

Web Storage API Complete Guide

Table of Contents

1. [Web Storage API Overview](#)
 2. [Local Storage](#)
 3. [Session Storage](#)
 4. [Storage Methods and Operations](#)
 5. [Different Ways to Access Storage](#)
 6. [Working with Complex Data](#)
 7. [Storage Events](#)
 8. [Best Practices](#)
 9. [Common Use Cases](#)
 10. [Security Considerations](#)
-

Web Storage API Overview

What is Web Storage?

The **Web Storage API** provides mechanisms for browsers to store key-value pairs in a web browser. It's much more intuitive and has a larger storage capacity than cookies.

Two Types of Web Storage

1. Local Storage (`localStorage`)

- Data persists permanently (until explicitly deleted)
- Shared across all tabs and windows from same origin
- Survives browser restarts
- ~5-10MB storage limit (browser dependent)

2. Session Storage (`sessionStorage`)

- Data exists only for the session (until tab/window closes)
- Isolated per tab/window
- Cleared when tab/window closes
- ~5-10MB storage limit (browser dependent)

Why Use Web Storage?

Advantages over Cookies:

- **Larger capacity:** 5-10MB vs 4KB for cookies
- **Not sent to server:** Reduces bandwidth usage
- **Simpler API:** Easy to use key-value interface
- **Better performance:** No HTTP header overhead
- **Client-side only:** More secure for client-side data

Common Use Cases:

- User preferences (theme, language)
 - Shopping cart data
 - Form data backup
 - Authentication tokens
 - Application state
 - User settings
 - Recently viewed items
 - Cached data
-

Local Storage

What is Local Storage?

Local Storage stores data with no expiration date. The data persists even after the browser is closed and reopened.

Characteristics of Local Storage

Persistence:

```
// Data saved today...
localStorage.setItem("username", "John");

// ...is still available tomorrow, next week, next year
// Until explicitly deleted or browser data is cleared
```

Scope:

- **Same origin only:** Only accessible from same protocol, domain, and port

- **Shared across tabs:** All tabs/windows from same origin access same data
- **Survives browser restart:** Data remains after closing browser

Storage Limit:

- Typically 5-10MB per origin
- Varies by browser:
 - Chrome: ~10MB
 - Firefox: ~10MB
 - Safari: ~5MB
 - Edge: ~10MB

Basic Local Storage Operations

From Your Code - Saving Data:

```
function saveData() {
  // Method 1: Direct property assignment
  localStorage.email = "mona@gmail.com";

  // Method 2: Bracket notation
  localStorage["username"] = "mona";

  // Method 3: setItem method (RECOMMENDED)
  localStorage.setItem("favColor", "#000");
}
```

All three methods work identically, but `setItem()` is recommended for:

- Consistency with `getItem()`
- Better code clarity
- Avoiding naming conflicts with Storage object properties

From Your Code - Retrieving Data:

```
function getData() {
  // Method 1: Bracket notation
  console.log(localStorage["email"]); // "mona@gmail.com"

  // Method 2: Direct property access
  console.log(localStorage.username); // "mona"

  // Method 3: getItem method (RECOMMENDED)
```

```

    console.log(localStorage.getItem("favColor")); // "#000"

    // Accessing non-existent keys
    console.log(localStorage.getItem("pass")); // null
    console.log(localStorage.pass); // undefined
}

```

Important Difference:

```

// getItem() returns null if key doesn't exist
localStorage.getItem("nonexistent"); // null

// Direct access returns undefined
localStorage.nonexistent; // undefined

```

From Your Code - Removing Data:

```

function removeItem() {
    // Remove single item
    localStorage.removeItem("email");

    // After removal
    console.log(localStorage.getItem("email")); // null
}

```

From Your Code - Clearing All Data:

```

function removeAll() {
    // Remove ALL items from localStorage
    localStorage.clear();

    // After clearing
    console.log(localStorage.length); // 0
}

```

Complete Example

HTML:

```

<button onclick="saveData()">Save Data</button>
<button onclick="getData()">Get Data</button>

```

```
<button onclick="removeItem()">Remove Item</button>
<button onclick="removeAll()">Remove All</button>
```

JavaScript:

```
function saveData() {
    localStorage.setItem("email", "mona@gmail.com");
    localStorage.setItem("username", "mona");
    localStorage.setItem("favColor", "#000");
    console.log("Data saved!");
}

function getData() {
    const email = localStorage.getItem("email");
    const username = localStorage.getItem("username");
    const favColor = localStorage.getItem("favColor");

    console.log("Email:", email);
    console.log("Username:", username);
    console.log("Favorite Color:", favColor);
}

function removeItem() {
    localStorage.removeItem("email");
    console.log("Email removed!");
}

function removeAll() {
    localStorage.clear();
    console.log("All data cleared!");
}
```

Session Storage

What is Session Storage?

Session Storage is similar to Local Storage, but data is only available for the duration of the page session.

