HyperionDev

# JSON and Web Storage APIs

## Task

Visit our website

# Introduction

In web development, managing user data and enabling communication between the client and server are essential tasks. JavaScript allows for capturing and handling user input, as well as storing data on the client side. This can be done through methods like cookies, local storage, and the Web Storage API.

Data is often transferred between the client and server in the form of JSON (JavaScript Object Notation), which is a lightweight format for representing JavaScript objects. Understanding JSON syntax and methods is key to working with data efficiently. In addition to JSON, XML is another format used for data exchange.

Effective data storage techniques, such as caching and session storage, help optimise performance by reducing load times and improving the user experience. This task will provide an overview of how these technologies work together, covering the various ways data is managed, stored, and transmitted in modern web development.

## User input and output with JavaScript

JavaScript enables developers to engage with users by receiving input and delivering output through interactive dialogue boxes. Dialogue boxes are a user interface element, built into web browsers, that interact directly with the user while they are interacting with a web page.

These dialogue boxes are known as a modal window. A modal window interrupts the normal execution of code, disables user interaction from the main application page, and requires users to focus on a specific window before continuing. A user can dismiss the dialogue box by clicking the “OK” button that’s typically present, after which the execution of the code resumes.

It is important to note that these dialogue boxes are not part of the development environment itself, such as VS Code, where the code is written and executed.

Two fundamental functions for engaging with these dialogue boxes are `prompt()` and `alert()`.

### Capturing user input with prompt()

In JavaScript, the `prompt()` function triggers a pop-up dialogue that prompts users to input data such as text or numbers. The prompt function accepts a message as its parameter and then displays it as a prompt to the user. After the user responds, the function returns the *entered value* that can be stored in a variable for later use.

## Displaying user output with alert()

In JavaScript, the `alert()` function triggers a pop-up dialogue box that alerts the user of an action or event by showcasing a specified message. Similar to the prompt function, the alert function takes a message as its parameter, and upon execution, displays the message to the user within the pop-up dialogue box.

## Combining input and output

You can use both `prompt()` and `alert()` functions together to create interactive experiences. Let's take a look at the following example that takes a user's input and immediately provides a response using an alert:

```
// Prompt user to input data
const userName = prompt("Please enter your name:");

// Display a greeting using the alert function if the data is valid
if (userName !== null && userName !== "") {
  const greeting = "Hello, " + userName + "!";
  alert(greeting);
}
// Display an appropriate response if the user dismisses the dialogue box without
entering valid data
else {
  alert("You did not provide a valid name.");
}
```

Try this:

- Open a new window in your browser and hit F12 to inspect it.
- In the console tab, enter `alert("This is an alert!")` and hit enter.
- As you execute the command, you will notice a dialogue box that promptly appears on your screen. This is the result of the `alert()` function in action.

## Handling multi-parameter input in forms

For multi-parameter input and for a more user-friendly approach, use HTML forms combined with DOM manipulation:

```
<body>
  <form id="userInputForm">
    <label for="param1">Parameter 1:</label>
    <input type="text" id="param1">
    <label for="param2">Parameter 2:</label>
    <input type="text" id="param2">
```

```
    <label for="param3">Parameter 3:</label>
    <input type="text" id="param3">
    <button type="button" onclick="createObject()">Submit</button>
  </form>

  <script>
    function createObject() {
      const param1 = document.getElementById('param1').value;
      const param2 = document.getElementById('param2').value;
      const param3 = document.getElementById('param3').value;

      const userObject = { param1, param2, param3 };
      console.log(userObject);
    }
  </script>
</body>
```

