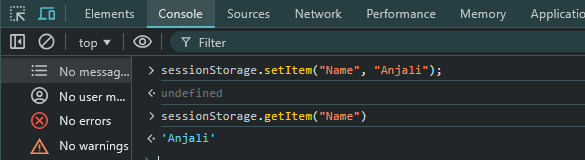
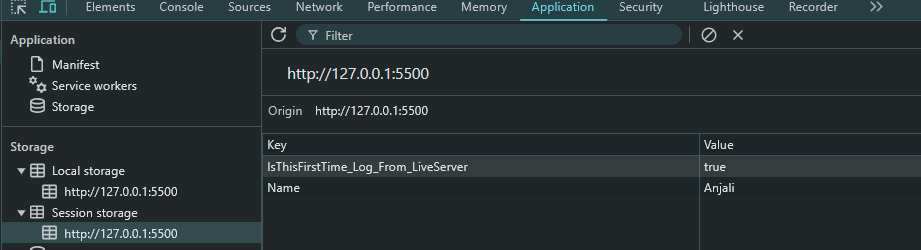
**Local Storage & Session Storage**

Both **Local Storage** and **Session Storage** are used for storing data on the client-side. However, they differ in terms of lifespan, scope, and purpose.

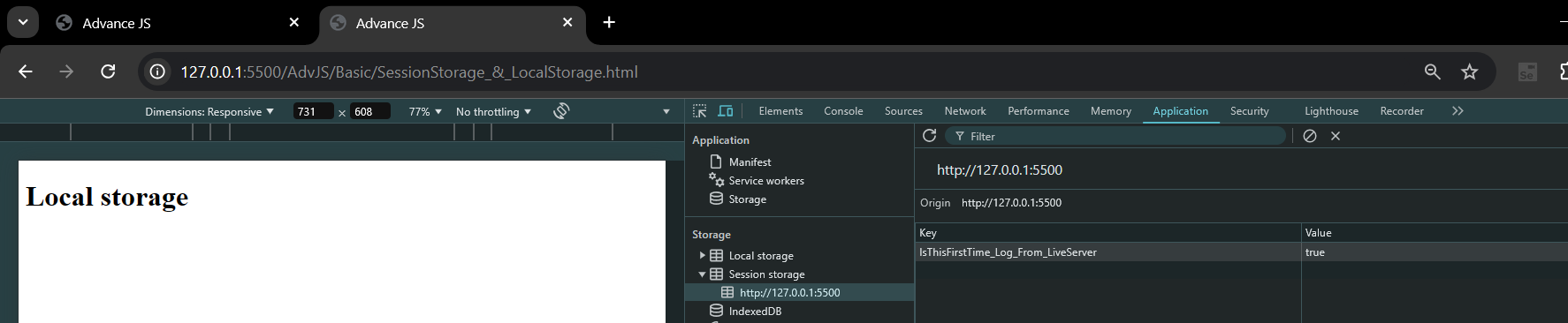
**Session Storage:**

* **Purpose**: Stores data for the duration of a session.
* **Session**: The time period from when a user opens a browser tab and interacts with a web application until the tab is closed.
* **Lifespan**: Data is cleared automatically when the tab is closed, meaning the storage is session-specific.
* **Advantages**:
  + **Not sent to the server**: Unlike cookies, session storage does not send stored data to the server during network requests, making it more efficient for storing client-side data that does not need to be sent with each request.
  + **Larger storage capacity**: Typically has more capacity than cookies. While cookies can store a maximum of about 4KB, session storage can hold at least 5MB.
* How to use Session storage:





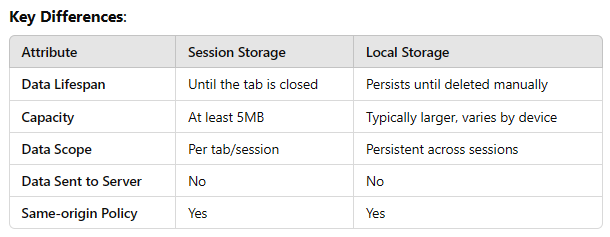
* **Problem:** If we switch to new tab then we lost our data store in Session Storage