Characteristics of Session Storage

Session Lifetime:

```
// Data saved in this tab...
sessionStorage.setItem("username", "John");

// ...is only available in THIS tab
// ...and disappears when tab is closed
```

Scope:

- **Per tab/window:** Each tab has its own separate sessionStorage
- **Same origin only:** Only accessible from same protocol, domain, and port
- **Session duration:** Cleared when tab/window closes
- **Page reload:** Survives page reloads and restores

Key Differences from localStorage:

Feature	localStorage	sessionStorage
Lifetime	Permanent	Until tab closes
Scope	All tabs/windows	Current tab only
Survives browser restart	Yes	No
Survives page reload	Yes	Yes
Survives tab close	Yes	No

Basic Session Storage Operations

From Your Code - Same API as localStorage:

Saving Data:

```
function saveData() {
    // Method 1: Direct property
    sessionStorage.email = "mona@gmail.com";

    // Method 2: Bracket notation
    sessionStorage["username"] = "mona";

    // Method 3: setItem (RECOMMENDED)
    sessionStorage.setItem("favColor", "#000");
}
```

Retrieving Data:

```

function getData() {
    // Method 1: Bracket notation
    console.log(sessionStorage["email"]);

    // Method 2: Direct property
    console.log(sessionStorage.username);

    // Method 3: getItem (RECOMMENDED)
    console.log(sessionStorage.getItem("favColor"));

    // Non-existent keys
    console.log(sessionStorage.getItem("pass")); // null
    console.log(sessionStorage.pass); // undefined
}

```

Removing Data:

```

function removeItem() {
    sessionStorage.removeItem("email");
}

function removeAll() {
    sessionStorage.clear();
}

```

When Session Storage is Cleared

Cleared When:

- Tab/window is closed
- User manually clears browser data

NOT Cleared When:

- Page is reloaded (F5)
- Page navigates to another page in same tab
- Browser crashes and restores session (Chrome, Firefox)

Example - Multi-step Form:

```

// Step 1: Save form data
function saveStep1() {
    const name = document.getElementById("name").value;
    const email = document.getElementById("email").value;
}

```

```

    sessionStorage.setItem("step1_name", name);
    sessionStorage.setItem("step1_email", email);

    // Navigate to step 2
    window.location.href = "step2.html";
}

// Step 2: Restore data from step 1
function loadStep1Data() {
    const name = sessionStorage.getItem("step1_name");
    const email = sessionStorage.getItem("step1_email");

    console.log("Previous step data:", name, email);
}

// Final step: Submit and clear
function submitForm() {
    // Get all data
    const step1Data = {
        name: sessionStorage.getItem("step1_name"),
        email: sessionStorage.getItem("step1_email")
    };

    // Submit to server
    submitToServer(step1Data);

    // Clear session data
    sessionStorage.clear();
}

```

Storage Methods and Operations

Complete API Reference

Both `localStorage` and `sessionStorage` share the same methods.

1. `setItem(key, value)`

Syntax:

```
storage.setItem(key, value);
```

Purpose: Store a key-value pair.

Parameters:

- `key` : String key name
- `value` : String value (non-strings are converted to strings)

Examples:

```
// String value
localStorage.setItem("username", "John");

// Number (automatically converted to string)
localStorage.setItem("age", 25);
console.log(typeof localStorage.getItem("age")); // "string"

// Boolean (converted to string)
localStorage.setItem("isLoggedIn", true);
console.log(localStorage.getItem("isLoggedIn")); // "true" (string)

// Overwriting existing key
localStorage.setItem("username", "Jane"); // Replaces "John"
```

Important Note:

Storage only accepts strings. Non-string values are automatically converted using `toString()`.

2. `getItem(key)`

Syntax:

```
storage.getItem(key);
```

Purpose: Retrieve value for a key.

Returns:

- Value as string if key exists
- `null` if key doesn't exist

Examples:

```
localStorage.setItem("username", "John");

// Get existing key
```

```
console.log(localStorage.getItem("username")); // "John"

// Get non-existent key
console.log(localStorage.getItem("password")); // null

// Check if key exists
if (localStorage.getItem("username") !== null) {
    console.log("Username exists");
}
```

3. removeItem(key)

Syntax:

```
storage.removeItem(key);
```

Purpose: Remove a specific key-value pair.

Examples:

```
localStorage.setItem("temp", "data");
console.log(localStorage.getItem("temp")); // "data"

localStorage.removeItem("temp");
console.log(localStorage.getItem("temp")); // null

// Removing non-existent key (no error)
localStorage.removeItem("nonexistent"); // Safe, does nothing
```

4. clear()

Syntax:

```
storage.clear();
```

Purpose: Remove ALL key-value pairs from storage.