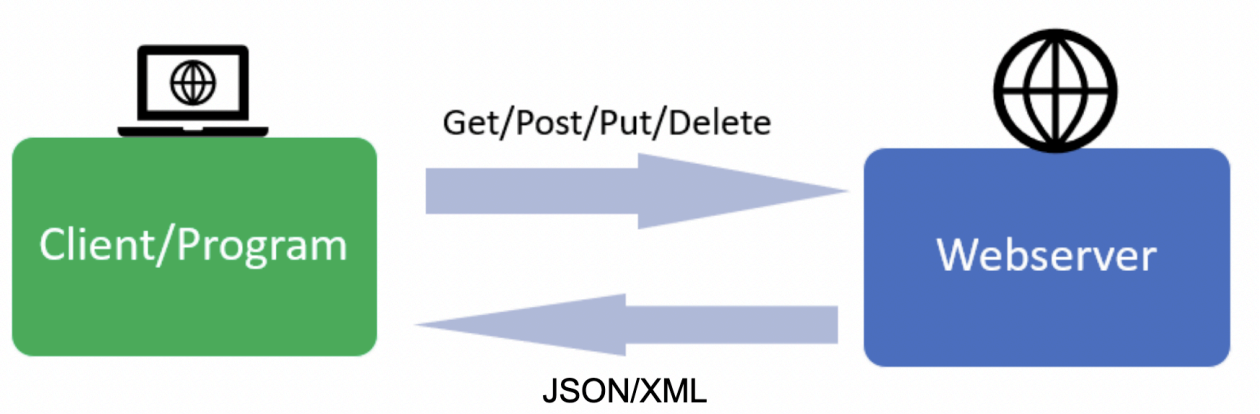
In this multi-parameter HTML form above, the emphasis is on collecting three separate inputs (`param1`, `param2`, and `param3`) and processing them together in the `createObject` function.

## Sending objects between a web server and clients

Everything that is transferred over the web is transferred using HTTP (HyperText Transfer Protocol). As the name suggests, this protocol can transfer text. All data that is transferred across the web is, therefore, transferred as text. As such, we cannot transfer JavaScript objects between a web server and a client.

There are two very important facts about HTTP that we need to keep in mind, though:

1. **HTTP is a stateless protocol**. That means HTTP does not see any link between two requests being successively carried out on the same connection. Cookies and the Web Storage API are used to store necessary state information.
2. **HTTP transfers text (not objects or other complex data structures)**. XML and JSON are commonly used to convert data structures, like objects, into a text format that can be transferred using HTTP.




## Extensible markup language (XML)

eXtensible Markup Language (XML) is a markup language used to annotate text or add additional information. Tags are used to annotate data. These tags are not shown to the end-user but are needed by the 'machine' to read and process the text correctly.

Below is an example of XML. Notice that the tags are placed on the left and right of the data you want to markup to wrap around the data:

```
<book id="bk101">
  <author>Gambardella, Matthew</author>
  <title>XML Developer's Guide</title>
  <genre>Computer</genre>
  <price>44.95</price>
  <publish_date>2000-10-01</publish_date>
  <description>An in-depth look at creating applications with XML.</description>
</book>
```

This is the general pattern that we have to follow for all tags in XML:

```
<opening tag>Some text here.</closing tag>
```

Looking at the example of XML above may remind you of HTML. They are both markup languages but whereas HTML focuses on displaying data, XML just carries data.
XML files do not do anything except carry informational data wrapped in tags. XML structures, stores, and transports the data independent of hardware or software. We do, however, need software such as XML parsers or applications capable of interpreting XML data, like web browsers, text editors with XML support, or specific development tools designed to work with XML to read and display this data in a meaningful way.

# JavaScript object notation (JSON)

JSON, or JavaScript Object Notation, is a syntax for converting objects, arrays, numbers, strings, and booleans into a format that can be transferred between the web server and the client. JSON is language-independent like XML.

Only text data can be exchanged between a browser and a server. JSON is text, and any JavaScript object can be converted into JSON, which can then be sent to the server. Any JSON data received from the server can also be converted into JavaScript objects. We can work more efficiently with data as JavaScript objects without the need for complicated parsing and translations. Data must also be in a certain format to store it. JSON makes it possible to store JavaScript objects as text which is always one of the legal formats.

JSON is much more lightweight than XML as it is shorter and quicker to read and write. JSON also does not use end tags. XML must also be parsed with an XML parser, while a standard JavaScript function can parse JSON.

## JSON syntax

The JSON syntax is a subset of the JavaScript syntax. However, this does not mean JSON cannot be used with other languages. JSON works well with PHP, Perl, Python, Ruby, Java, and Ajax, to name but a few.

