligent solution is to:

* **Avoid Clearing the DOM Upfront:**

Instead of clearing the entire parent element and starting from scratch, maintain the existing DOM elements.

* **Update Based on Changes:**

Update the DOM selectively based on what has changed in the state.

**Question Arises:**

* How does the application determine what has changed?
* Has a todo been marked as complete?
* Has a todo been removed from the backend?

**Solution:** Keep track of the old todos in a variable, essentially creating a "Virtual DOM.

**Virtual DOM**

The concept of a Virtual DOM comes into play when dealing with efficient updates to the actual DOM.

The Virtual DOM is a lightweight copy of the actual DOM. When updates are made to the state of an application, a new Virtual DOM is created with the changes. This Virtual DOM is then compared with the previous Virtual DOM to identify the differences (known as "diffing").

In the context of a todo application:

1. **State Changes:**
   * + If a todo has been marked as complete or removed from the backend, the state of the application changes.
2. **Virtual DOM Comparison:**
   * + The new state is used to create a new Virtual DOM.
     + This new Virtual DOM is compared with the previous Virtual DOM.
3. **Identifying Changes:**
   * + The difference between the new and previous Virtual DOMs is determined.
     + For example, if a todo has been marked as complete, the corresponding element in the Virtual DOM is updated to reflect this change.
4. **Efficient Updates:**
   * + Instead of clearing the entire parent element and re-rendering everything, the Virtual DOM helps identify specifically what has changed.
5. **Selective DOM Manipulation:**
   * + Only the elements that have changed are manipulated in the actual DOM.
     + This process is more efficient than a naive approach of clearing and re-rendering the entire DOM.

By employing a Virtual DOM, React optimizes the process of updating the actual DOM, leading to better performance and a smoother user experience. This mechanism is a key feature of React and contributes to its popularity for building dynamic and responsive web applications.

**Complete Solution of the Todo Application**

Now, let us have a look at our final complete code and draw conclusions out of this exercise:

**Conclusions**

To build a dynamic frontend website in the easiest way possible we need 3 things.

A function to :

1. Update a state variable here addTodo()
2. Delegate the task of figuring out difference in the DOM tree here updateStateTodo()
3. Tell the hefty function how to add, update and remove elements here updateTodo() and removeTodo()

Now, creating a function to Update a state variable addTodo() and a function to Update and Remove elements updateTodo() removeTodo() is the job of a frontend developer. While this task of a hefty function to figure out the difference in the DOM tree is done by React.

**Challenges Faced Throughout**

As we transition into React, it's crucial to reflect on the challenges faced while building the Todo application using vanilla JavaScript.

1. The manual manipulation of the DOM, appending and removing elements,
2. lack of a centralized state, and
3. the absence of efficient methods for handling updates posed significant hurdles.

The approach of overwriting the entire DOM or clearing elements upfront proved cumbersome, especially when dealing with dynamic changes such as marking a Todo as complete or removing items from the backend. React, as a declarative and component-based library, addresses these pain points directly.

**Starting with React**

By abstracting away the complexities of DOM manipulation, React provides a more efficient and maintainable way to handle state changes, updates, and dynamic rendering. The transition to React promises not only a cleaner and more organized code structure but also a streamlined approach to building interactive web applications. Let's delve into React's features and explore how it revolutionizes frontend development, simplifying the implementation of functionalities like updating Todos and managing state seamlessly.

**Building our First Application in React**

Let’s start a new React project using Vite, you can follow these steps:

1. Open your terminal and run the following command to create a new Vite project:
2. Follow the prompts to set up your project. You can choose the default settings for now.
3. Once the project is created, navigate to the project directory using:
4. Open the project in your code editor.

Now, let's build a simple React component with a button that increases the count dynamically on clicks using useState. Update the src/App.jsx file with the following code:

Explanation:

* We import React and the useState hook from the 'react' package.
* Inside the App component, we declare a state variable count and a function setCount to update it using useState(0). The initial value of count is set to 0.
* We define a function handleButtonClick that increases the count by 1 when the button is clicked. We use the setCount function to update the state.
* The JSX returned by the component displays the current count and a button. The onClick attribute of the button is set to the handleButtonClick function.

Now, save the file and run your project using:

Visit http://localhost:3000 in your browser, and you should see your React app with a button that dynamically increases the count on each click.