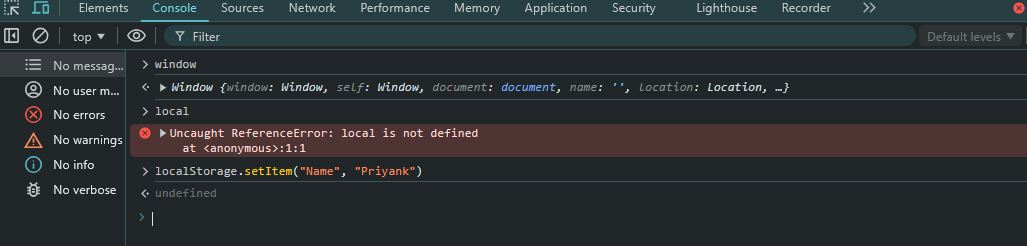
* Why we use Session storage?
  + For trigger **Session Events** like storage event, exception event or connection events etc.
    - Ex: session.addEventListener() etc.

**Local Storage:**

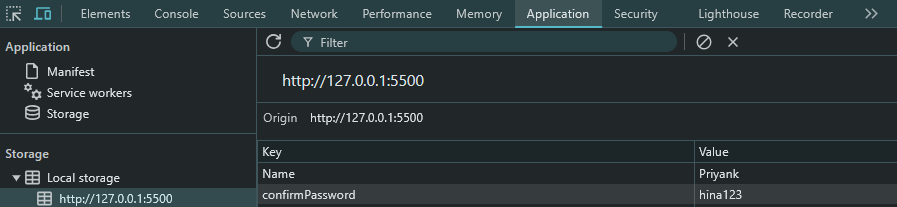
* **Purpose**: Similar to session storage, but data persists even after the browser or its tabs are closed.
* **Lifespan**: Data is retained until explicitly deleted by the user or the application (there is no automatic expiration).
* **Advantages**:
  + **Persistent storage**: Data is not cleared automatically, making it suitable for storing long-term data that does not need to expire.
  + **Higher capacity**: Local storage typically offers more storage space than session storage, with at least 5MB on mobile devices. Actual capacity may vary depending on the device.
  + **Faster access**: Fetching data from local storage is faster than making a network request, leading to better web application performance.
  + **Same-origin policy**: Local storage adheres to the same-origin policy, meaning data is only accessible from pages sharing the same origin (combination of protocol, host, and port). For example:
    - If data is stored from www.google.com, it will also be available on www.google.com/myaccount because they share the same origin.



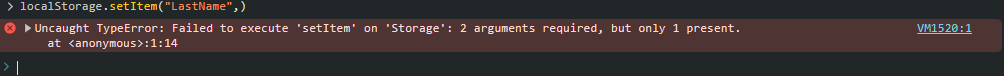
How to use local storage for set data?



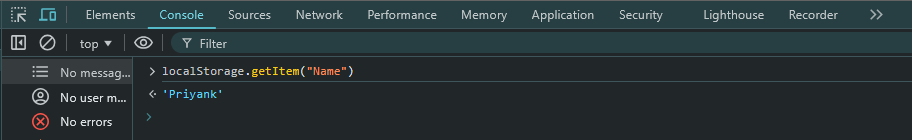
Result:



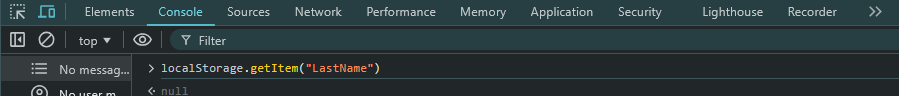
* + - Result will store in form of key value pair
    - If we not change key name and change value part in code then it will be overwritten and if we change key name then new key-value pair will be generated.
  + For store data into local storag**e: setItem()**
    - It must have 2 args(key, value), o/w it will give ERROR.