Below are some JSON Syntax rules:

- Data is in key-value pairs.
- Property names must be in double-quoted strings.
- Data is separated by commas.
- Objects are held by curly braces - `{ }`
- Arrays are held by square brackets - `[ ]`

As mentioned above, JSON data is written as key/value pairs. Each pair consists of a field name or key written in double quotes, a colon, and a value.

Example:

```
"name" : "Jason"
```

The key must be a string, enclosed in double quotes, while the value can be a string, a number, a JSON object, an array, a boolean, or null.

JSON uses JavaScript syntax, so you do not need any additional JS library to work with JSON within JavaScript. The file type for JSON files is **.json** and the MIME (Multipurpose Internet Mail Extensions) type for JSON text is **`"application/json"`**. MIME is a standard that extends the format of email messages to support text in character sets other than ASCII. It also supports audio messages, videos, and images. In the context of an email, MIME headers are used to specify the type of content within an email. However, MIME is not just limited to emails. It is also used in HTTP to specify the type of the data being sent in an HTTP response i.e. audios, videos, and images.

## JSON objects

Let's take a look at the example below of a JSON object. Object values can be accessed using either the dot (.) notation or the square brackets (`[]`) notation.

```
let myObj = { name: "Jason", age: 30, car: null };
// Assign 'name' to x using dot notation
let x = myObj.name;
// Or assign 'name' to y using bracket notation
let y = myObj["name"];
```

The dot or bracket notation can also be used to modify any value in a JSON object:

```
myObj.age = 31; // Using dot notation
myObj["age"] = 31; // Using bracket notation
```

To delete properties from a JSON object, you use the **`delete`** keyword:

```
delete myObj.age;
```

Saving an array of objects with JSON is also possible. See the example below where an array of objects is stored. The first object in the array describes a person object with the name "Tom Smith," and the second object describes a person called "Jack Daniels":

```
let personArray = [
  {
    name: { first: "Tom", last: "Smith" },
    age: 21,
    gender: "male",
    interests: "Programming"
  },
  {
    name: { first: "Jack", last: "Daniels" },
    age: 19,
    gender: "male",
    interests: "Gaming"
```

```
  },
];
```

You can use a for-in loop together with bracket notation to loop through an object's properties and access the property values, as shown below. **hasOwnProperty()** is a built-in method in JavaScript that determines whether an object has a specific property as its own and is not inherited from an object's prototype:

```
const myObj = { name: "Jason", age: 30, car: null };
for (let key in myObj) {
  if (myObj.hasOwnProperty(key)) {
    console.log(`${key}: ${myObj[key]}`);
  }
}
```

Once the above has been logged to the console, you can then access the property values in a for-in loop, using the bracket notation, to load and display it in the browser:

```
  <body>
    <p id="demo"></p>
    <script>
      let myObj = { name: "Jason", age: 30, car: null };
      for (x in myObj) {
        document.getElementById("demo").innerHTML += myObj[x];
      }
    </script>
  </body>
```

A JSON object can contain other JSON objects:

```
let myObj = {
  name:"Jason",
  age:30,
  cars: {
    car1:"Ford",
    car2:"BMW",
    car3:"VW",
  },
};
```

To access a nested JSON object, simply use the dot or bracket notation:

```
let x = myObj.cars.car2;
// Or
let x = myObj.cars["car2"];
```

## JSON methods

As mentioned before, one of the key reasons for using JSON is to convert certain JavaScript data types (like arrays) into text that can be transferred using HTTP. To do this, we use the following two JSON methods:

- `JSON.parse()`

The data becomes a JavaScript object by parsing the data using `JSON.parse()`. Imagine that you receive the following text from a web server:

```
'{ "name":"Jason", "age":30, "city":"New York"}'
```

By using the `JSON.parse()` function, the text is converted into a JavaScript object:

```
let obj = JSON.parse('{ "name":"Jason", "age":30, "city":"New York"}');
```

You can now access the properties of this object using the dot notation, such as `obj.name`, `obj.age`, and `obj.city`.