Examples:

```
// Add multiple items
localStorage.setItem("name", "John");
localStorage.setItem("age", "25");
localStorage.setItem("city", "NYC");
```

```
console.log(localStorage.length); // 3

// Clear everything
localStorage.clear();

console.log(localStorage.length); // 0
console.log(localStorage.getItem("name")); // null
```

5. key(index)

Syntax:

```
storage.key(index);
```

Purpose: Get the key name at a specific index.

Returns:

- Key name as string
- `null` if index out of bounds

Examples:

```
localStorage.setItem("username", "John");
localStorage.setItem("email", "john@example.com");
localStorage.setItem("age", "25");

// Get key at index 0
console.log(localStorage.key(0)); // Might be "username"

// Get key at index 1
console.log(localStorage.key(1)); // Might be "email"

// Out of bounds
console.log(localStorage.key(10)); // null
```

Note: Order is not guaranteed! Don't rely on specific key ordering.

6. length Property

Syntax:

```
storage.length
```

Purpose: Get number of stored items.

Examples:

```
console.log(localStorage.length); // 0

localStorage.setItem("key1", "value1");
console.log(localStorage.length); // 1

localStorage.setItem("key2", "value2");
console.log(localStorage.length); // 2

localStorage.removeItem("key1");
console.log(localStorage.length); // 1

localStorage.clear();
console.log(localStorage.length); // 0
```

Iterating Through Storage

Method 1: Using length and key()

```
for (let i = 0; i < localStorage.length; i++) {
  const key = localStorage.key(i);
  const value = localStorage.getItem(key);
  console.log(`${key}: ${value}`);
}
```

Method 2: Using for...in loop

```
for (let key in localStorage) {
  // Skip inherited properties
  if (localStorage.hasOwnProperty(key)) {
    console.log(`${key}: ${localStorage[key]}`);
  }
}
```

Method 3: Using Object.keys()

```
Object.keys(localStorage).forEach(key => {
  console.log(`${key}: ${localStorage.getItem(key)})`);
```

```
});
```

Method 4: Get all as object

```
function getAllStorage() {
  const items = {};
  for (let i = 0; i < localStorage.length; i++) {
    const key = localStorage.key(i);
    items[key] = localStorage.getItem(key);
  }
  return items;
}

console.log(getAllStorage());
// { username: "John", email: "john@example.com", age: "25" }
```

Different Ways to Access Storage

Three Access Methods

Your code demonstrates all three ways to access Web Storage:

Method 1: Direct Property Assignment

Syntax:

```
storage.propertyName = value;
const value = storage.propertyName;
```

From Your Code:

```
// Set
localStorage.email = "mona@gmail.com";
sessionStorage.email = "mona@gmail.com";

// Get
console.log(localStorage.email);
console.log(sessionStorage.email);

// Delete
delete localStorage.email;
```

Pros:

- Shortest syntax
- Quick for simple cases

Cons:

- Can conflict with Storage object methods/properties
- Less clear what's happening
- Can't use keys with special characters or spaces
- Returns `undefined` for non-existent keys (not `null`)

Method 2: Bracket Notation

Syntax:

```
storage["key"] = value;  
const value = storage["key"];
```

From Your Code:

```
// Set  
localStorage["username"] = "mona";  
sessionStorage["username"] = "mona";  
  
// Get  
console.log(localStorage["username"]);  
console.log(sessionStorage["username"]);  
  
// Delete  
delete localStorage["username"];
```

Pros:

- Can use dynamic keys
- Can use keys with special characters
- More flexible than dot notation

Cons:

- Still less clear than `setItem/getItem`
- Returns `undefined` for non-existent keys

Dynamic Keys Example:

```
const keyName = "user_" + userId;  
localStorage[keyName] = userData;  
  
// With special characters  
localStorage["user-email"] = "test@example.com";  
localStorage["my key with spaces"] = "value";
```

Method 3: setItem/getItem Methods (RECOMMENDED)

Syntax:

```
storage.setItem(key, value);  
const value = storage.getItem(key);
```

From Your Code:

```
// Set  
localStorage.setItem("favColor", "#000");  
sessionStorage.setItem("favColor", "#000");  
  
// Get  
console.log(localStorage.getItem("favColor"));  
console.log(sessionStorage.getItem("favColor"));  
  
// Remove  
localStorage.removeItem("favColor");
```

Pros:

- Most explicit and clear
- Returns `null` (not `undefined`) for non-existent keys
- Consistent with `removeItem()` and `clear()`
- Best practice in professional code
- No naming conflicts

Cons:

- Slightly more verbose

Comparison Example

```

// All three set the same value:

// Method 1: Direct
localStorage.username = "John";

// Method 2: Bracket
localStorage["username"] = "John";

// Method 3: setItem
localStorage.setItem("username", "John");

// All three read the same way (but different returns for non-existent keys):

// Returns undefined if doesn't exist
console.log(localStorage.username);

// Returns undefined if doesn't exist
console.log(localStorage["username"]);

// Returns null if doesn't exist (BETTER)
console.log(localStorage.getItem("username"));

```

Best Practice Recommendation

Use `setItem()` and `getItem()` for these reasons:

```

// ✅ RECOMMENDED
localStorage.setItem("username", "John");
const username = localStorage.getItem("username");

// ❌ AVOID (except for quick prototyping)
localStorage.username = "John";
const username = localStorage.username;

```

Why `getItem()` is better:

```

// Non-existent key handling

// Direct access - unclear
if (localStorage.username) { // Could be undefined
    // Do something
}

// getItem - clear and consistent

```

```
if (localStorage.getItem("username") !== null) { // Always null if missing
    // Do something
}
```

Working with Complex Data

The String-Only Problem

Web Storage only stores strings. Everything else must be converted.