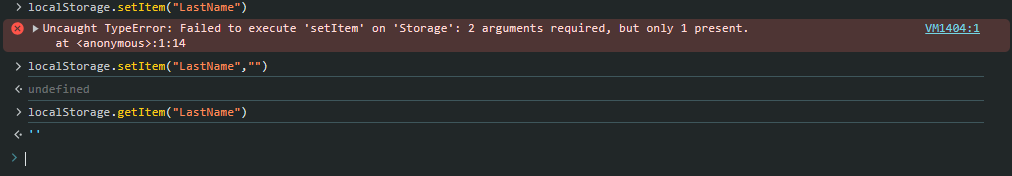


* + For fetch/access data from local storage: **getItem(“<key name>”)**

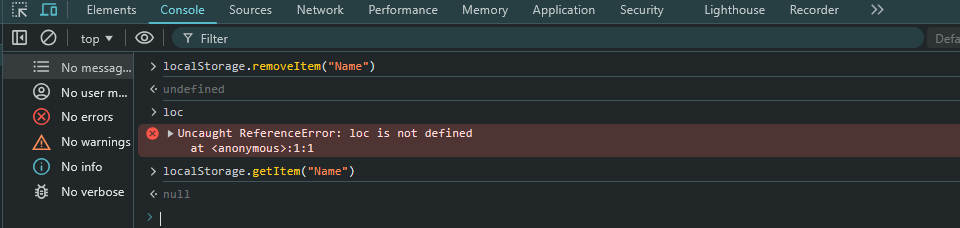


* + - When input key value is not present in local storage it will give output: **null**

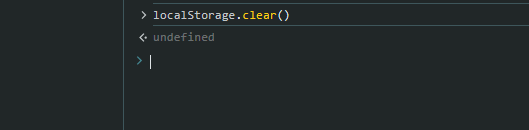


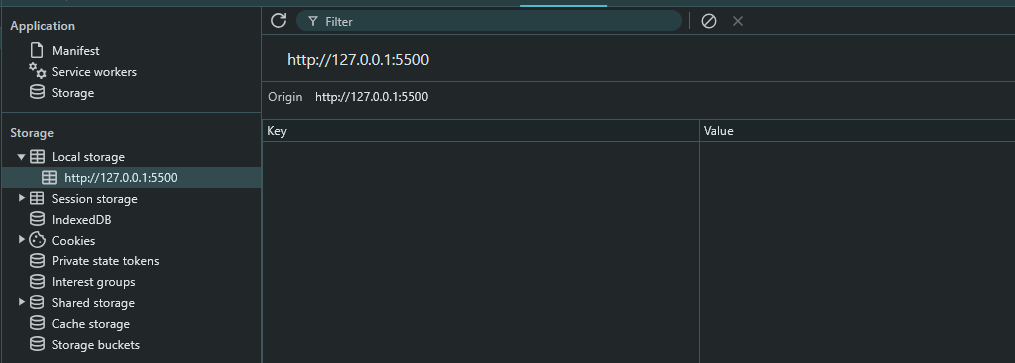


* + For remove particular data from local storage using key value: **removeItem(“<key name>”)**

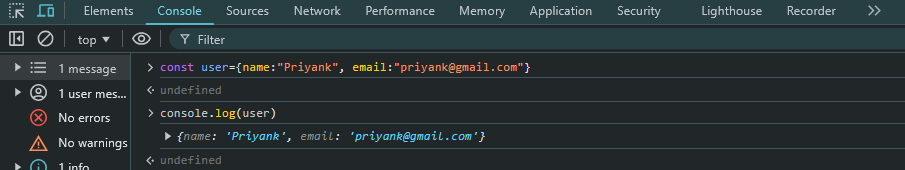


* + For Clear local storage: **clear()**





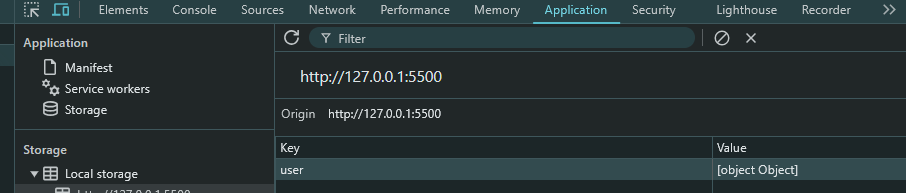
Local Storage just accept only String but if we want to store OBJECTS then….