Once we have the data as a JavaScript object, we can use it to manipulate the content of our HTML page. In the following example, we will display the name and age properties of our object inside a paragraph element with the id `"demo"`:

```
  <body>
    <p id="demo"></p>
    <script>
      // Define the object
      let obj = JSON.parse('{ "name":"Jason", "age":30, "city":"New York"}');

      // Update the content of the paragraph
      document.getElementById("demo").innerHTML = obj.name + ", " + obj.age;
    </script>
  </body>
```

When this code is run, the content of the paragraph element will be updated to show:

```
Jason, 30
```

Here is the complete code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>JSON Parsing Example</title>
```

```
  </head>
  <body>
    <p id="demo"></p>
    <script>
      // Define the object
      let obj = JSON.parse('{ "name":"Jason", "age":30, "city":"New York"}');

      // Update the content of the paragraph
      document.getElementById("demo").innerHTML = obj.name + ", " + obj.age;
    </script>
  </body>
</html>
```

- `JSON.stringify()`

  All the data you send to a web server has to be a string. To convert a JavaScript object into a string, you use **`JSON.stringify()`**.

Imagine that you have the following JavaScript object:

```
let obj = { name:"Jason", age:30, city:"New York"};
```

Using the **`JSON.stringify()`** function converts this object into a string:

```
let myJSON = JSON.stringify(obj);
```

Now, **`myJSON`** is a string that represents the original object, and it can be sent to a server or displayed within an HTML document. Here's how you can display the JSON string inside a paragraph element with the id "demo":

```
let obj = { name:"Jason", age:30, city:"New York"};
let myJSON = JSON.stringify(obj);
document.getElementById("demo").innerHTML = myJSON;
```

Here is the complete code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>JSON Example</title>
  </head>
  <body>
    <p id="demo"></p>
    <script>
      let obj = { name: "Jason", age: 30, city: "New York" };
      let myJSON = JSON.stringify(obj);
      document.getElementById("demo").innerHTML = myJSON;
    </script>
  </body>
</html>
```

## Cookies introduction and use cases

Cookies are small files stored on a user's browser by websites. They store data like session IDs or user preferences. Cookies are sent back to the server with every HTTP request, making them ideal for stateful interactions like remembering login sessions or tracking user preferences.

Setting a cookie:

```
document.cookie = "username=JohnDoe; expires=Fri, 31 Dec 2024 23:59:59 GMT; path=/";
```

Retrieving a cookie:

```
console.log(document.cookie);
```

Deleting a cookie:

```
document.cookie = "username=JohnDoe; expires=Thu, 01 Jan 1970 00:00:00 GMT; path=/";
```

Here is the complete code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cookies Example</title>
  </head>
  <body>
    <script>
      // Set a username cookie with an expiration date
      document.cookie = "username=JohnDoe; expires=Fri, 31 Dec 2024 23:59:59 GMT;
path=/";
      console.log(document.cookie);

      // Delete the username cookie by setting its expiration date to a past date
      document.cookie = "username=JohnDoe; expires=Thu, 01 Jan 1970 00:00:00 GMT;
path=/";
    </script>
  </body>
</html>
```

You can view cookies in your browser's developer tools. First, right-click on the page and select Inspect. Thereafter, go to the Application or Storage tab and select Cookies to view all stored cookies.

### Take note

Chrome has strict settings for third-party cookies, and it may block cookies if the site is not considered secure. This can cause issues when you're testing cookies locally.

To solve this, you can use the **Live Server** extension in VS Code. This extension serves your project over http://localhost:<port>, making it behave more like a live website and helping to resolve cookie issues during testing.

However, if you still run into problems, it might be a good idea to test your cookies in other browsers like Firefox or Edge, as they tend to have fewer restrictions on local cookie testing compared to Chrome.

---

## Privacy considerations and cookie consent

Websites must notify users about cookie usage and obtain consent due to privacy laws (e.g., GDPR). This ensures transparency about data collection.

A simple cookie consent banner:

```
  <body>
    <div id="cookie-consent" style="display: none;">
      This website uses cookies to enhance your experience.
      <button onclick="acceptCookies()">Accept</button>
    </div>

    <script>
      function acceptCookies() {
        document.cookie = "consent=true; expires=Fri, 31 Dec 2024 23:59:59 GMT;
path=/";
        document.getElementById("cookie-consent").style.display = "none";
      }