The Problem:

```
// Numbers become strings
localStorage.setItem("age", 25);
console.log(typeof localStorage.getItem("age")); // "string", not "number"
console.log(localStorage.getItem("age") === 25); // false
console.log(localStorage.getItem("age") === "25"); // true

// Booleans become strings
localStorage.setItem("isActive", true);
console.log(localStorage.getItem("isActive") === true); // false
console.log(localStorage.getItem("isActive") === "true"); // true

// Objects become "[object Object]"
const user = { name: "John", age: 25 };
localStorage.setItem("user", user);
console.log(localStorage.getItem("user")); // "[object Object]" ✗
```

Solution: JSON.stringify() and JSON.parse()

Storing Complex Data:

```
const user = {
    name: "John Doe",
    age: 25,
    email: "john@example.com",
    preferences: {
        theme: "dark",
        language: "en"
    }
};
```

```
// Stringify before storing
localStorage.setItem("user", JSON.stringify(user));
```

Retrieving Complex Data:

```
// Parse after retrieving
const userString = localStorage.getItem("user");
const user = JSON.parse(userString);

console.log(user.name); // "John Doe"
console.log(user.age); // 25 (number)
console.log(user.preferences.theme); // "dark"
```

Complete Example - User Preferences

```
// Save user preferences
function savePreferences(preferences) {
    localStorage.setItem("userPreferences", JSON.stringify(preferences));
}

// Load user preferences
function loadPreferences() {
    const data = localStorage.getItem("userPreferences");

    // Check if data exists
    if (data === null) {
        // Return default preferences
        return {
            theme: "light",
            language: "en",
            notifications: true
        };
    }

    // Parse and return
    return JSON.parse(data);
}

// Update specific preference
function updatePreference(key, value) {
    const preferences = loadPreferences();
    preferences[key] = value;
    savePreferences(preferences);
}
```

```

// Usage
const prefs = {
  theme: "dark",
  language: "es",
  notifications: false
};

savePreferences(prefs);

// Later...
const loaded = loadPreferences();
console.log(loaded.theme); // "dark"

// Update single preference
updatePreference("theme", "light");

```

Storing Arrays

```

// Save array
const todos = [
  { id: 1, text: "Buy milk", done: false },
  { id: 2, text: "Walk dog", done: true },
  { id: 3, text: "Study JavaScript", done: false }
];

localStorage.setItem("todos", JSON.stringify(todos));

// Load array
const loadedTodos = JSON.parse(localStorage.getItem("todos"));

console.log(loadedTodos[0].text); // "Buy milk"
console.log(Array.isArray(loadedTodos)); // true

// Add new todo
loadedTodos.push({ id: 4, text: "New task", done: false });
localStorage.setItem("todos", JSON.stringify(loadedTodos));

```

Storing Numbers and Booleans

Option 1: Parse when retrieving

```

// Store
localStorage.setItem("age", "25");
localStorage.setItem("isActive", "true");

```

```
// Retrieve and convert
const age = parseInt(localStorage.getItem("age"));
const isActive = localStorage.getItem("isActive") === "true";

console.log(age); // 25 (number)
console.log(isActive); // true (boolean)
```

Option 2: Use objects

```
// Store as object
const data = {
  age: 25,
  isActive: true
};
localStorage.setItem("data", JSON.stringify(data));

// Retrieve as object
const loaded = JSON.parse(localStorage.getItem("data"));
console.log(loaded.age); // 25 (number)
console.log(loaded.isActive); // true (boolean)
```

Helper Functions

Safe JSON Parse:

```
function safeParseJSON(jsonString, defaultValue = null) {
  try {
    return JSON.parse(jsonString);
  } catch (e) {
    console.error("Failed to parse JSON:", e);
    return defaultValue;
  }
}

// Usage
const data = localStorage.getItem("user");
const user = safeParseJSON(data, { name: "Guest" });
```

Storage Wrapper:

```
const Storage = {
  set(key, value) {
    try {
      const stringValue = JSON.stringify(value);
      localStorage.setItem(key, stringValue);
    } catch (e) {
      console.error(`Error setting ${key}: ${e}`);
    }
  }
};

export default Storage;
```

```

        localStorage.setItem(key, stringValue);
        return true;
    } catch (e) {
        console.error("Storage set error:", e);
        return false;
    }
}

get(key, defaultValue = null) {
    try {
        const item = localStorage.getItem(key);
        return item ? JSON.parse(item) : defaultValue;
    } catch (e) {
        console.error("Storage get error:", e);
        return defaultValue;
    }
}

remove(key) {
    localStorage.removeItem(key);
}

clear() {
    localStorage.clear();
}
};

// Usage
Storage.set("user", { name: "John", age: 25 });
const user = Storage.get("user");
console.log(user.name); // "John"

```

Storage Events

Storage Event Listener

The `storage` event fires when storage is modified in **another** tab/window.