Try to store object as key-value pair…



Output…



**Problem: We not get proper value of object**

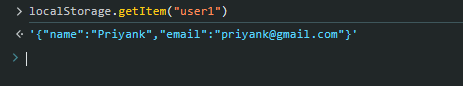
* + - **Because, Local storage only stores string values so when it gets any data with different datatype then it will first convert that data into String and then store it.** 
      * **But in case of object like data types it will not able to properly convert data and lost actual data …**

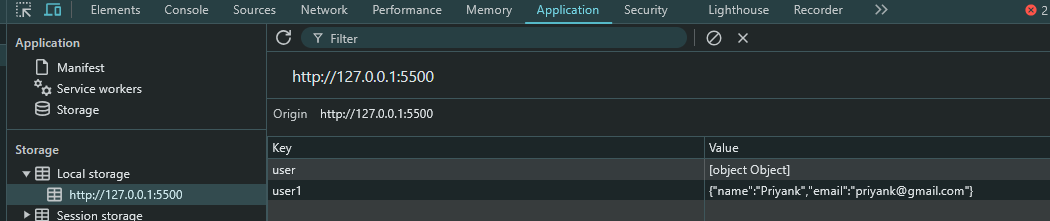
**Solution:**

* + - **JSON.stringify(<obj name>); 🡪 use this at the place where we write object as value.**

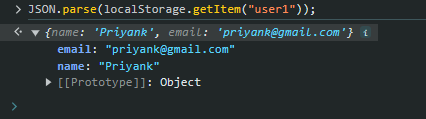
****

**Output:**

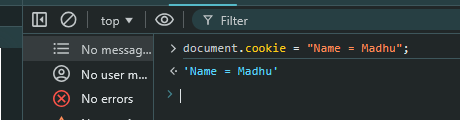
****

****

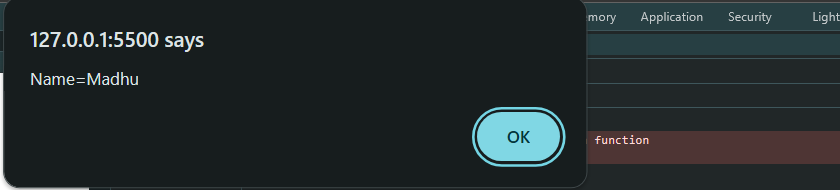
**For fetch actual object from local storage:**

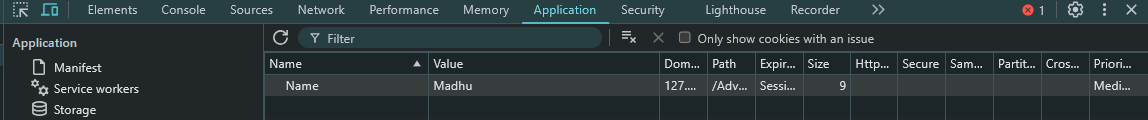
****

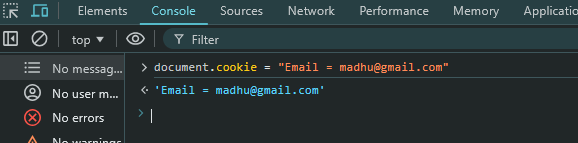
**Cookies:**

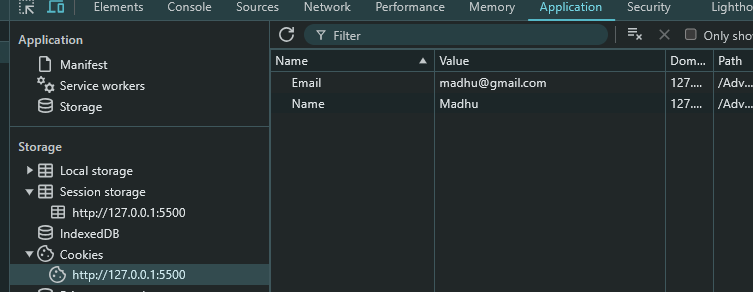
* **Purpose**: It is small strings of data which directly stores into browser.
* **document.cookie():** It will give access of all cookies ( key – value pair)
* Cookie sets by web browser using : **set-cookie HTTP header,** which useful for get user information when user visits same webApp second time.
* **Add cookie:**
  + **Use: document.cookie() :** It has one special feature that it will not overwrite all cookie when we set new cookie but instant of this it will add new cookie.