     // Show the banner if consent not given
     if (!document.cookie.includes("consent=true")) {
       document.getElementById("cookie-consent").style.display = "block";
     }
    </script>
  </body>
```

# Local storage: Persistent storage on the client

`localStorage` is a built-in JavaScript API for storing data directly on a user's browser. Unlike cookies, it does not send stored data to the server with every request, making it lightweight and efficient for client-side storage. Data in `localStorage` persists even after the user closes the browser or restarts their device, as long as they do not manually clear it. It is most commonly used for:

1. Storing user settings (e.g., dark theme preference).
2. Saving application state for quick retrieval.
3. Caching less sensitive data that doesn't require constant syncing with a server.

**Key features**:

- **Capacity**: `localStorage` can store up to 5MB of data in most modern browsers.
- **Data format**: Stores data as key-value pairs in string format. Non-string data must be converted (e.g., JSON).
- **Lifespan**: Persistent until explicitly removed by the application or user.

Setting an item in `localStorage`:

```
localStorage.setItem("theme", "dark");
```

Retrieving an item:

```
let theme = localStorage.getItem("theme");
console.log(theme); // Outputs: dark
```

Removing an item:

```
localStorage.removeItem("theme");
```

### Converting data to strings

To store complex objects:

```
const user = { id: 1, name: "JohnDoe" };
localStorage.setItem("user", JSON.stringify(user));
```

To retrieve and parse:

```
const user = JSON.parse(localStorage.getItem("user"));
console.log(user.name); // Output: JohnDoe
```

Limitations:

- Data must be manually synced to a server for backup or shared usage.
- Not suitable for sensitive data like passwords or authentication tokens.
- Doesn't support expiration times, unlike cookies.

## Cache: Efficient data storage

A cache temporarily stores resources (e.g., images, scripts, or data) to improve web performance. By storing frequently accessed resources locally, a cache reduces the need for repeated downloads, saving bandwidth and speeding up load times.

**Service workers** are JavaScript scripts that allow developers to control the caching process programmatically. They work in the background, intercepting and managing network requests to serve files from the cache when possible, providing offline support and optimised performance.

**Tip**: Use service workers to control cache behaviour programmatically.

A basic service worker script to cache files:

```
self.addEventListener("install", (event) => {
  event.waitUntil(
      caches.open("my-cache").then((cache) => {
      return cache.addAll(["/index.html", "/style.css", "/script.js"]);
```

```
      })
  );
});
```

Retrieving from the cache:

```
self.addEventListener("fetch", (event) => {
  event.respondWith(
      caches.match(event.request).then((response) => {
      return response || fetch(event.request);
      })
  );
});
```

This script ensures that specified files are available offline and speeds up the user experience by serving cached versions of files when they are already available.

To deepen your understanding of caching strategies and service workers, refer to the following resource for a comprehensive guide: **MDN Web Docs**.

## The web storage API: session storage

Thus far, we have used variables to store data that is used in our programmes. When storing data that is used for web applications it is important to keep in mind that HTTP is a *stateless protocol*. That basically means that the web server does not store information about each user's interaction with the website. For example, if 100 people are shopping online, the web server that hosts the online shopping application doesn't necessarily store the state of each person's shopping experience (e.g. how many items each person has added or removed from their shopping cart). Instead, that type of information is stored on the browser using **Cookies** or the **Web Storage API** (the more modern and efficient solution for client storage). The Web Storage API stores data using key-value pairs. This mechanism of storing data has, to a large extent, replaced the use of cookies.

The Web Storage API allows us to store state information in two ways:

1. `sessionStorage` stores state information for each given origin for as long as the browser is open.
2. `localStorage` stores state information for each given origin even when the browser is closed and reopened.

You can add items to sessionStorage as shown below:

```
sessionStorage.setItem("totalPersonObjs", 1);
```

This will add the key value pair {“totalPersonObjs”, 1} to sessionStorage. You could retrieve a value from sessionStorage as shown below:

```
let total = parseInt(sessionStorage.getItem("totalPersonObjs"));
```

For more information about how to use `sessionStorage`, refer to the following resource **here**.

# When to use: practical use cases

Each storage method has specific advantages, making it ideal for particular situations depending on your application's needs.

1. **Cookies** are best suited for scenarios where you need to share data between the client and server. For instance, they are commonly used for maintaining user sessions by storing authentication tokens or user preferences. Cookies are sent with every request to the server, which makes them ideal for actions that need server validation, such as remembering a logged-in user across pages.

2. **sessionStorage** is a great option for storing temporary data that only needs to exist for the duration of a single session. Once the browser tab is closed, all data stored in sessionStorage is automatically cleared. This makes it perfect for cases where you want to hold user input during a multi-step form process, or to cache data temporarily for fast retrieval within the same browsing session.

3. **localStorage** is ideal for data that needs to persist even after the browser or device is restarted, but doesn't require frequent syncing with a server. This makes it an excellent choice for saving user preferences, such as theme settings (e.g., dark mode) or other personal settings that should be available the next time the user visits the website. Additionally, localStorage is useful for caching static data like API responses, reducing the need to fetch the same data multiple times.

4. **Cache storage** is primarily used to enhance performance by storing resources, such as images, CSS files, and JavaScript, in the browser so that they can be served quickly and efficiently without the need to re-download them from the server. This approach is crucial for Progressive Web Apps (PWAs) where offline functionality is important. By caching essential resources, you can provide users with a seamless experience even when they lose their internet connection or have slow connectivity.

## Use case: applying dark mode with localStorage

```
// Save user preference in localStorage
localStorage.setItem("theme", "dark");

// Apply theme on page load
if (localStorage.getItem("theme") === "dark") {
  document.body.classList.add("dark-mode");
}

// Add a toggle button
document.querySelector("#toggle-theme").addEventListener("click", () => {
  if (localStorage.getItem("theme") === "dark") {
    localStorage.setItem("theme", "light");
    document.body.classList.remove("dark-mode");
  } else {
    localStorage.setItem("theme", "dark");
    document.body.classList.add("dark-mode");
  }
});
```

HTML & CSS for dark mode implementation:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Dark Theme Toggle</title>
    <style>
    .dark-mode {
        background-color: black;
        color: white;
    }
    </style>
  </head>
  <body>
    <h1>Theme Toggle with Local Storage</h1>
    <button id="toggle-theme">Toggle Theme</button>
    <script src="script.js"></script>
  </body>
</html>
```

JSON plays a key role in web development, making data transfer and storage straightforward. Its simplicity and compatibility with JavaScript ensure it remains a top choice for developers. As you move forward, keep JSON in mind to enhance your web applications.

# Data privacy laws and compliance

In web development, handling user data involves technologies like JSON for efficient data exchange and Web Storage APIs (localStorage and sessionStorage) for client-side data persistence. However, these tools must be used responsibly to comply with data privacy laws, such as the General Data Protection Regulation (**GDPR**) in the European Union and the California Consumer Privacy Act (**CCPA**) in the United States.

For example, when using JSON to transfer user data between a client and server, or Web Storage APIs to store user preferences, developers must ensure transparency about data usage. This often includes gaining explicit user consent, especially for storing or processing sensitive information. Additionally, users should have the ability to review, delete, or restrict access to their data.

By aligning with regulations and incorporating robust privacy mechanisms, developers can effectively leverage tools like JSON and Web Storage APIs while maintaining user trust. For further information, refer to **GDPR Overview.**

## Cookie consent and example

A key requirement of GDPR and other regulations is obtaining explicit consent before storing or accessing cookies that track user activity, except for essential cookies. Developers typically implement a cookie consent banner that informs users about the types of cookies used and asks for their permission.