Important: Event does NOT fire in the tab that made the change!

Syntax:

```
window.addEventListener("storage", function(e) {
  console.log("Storage changed!");
});
```

Storage Event Object Properties

```
window.addEventListener("storage", function(event) {
  console.log("Key:", event.key);           // Key that changed
  console.log("Old Value:", event.oldValue); // Previous value
  console.log("New Value:", event.newValue); // New value
  console.log("URL:", event.url);          // Page URL where change occurred
  console.log("Storage:", event.storageArea); // localStorage or
sessionStorage
});
```

Practical Example - Sync Across Tabs

Tab 1: User logs in

```
function login(username) {
  localStorage.setItem("currentUser", username);
  // This tab doesn't receive storage event
  updateUI(username);
}
```

Tab 2: Listen for login

```
// Listen for storage changes
window.addEventListener("storage", function(e) {
  if (e.key === "currentUser") {
    if (e.newValue) {
      // User logged in another tab
      console.log("User logged in:", e.newValue);
      updateUI(e.newValue);
    } else {
      // User logged out in another tab
      console.log("User logged out");
      showLoginScreen();
    }
  }
});

function updateUI(username) {
```

```
        document.getElementById("username").textContent = username;
        document.getElementById("loginBtn").style.display = "none";
    }
}
```

Complete Multi-Tab Sync Example

```
// Sync user preferences across tabs
const PreferenceSync = {
    init() {
        // Listen for changes in other tabs
        window.addEventListener("storage", (e) => {
            if (e.key === "preferences") {
                this.onPreferencesChanged(e newValue);
            }
        });
    },
    savePreferences(prefs) {
        localStorage.setItem("preferences", JSON.stringify(prefs));
        // Apply in current tab
        this.applyPreferences(prefs);
    },
    onPreferencesChanged(newValue) {
        if (newValue) {
            const prefs = JSON.parse(newValue);
            this.applyPreferences(prefs);
        }
    },
    applyPreferences(prefs) {
        // Apply theme
        document.body.className = prefs.theme;

        // Apply language
        document.documentElement.lang = prefs.language;

        console.log("Preferences applied:", prefs);
    }
};

// Initialize
PreferenceSync.init();

// Usage
```

```
PreferenceSync.savePreferences({
  theme: "dark",
  language: "en"
});
```

Storage Event Limitations

Event does NOT fire:

- In the same tab that made the change
- When using sessionStorage (each tab has separate sessionStorage)
- For clear() operation (event.key will be null)

Detecting clear():

```
window.addEventListener("storage", function(e) {
  if (e.key === null) {
    console.log("Storage was cleared!");
    // All items removed
  }
});
```

Best Practices

1. Always Check for Support

```
if (typeof Storage !== "undefined") {
  // Storage is supported
  localStorage.setItem("test", "value");
} else {
  // Storage not supported
  console.error("Web Storage not supported");
  // Fallback to cookies or server storage
}
```

Feature Detection Function:

```
function storageAvailable(type) {
  try {
    const storage = window[type];
    const test = "__storage_test__";
```

```

        storage.setItem(test, test);
        storage.removeItem(test);
        return true;
    } catch (e) {
        return false;
    }
}

if (storageAvailable("localStorage")) {
    // Use localStorage
} else {
    // Use alternative
}

```

2. Handle Quota Exceeded Errors

```

function safesetItem(key, value) {
    try {
        localStorage.setItem(key, value);
        return true;
    } catch (e) {
        if (e.name === "QuotaExceededError") {
            console.error("Storage quota exceeded!");
            // Handle by removing old items or notifying user
            return false;
        }
        throw e;
    }
}

// Usage
if (!safesetItem("bigData", largeString)) {
    alert("Storage full! Please clear some data.");
}

```

3. Use Namespacing

Problem: Key conflicts between different parts of application

Solution: Prefix keys with namespace

```

// Without namespace (risky)
localStorage.setItem("user", "John"); // Could conflict with other apps

// With namespace (safe)

