SetTimeOut

setTimeout() is a function serviced globally by the window object provided by the user's browser. It allows users to execute callbacks after a period of time expressed in milliseconds.

Syntax

setTimeout() is capable of receiving multiple parameters where the first is a callback function. The second parameter receives a number that represents the time in milliseconds (1s = 1000ms), which defines the time needed for the callback to execute. The third parameter onwards will be the parameters that the callback function would take in case arguments are defined within the callback.

Below are some examples of how setTimeout() is composed:

setTimeout(callback)

setTimeout(callback, delay)

setTimeout(callback, delay, param)

The setInterval() Method

The setInterval() method repeats a given function at every given time-interval.

window.setInterval(*function*, *milliseconds*);

The window.setInterval() method can be written without the window prefix.

The first parameter is the function to be executed.

The second parameter indicates the length of the time-interval between each execution.

This example executes a function called "myTimer" once every second (like a digital watch).

How to Stop the Execution?

The clearInterval() method stops the executions of the function specified in the setInterval() method.

window.clearInterval(*timerVariable*)

The window.clearInterval() method can be written without the window prefix.

The clearInterval() method uses the variable returned from setInterval():

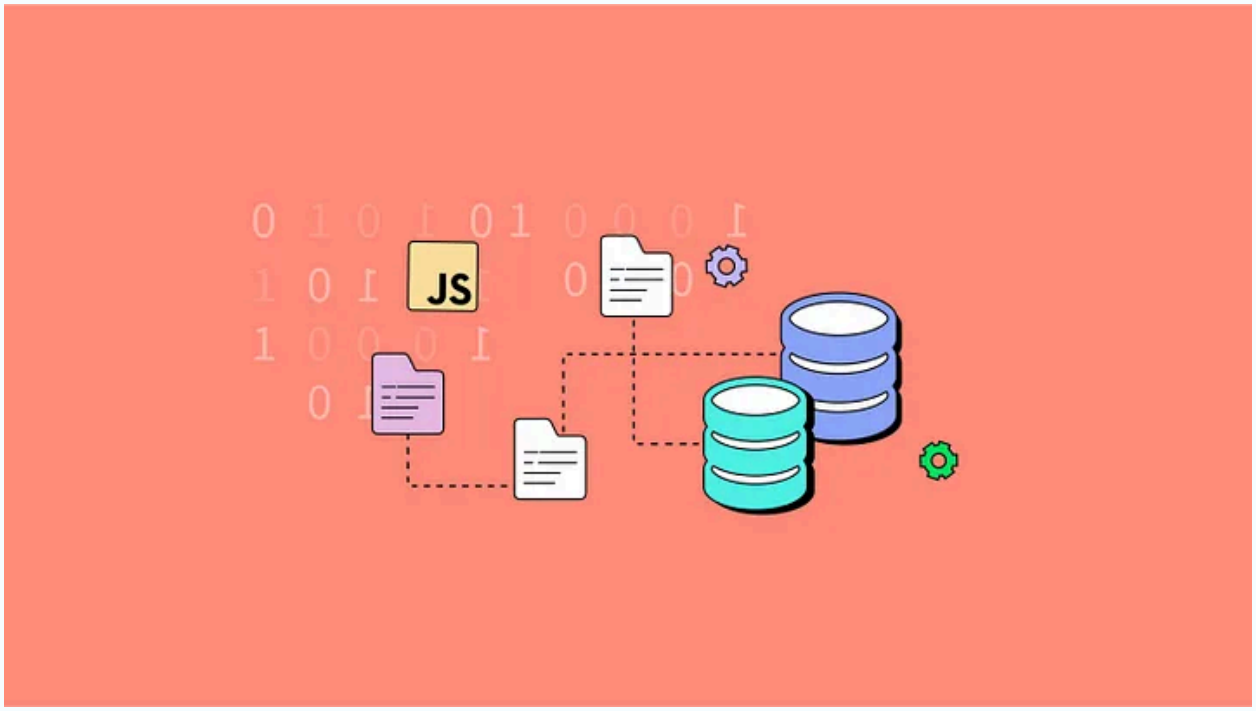
let myVar = setInterval(*function*, *milliseconds*);

clearInterval(myVar);

The JavaScript setTimeout() method is a built-in method that allows you to time the execution of a certain function . You need to pass the amount of time to wait for in milliseconds , which means to wait for one second, you need to pass one thousand milliseconds.

To cancel a setTimeout() method from running, you need to use the clearTimeout() method, passing the ID value returned when you call the setTimeout() method.

Locale Storage vs Session Storage



What are Local Storage and Session Storage?

Local storage and session storage are both mechanisms provided by modern web browsers to allow web applications to store data locally within the user’s browser.

Local Storage:

- Local storage is designed to persist data across browser sessions.
- Data stored in local storage remains available even after the browser is closed and reopened.
- It is typically used for storing long-term, or “permanent” data such as user preferences, cached data, or application state.

Session Storage:

- Session storage, on the other hand, is intended to store data for the duration of the page session.
- Data stored in session storage is accessible only as long as the browser tab or window is open. Once the tab or window is closed, the data is cleared.

- Session storage is commonly used for temporary data that is only relevant to the current browsing session, such as form data or navigation history within a single tab.

Key Differences

1. Persistence:

- The primary distinction between local storage and session storage lies in their persistence.
- Local storage persists indefinitely until explicitly cleared by the user or the application.
- Session storage, on the other hand, is temporary and is cleared when the browsing session ends.