This code example demonstrates a simple cookie consent banner to comply with GDPR and similar regulations. It explains how websites can obtain and store user consent for cookies:

```
  <body>
    <!-- Cookie consent banner (hidden by default with "display: none") -->
    <div id="cookie-consent-banner" style="display: none;">
      <p>
        We use cookies to improve your experience. By continuing, you accept
our
        <a href="/privacy-policy">Privacy Policy</a>.
      </p>
      <button id="accept-cookies">Accept Cookies</button>
      <button id="decline-cookies">Decline</button>
    </div>

    <script>
      document.addEventListener("DOMContentLoaded", function () {
        // Check if the user has already given consent (stored in
localStorage)
```

```
        const consentGiven = localStorage.getItem("cookieConsent");
        const banner = document.getElementById("cookie-consent-banner");

        // If no consent is found, display the banner
        if (!consentGiven) {
          banner.style.display = "block"; // Show the banner
        }

        // When "Accept Cookies" is clicked, save consent and hide the
banner
        document
          .getElementById("accept-cookies")
          .addEventListener("click", () => {
            localStorage.setItem("cookieConsent", "true");
            banner.style.display = "none"; // Hide the banner
           });

        // When "Decline" is clicked, save no consent and hide the banner
        document
          .getElementById("decline-cookies")
          .addEventListener("click", () => {
            localStorage.setItem("cookieConsent", "false");
            banner.style.display = "none"; // Hide the banner
          });
      });
    </script>
  </body>
```

The banner asks users for consent to use cookies and stores their choice (accept or decline) in the browser's `localStorage`. This ensures the site remembers their preference across visits.

What it does:

- When the web page finishes loading: Checks `localStorage` for a previous consent decision. If none is found, the banner is displayed.
- Accept/decline buttons: Clicking "Accept" or "Decline" saves the choice in localStorage and hides the banner.

The code ensures compliance with regulations like GDPR by showing a cookie consent banner to users who haven't yet made a decision. This approach provides a clear and user-friendly way to manage consent while respecting privacy preferences.

# Instructions

Read and run the accompanying examples files provided before doing the compulsory task to become more comfortable with the concepts covered in this task.

## Practical task

Create a web application for an e-commerce cart system that uses `localStorage`, `sessionStorage`, cache, and cookies to manage and store shopping cart data, product preferences, and session-related activities.

Follow these steps:

1. Create a basic HTML structure.
2. Build a simple web page with the following elements:
    - A product catalog section (with a few items like name, price, and add-to-cart button).
    - A shopping cart section that shows the user's selected items.
    - A button to clear the cart (empty cart functionality).
    - A section to display the total price of items in the cart.
3. Use `localStorage` for persistent cart data:
    - Store cart data across sessions:

      Whenever the user adds an item to the cart, save the cart data (including item names and quantities) in `localStorage`. This allows the cart data to persist even if the user closes the browser and returns later.
    - Display saved Cart:

      When the page loads, check if there are items in `localStorage`. If items exist, display them in the cart section. If the cart is empty, display a message like “Your cart is empty.”
    - Update Cart:

      Whenever an item is added or removed, update the cart data in `localStorage` so the cart remains updated even after page refreshes.
4. Store font preference in `sessionStorage`:

- Allow users to choose a font preference for the current session (e.g., Arial, Times New Roman, etc.). This could be from a dropdown menu.
- Save this preference in `sessionStorage` so it applies only during the session.
- When the page loads, check `sessionStorage` for the font preference. If it exists, apply the font preference to the page elements dynamically.

5. Implement caching for static resources:
    - Cache product images and static data:

      Cache static resources such as product images and CSS files so that these are loaded faster on subsequent visits, improving performance. Use the browser cache to reduce the need to reload these resources from the server.
    - Notify users of cached data:

      Display a message or indicator showing that the page is using cached product images or styles to speed up loading. This lets users know the benefit of caching.
6. Use cookies for user login and preferences:
    - Store username in Cookies:

      When a user logs in (or enters their name), store their name in a cookie so that it persists across different sessions. This allows you to greet the user with their name when they return.
    - Track cart preferences in cookies:

      Optionally, store small preferences in cookies (such as the user's preferred currency or shipping method) to personalize their shopping experience. This should persist across sessions.
7. Clear cart and reset preferences:
    - Clear cart with `localStorage`:

      Implement a button that allows users to clear the cart. Ensure this button clears the localStorage (for persistent cart data). Also, ensure that cookies are deleted if the user opts to reset their preferences.
8. Final summary of cart and user information:
    - Display total price:

      Calculate the total price of all items in the cart and display it dynamically as users add or remove items. Update the cart data in `localStorage` accordingly.
    - Greet user:

On loading the page, check the cookies for the user's name and greet them with a personalised message (e.g., "Welcome back, [User]!"). This data should be fetched from cookies and used to personalize the shopping experience.

**Important:** Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

## Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.