```

```

localStorage.setItem("myApp_user", "John");
localStorage.setItem("myApp_settings", "{}");
localStorage.setItem("myApp_cache", "{}");

// Helper object
const AppStorage = {
  prefix: "myApp_",

  set(key, value) {
    localStorage.setItem(this.prefix + key, JSON.stringify(value));
  },

  get(key) {
    const item = localStorage.getItem(this.prefix + key);
    return item ? JSON.parse(item) : null;
  },

  remove(key) {
    localStorage.removeItem(this.prefix + key);
  },

  clear() {
    // Clear only app-specific keys
    Object.keys(localStorage).forEach(key => {
      if (key.startsWith(this.prefix)) {
        localStorage.removeItem(key);
      }
    });
  }
};

// Usage
AppStorage.set("user", { name: "John" });
const user = AppStorage.get("user");

```

4. Add Expiration Dates

Local Storage doesn't expire automatically. Implement your own expiration:

```

const StorageWithExpiry = {
  set(key, value, ttl) {
    const now = new Date();
    const item = {
      value: value,
      expiry: now.getTime() + ttl // ttl in milliseconds
    }
    localStorage.setItem(this.prefix + key, JSON.stringify(item));
  }

  get(key) {
    const item = localStorage.getItem(this.prefix + key);
    if (!item) return null;
    const parsedItem = JSON.parse(item);
    if (parsedItem.expiry < now.getTime()) {
      return null;
    }
    return parsedItem.value;
  }

  remove(key) {
    localStorage.removeItem(this.prefix + key);
  }
};

```

```

    };

    localStorage.setItem(key, JSON.stringify(item));
}

get(key) {
    const itemStr = localStorage.getItem(key);

    if (!itemStr) {
        return null;
    }

    const item = JSON.parse(itemStr);
    const now = new Date();

    // Check if expired
    if (now.getTime() > item.expiry) {
        localStorage.removeItem(key);
        return null;
    }

    return item.value;
}
};

// Usage - set data with 1 hour expiry
StorageWithExpiry.set("sessionData", { user: "John" }, 60 * 60 * 1000);

// Later - get data (returns null if expired)
const data = StorageWithExpiry.get("sessionData");

```

5. Validate Retrieved Data

```

function getValidatedUser() {
    const userString = localStorage.getItem("user");

    // Check if exists
    if (!userString) {
        return null;
    }

    try {
        const user = JSON.parse(userString);

        // Validate structure
        if (!user.name || !user.email) {

```

```

        console.error("Invalid user data");
        localStorage.removeItem("user");
        return null;
    }

    // Validate email format
    const emailRegex = /^[^@\s]+@[^\s@]+\.\[^@\s]+$/;
    if (!emailRegex.test(user.email)) {
        console.error("Invalid email");
        return null;
    }

    return user;
} catch (e) {
    console.error("Failed to parse user data:", e);
    localStorage.removeItem("user");
    return null;
}
}

```

6. Minimize Storage Usage

```

// ❌ BAD - Storing large, repeated data
localStorage.setItem("data1", JSON.stringify({
    timestamp: "2024-01-15T10:30:00Z",
    user: "john@example.com",
    // ...lots of data
}));

// ✅ GOOD - Compress and minimize
localStorage.setItem("data1", JSON.stringify({
    t: 1705318200000, // Unix timestamp
    u: "john@example.com",
    // abbreviated keys
}));

// Store common data once
localStorage.setItem("commonConfig", JSON.stringify({ ... }));

```

7. Clear Old Data Regularly

```

function cleanupOldData() {
    const maxAge = 30 * 24 * 60 * 60 * 1000; // 30 days
    const now = Date.now();

```

```

Object.keys(localStorage).forEach(key => {
  try {
    const item = JSON.parse(localStorage.getItem(key));

    if (item.timestamp && (now - item.timestamp > maxAge)) {
      localStorage.removeItem(key);
      console.log("Removed old item:", key);
    }
  } catch (e) {
    // Not a JSON object, skip
  }
});

// Run on app startup
cleanupOldData();

```

Common Use Cases

1. User Preferences/Settings

```

const UserSettings = {
  save(settings) {
    localStorage.setItem("userSettings", JSON.stringify(settings));
  },

  load() {
    const settings = localStorage.getItem("userSettings");
    return settings ? JSON.parse(settings) : this.getDefaults();
  },

  getDefaults() {
    return {
      theme: "light",
      language: "en",
      fontSize: 16,
      notifications: true
    };
  },

  update(key, value) {
    const settings = this.load();

```

```

        settings[key] = value;
        this.save(settings);
    }
};

// Usage
UserSettings.update("theme", "dark");
const settings = UserSettings.load();
document.body.className = settings.theme;

```

2. Shopping Cart

```

const ShoppingCart = {
    add(product) {
        const cart = this.get();

        // Check if product exists
        const existing = cart.find(item => item.id === product.id);

        if (existing) {
            existing.quantity += 1;
        } else {
            cart.push({ ...product, quantity: 1 });
        }

        this.save(cart);
    },

    remove(productId) {
        let cart = this.get();
        cart = cart.filter(item => item.id !== productId);
        this.save(cart);
    },

    updateQuantity(productId, quantity) {
        const cart = this.get();
        const item = cart.find(item => item.id === productId);

        if (item) {
            item.quantity = quantity;
            this.save(cart);
        }
    },
}

get() {

