

Thursday, 18 April 2024

local Storage, Promises

What is Local Storage?

Local Storage is a web browser feature that allows web applications to store data locally within the user's browser.

It provides a simple key-value store that persists even after the browser is closed and reopened, unlike session storage which is cleared when the browser session ends.

Why Use Local Storage?

Local Storage is commonly used for storing user preferences, application settings, and small amounts of user data that need to persist across browser sessions.

It provides a convenient and efficient way to store data client-side without relying on server-side databases or cookies.

Using Local Storage:

Local Storage is accessed through the `localStorage` object, which is available in the global scope of JavaScript.

Data is stored as key-value pairs, where both the key and the value are strings.

Common methods for interacting with local storage include **`setItem()`**, **`getItem()`**, **`removeItem()`**, and **`clear()`**.

Promises in Js

Promises are a fundamental feature of asynchronous programming in JavaScript, introduced in ECMAScript 6 (ES6).

They provide a cleaner and more intuitive way to handle asynchronous operations compared to traditional callback-based approaches.

Asynchronous Operations:

Asynchronous operations in JavaScript include fetching data from a server, reading files from disk, or waiting for user input.

Promises are particularly useful for handling asynchronous operations that involve waiting for a result to be available.

State:

A Promise represents the eventual completion or failure of an asynchronous operation.

Promises have three possible states:

Pending: Initial state, neither fulfilled nor rejected.

Fulfilled (Resolved): The operation completed successfully.

Rejected: The operation failed or encountered an error.

Chaining:

Promises can be chained together using `.then()` to handle the result of a resolved promise or `.catch()` to handle errors.

Chaining allows for more readable and concise code compared to nested callbacks

Creating a Promise:

A Promise is created using the `new Promise()` constructor, which takes a function as an argument.


```
javascript Copy code  
  
const myPromise = new Promise((resolve, reject) => {  
  // Asynchronous operation  
  setTimeout(() => {  
    const randomNum = Math.random();  
    if (randomNum < 0.5) {  
      resolve(randomNum); // Resolve the Promise with a value  
    } else {  
      reject(new Error('Random number is greater than or equal to 0.5')); // Reject  
    }  
  }, 1000);  
});
```

This function, called the executor function, receives two parameters: resolve and reject, which are functions used to fulfill or reject the Promise.

Consuming a Promise:

- Promises are consumed using the `.then()` and `.catch()` methods to handle successful resolution or rejection, respectively.

javascript

 Copy code

```
myPromise.then((result) => {  
  console.log('Promise resolved with value:', result);  
}).catch((error) => {  
  console.error('Promise rejected with error:', error.message);  
});
```