2. Scope:

- Both local storage and session storage have a similar API and can store data in key-value pairs.

- However, the scope of their data storage differs.
- Local storage is scoped to the origin of the website, meaning that data stored in local storage is accessible across all tabs and windows from the same origin.
- Session storage, on the other hand, is scoped to the individual browser tab or window. Data stored in session storage is not accessible to other tabs or windows.

3. Use Cases:

- Local storage is suitable for storing user preferences, application settings, and other long-term data that needs to persist across multiple sessions.
- Session storage is ideal for storing temporary data that is relevant only to the current browsing session, such as form data, shopping cart items, or transient application state.

4. Security Considerations:

- Since data stored in local storage persists across sessions and is accessible across all tabs and windows from the same origin, developers must be cautious about storing sensitive information, such as authentication tokens or personal data, in local storage.
- Session storage, with its shorter lifespan and tab/window scope, provides a more secure option for storing transient data, reducing the risk of data exposure or leakage.

Deciding whether to use local storage or session storage depends on the specific requirements and characteristics of your web application. Here’s a breakdown of scenarios where each storage mechanism is most appropriate:

Use Local Storage When:

1. Persistent Data Storage: Use local storage for data that needs to persist across browser sessions. This could include user preferences, application settings, or cached data that should remain available even after the user closes and reopens the browser.
2. Long-Term User Data: If your application requires storing user-generated content or long-term user data such as profiles, saved preferences, or history, local storage is the preferred choice. It ensures that users' data remains accessible across multiple sessions, enhancing the overall user experience.
3. Cross-Tab/Window Accessibility: Local storage is accessible across all tabs and windows from the same origin. If your application needs data to be shared and

synchronized seamlessly across different browser instances, local storage provides a convenient solution.

4. Large Data Sets: Local storage is suitable for storing larger data sets, as it offers a relatively generous storage limit (usually several megabytes per origin, depending on the browser).

Use Session Storage When:

1. Temporary Data Storage: Session storage is ideal for storing temporary data that is relevant only for the duration of the browsing session. This could include form data, temporary application state, or transient user interactions.
2. Tab-Specific Data: Since session storage is scoped to individual browser tabs or windows, it’s useful for scenarios where data needs to be isolated to a specific

tab. For example, maintaining separate shopping carts or form submissions in different tabs.

3. Enhanced Security Requirements: If your application deals with sensitive data, such as authentication tokens or temporary session identifiers, session storage offers better security. Since session storage data is cleared when the browser tab or window is closed, it reduces the risk of data exposure or unauthorized access.
4. Lightweight Data Storage: Session storage has a smaller storage capacity compared to local storage, typically limited to a few megabytes per tab. It's suitable for storing smaller amounts of data that are required for short-term operations.

Considerations:

- Data Sensitivity: Always consider the sensitivity of the data you're storing. Avoid storing sensitive information, such as passwords or personal data, in client-side storage mechanisms whenever possible, or employ encryption techniques to enhance security.
- Browser Compatibility: While local storage and session storage are supported by most modern browsers, it's essential to verify compatibility and potential limitations, especially when targeting older browser versions or specific environments.
- User Experience: Ultimately, the choice between local storage and session storage should prioritize the user experience. Ensure that your storage mechanism aligns with the application's functionality, data persistence requirements, and security considerations to deliver a

seamless and secure browsing experience for your users.

Conclusion

In summary, local storage and session storage are both valuable tools for client-side data storage in web applications, each serving distinct purposes based on their persistence, scope, and use cases.

When deciding between local storage and session storage, developers should consider the longevity of the data, its scope of accessibility, and any security implications associated with storing sensitive information. By understanding the differences between these two storage mechanisms, developers can make informed decisions to ensure efficient and secure data management in their web applications.

Cookies in JS

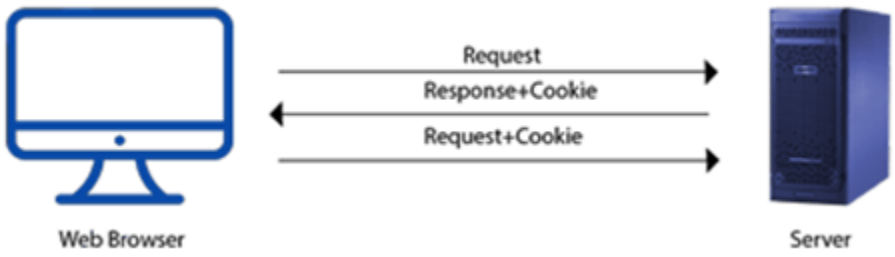
JavaScript Cookies

A cookie is an amount of information that persists between a server-side and a client-side. A web browser stores this information at the time of browsing.

A cookie contains the information as a string generally in the form of a name-value pair separated by semi-colons. It maintains the state of a user and remembers the user's information among all the web pages.

How Cookies Works?

- When a user sends a request to the server, then each of that request is treated as a new request sent by the different user.
- So, to recognize the old user, we need to add the cookie with the response from the server.
- browser at the client-side.
- Now, whenever a user sends a request to the server, the cookie is added with that request automatically. Due to the cookie, the server recognizes the users.



How to create a Cookie in JavaScript?

In JavaScript, we can create, read, update and delete a cookie by using document.cookie property.