```

```

        const cart = localStorage.getItem("shoppingCart");
        return cart ? JSON.parse(cart) : [];
    },

    save(cart) {
        localStorage.setItem("shoppingCart", JSON.stringify(cart));
    },

    clear() {
        localStorage.removeItem("shoppingCart");
    },

    getTotal() {
        const cart = this.get();
        return cart.reduce((total, item) => {
            return total + (item.price * item.quantity);
        }, 0);
    },

    getItemCount() {
        const cart = this.get();
        return cart.reduce((total, item) => total + item.quantity, 0);
    }
};

// Usage
ShoppingCart.add({ id: 1, name: "T-Shirt", price: 19.99 });
ShoppingCart.add({ id: 2, name: "Jeans", price: 49.99 });
console.log("Cart total:", ShoppingCart.getTotal());

```

3. Form Auto-Save

```

const FormAutoSave = {
    init(formId) {
        const form = document.getElementById(formId);
        const inputs = form.querySelectorAll("input, textarea, select");

        // Load saved data
        this.load(formId, inputs);

        // Save on input change
        inputs.forEach(input => {
            input.addEventListener("input", () => {
                this.save(formId, inputs);
            });
        });
    }

    load(formId, inputs) {
        const savedData = localStorage.getItem(formId);
        if (savedData) {
            inputs.forEach(input => {
                const value = JSON.parse(savedData)[input.name];
                if (value !== undefined) {
                    input.value = value;
                }
            });
        }
    }

    save(formId, inputs) {
        const data = {};
        inputs.forEach(input => {
            data[input.name] = input.value;
        });
        localStorage.setItem(formId, JSON.stringify(data));
    }
};

```

```
});

// Clear on submit
form.addEventListener("submit", () => {
    this.clear(formId);
});

},
};

save(formId, inputs) {
    const data = {};

    inputs.forEach(input => {
        if (input.type === "checkbox") {
            data[input.name] = input.checked;
        } else if (input.type === "radio") {
            if (input.checked) {
                data[input.name] = input.value;
            }
        } else {
            data[input.name] = input.value;
        }
    });
}

sessionStorage.setItem(`form_${formId}`, JSON.stringify(data));
},

load(formId, inputs) {
    const data = sessionStorage.getItem(`form_${formId}`);

    if (!data) return;

    const savedData = JSON.parse(data);

    inputs.forEach(input => {
        const value = savedData[input.name];

        if (value !== undefined) {
            if (input.type === "checkbox") {
                input.checked = value;
            } else if (input.type === "radio") {
                if (input.value === value) {
                    input.checked = true;
                }
            } else {
                input.value = value;
            }
        }
    });
}
```

```

        }
    });
},
};

clear(formId) {
    sessionStorage.removeItem(`form_${formId}`);
}
};

// Usage
FormAutoSave.init("contactForm");

```

4. Recently Viewed Items

```

const RecentlyViewed = {
    maxItems: 10,

    add(item) {
        let items = this.get();

        // Remove if already exists
        items = items.filter(i => i.id !== item.id);

        // Add to beginning
        items.unshift(item);

        // Keep only max items
        items = items.slice(0, this.maxItems);

        localStorage.setItem("recentlyViewed", JSON.stringify(items));
    },

    get() {
        const items = localStorage.getItem("recentlyViewed");
        return items ? JSON.parse(items) : [];
    },

    clear() {
        localStorage.removeItem("recentlyViewed");
    }
};

// Usage
RecentlyViewed.add({
    id: 123,

```

```

        name: "Product Name",
        image: "url.jpg"
    });

const recent = RecentlyViewed.get();
console.log("Recently viewed:", recent);

```

5. Authentication Token Storage

```

const Auth = {
    setToken(token, expiresIn) {
        const expiryTime = Date.now() + (expiresIn * 1000);

        localStorage.setItem("authToken", token);
        localStorage.setItem("tokenExpiry", expiryTime.toString());
    },

    getToken() {
        const token = localStorage.getItem("authToken");
        const expiry = localStorage.getItem("tokenExpiry");

        if (!token || !expiry) {
            return null;
        }

        // Check if expired
        if (Date.now() > parseInt(expiry)) {
            this.clearToken();
            return null;
        }

        return token;
    },

    clearToken() {
        localStorage.removeItem("authToken");
        localStorage.removeItem("tokenExpiry");
    },

    isAuthenticated() {
        return this.getToken() !== null;
    }
};

// Usage

```

```
Auth.setToken("abc123xyz", 3600); // Token expires in 1 hour

if (Auth.isAuthenticated()) {
    // User is logged in
    const token = Auth.getToken();
    // Make API calls with token
} else {
    // Redirect to login
}
```

Security Considerations

1. Don't Store Sensitive Data

NEVER store in localStorage/sessionStorage:

- Passwords
- Credit card numbers
- Social security numbers
- API keys (client-side)
- Personally identifiable information (PII)

Why?

- Storage is accessible to any JavaScript on the page
- Vulnerable to XSS (Cross-Site Scripting) attacks
- Not encrypted
- Readable in browser dev tools

2. XSS Vulnerability

```
// ❌ DANGEROUS - User input directly stored
const userComment = getUserInput();
localStorage.setItem("comment", userComment);

// Later, displayed without sanitization
document.getElementById("comment").innerHTML =
localStorage.getItem("comment");

// If user input is: <script>alert('XSS')</script>
// It will execute!
```

Solution: Sanitize data

```
function sanitize(str) {
  const div = document.createElement('div');
  div.textContent = str;
  return div.innerHTML;
}

// Store sanitized data
const userComment = sanitize(getUserInput());
localStorage.setItem("comment", userComment);

// Or use textContent instead of innerHTML
document.getElementById("comment").textContent =
localStorage.getItem("comment");
```

3. Validate Data on Retrieval

```
function getValidatedData(key) {
  const data = localStorage.getItem(key);

  if (!data) {
    return null;
  }

  try {
    const parsed = JSON.parse(data);

    // Validate structure
    if (typeof parsed !== 'object' || parsed === null) {
      throw new Error("Invalid data structure");
    }

    // Additional validation based on expected structure
    // ...

    return parsed;
  } catch (e) {
    console.error("Data validation failed:", e);
    localStorage.removeItem(key);
    return null;
  }
}
```

4. Same-Origin Policy

Storage is isolated by origin (protocol + domain + port):

```
// https://example.com can access:  
localStorage.setItem("data", "value");  
  
// https://example.com:8080 CANNOT access the same storage  
// http://example.com CANNOT access the same storage  
// https://subdomain.example.com CANNOT access the same storage  
  
// Each origin has completely separate storage
```

5. Size Limits

```
function checkStorageSize() {  
    let total = 0;  
  
    for (let key in localStorage) {  
        if (localStorage.hasOwnProperty(key)) {  
            total += localStorage[key].length + key.length;  
        }  
    }  
  
    // Convert to KB  
    const sizeKB = (total / 1024).toFixed(2);  
    console.log(`Storage used: ${sizeKB} KB`);  
  
    // Typical limit is 5-10MB (5120-10240 KB)  
    const percentUsed = (total / (5 * 1024 * 1024) * 100).toFixed(2);  
    console.log(`Approximately ${percentUsed}% of 5MB used`);  
  
    return { bytes: total, kb: sizeKB, percentUsed };  
}  
  
checkStorageSize();
```

6. Privacy Mode

```
// Some browsers block storage in private/incognito mode  
function checkStorageWorking() {  
    try {  
        const test = '__test__';  
        localStorage.setItem(test, test);  
    } catch (err) {  
        console.error(`Error: ${err.message}`);  
    }  
}
```

```

        localStorage.removeItem(test);
        return true;
    } catch (e) {
        console.warn("Storage not available (private mode or disabled)");
        return false;
    }
}

if (!checkStorageWorking()) {
    // Fallback to in-memory storage or cookies
}

```

Summary

Local Storage vs Session Storage Quick Reference

Feature	localStorage	sessionStorage
Lifetime	Permanent	Until tab closes
Scope	All tabs	Current tab only
Survives	Browser restart	Page reload only
Use For	Long-term data	Temporary session data
Examples	Preferences, cart	Form data, temp state

Key Methods Summary

```

// Both storage types use same API

// Set
storage.setItem(key, value);

// Get (returns null if not found)
storage.getItem(key);

// Remove
storage.removeItem(key);

// Clear all
storage.clear();

```

```
// Get number of items  
storage.length;  
  
// Get key at index  
storage.key(index);
```

Best Practices Checklist

- Use `setItem()` and `getItem()` methods
- Always stringify objects with `JSON.stringify()`
- Always parse retrieved objects with `JSON.parse()`
- Check for storage support before using
- Handle `QuotaExceededError` exceptions
- Use namespacing to avoid key conflicts
- Validate retrieved data
- Don't store sensitive information
- Add expiration for cached data
- Clean up old data regularly

When to Use Each

Use `localStorage` for:

- User preferences (theme, language)
- Shopping cart
- Recently viewed items
- Cached data
- User settings
- Long-term application state

Use `sessionStorage` for:

- Multi-step form data
- Temporary authentication state
- Single-page app state
- Tab-specific data
- Wizard/workflow progress
- Temporary filters/search state

Web Storage API provides a simple, powerful way to persist data in the browser, making web applications faster and more user-friendly by reducing server requests and maintaining state across sessions.

Abdullah Ali

Contact : +201012613453