****

****

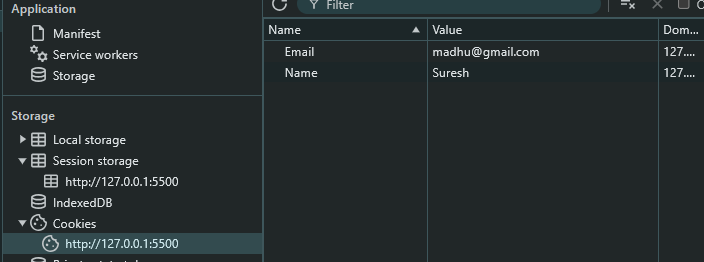
****

****

****

* **For Update existing cookie:**



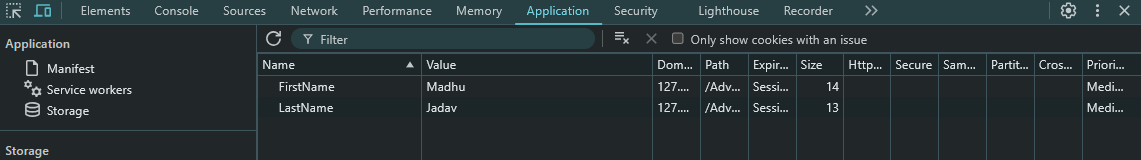


**Code for insert cookie every time browser is reloaded**

let key = prompt("Enter cookie key");

let value = prompt("Enter cookie value");

document.cookie = key + "=" + value;

****

let key = prompt("Enter cookie key");

let value = prompt("Enter cookie value");

// document.cookie = key + "=" + value;

// encodeURIComponent() : encode value of cookies key-value before store cookie, For improve security

// decodeURIComponent() : decode value of cookies key-value after store

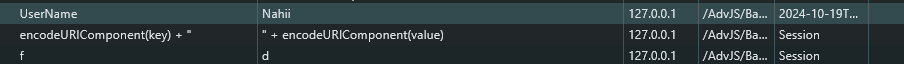
// Store the encoded key-value pair

document.cookie = `${encodeURIComponent(key)}=${encodeURIComponent(value)}`;

// This will output the decoded value in the console (browser decodes it automatically when reading cookies)

console.log(document.cookie);

* + We can set particular page path on which cookies are available.
    - Ex: show some data to only admin panel not to user.
  + We can also set cookie expiration time
    - Ex: store users’ data up to 7 days after login
* // set cookie expire time.
* document.cookie = "UserName = Nahii; expires=Sat, 19 Oct 2024 09:35:00 GMT";



**Caching:**

* Caching is a technique used to store data temporarily, allowing faster access to data by avoiding repeated fetches from the database.
* It increases speed and improves performance by reducing the need to query the database for the same information multiple times.
* When data is fetched for the first time, it is stored in a caching layer, so subsequent requests can retrieve the data from the cache rather than the database.
* Caching is fast because data is typically stored in primary memory (RAM), which is much quicker than accessing disk-based storage or remote databases.
* It reduces API and network calls.

**Example:**

* In a web application, when showing a user's profile (which may include personal information, payment details, etc.), multiple APIs may be called to fetch this information. If these APIs are called every time a user views their profile, it can slow down the application and increase the load on the server.
* Instead, after fetching the data from the database and APIs the first time, the information is stored in the cache (RAM). Future requests can retrieve the data from the cache, speeding up access and reducing server load.
* const USER\_API = "https://api.github.com/users/HinaJadav";
* async function handleUserData() {
* // Error handling
* try {
* const data = await fetch(USER\_API);
* const jsonData = await data.json();
* console.log(jsonData);
* } catch (error) {
* console.log(error);
* }
* }
* handleUserData();
* // o/p:
* // {
* //   login: 'HinaJadav',
* //   id: 105739881,
* //   node\_id: 'U\_kgDOBk12aQ',
* //   avatar\_url: 'https://avatars.githubusercontent.com/u/105739881?v=4',
* //   gravatar\_id: '',
* //   url: 'https://api.github.com/users/HinaJadav',
* //   html\_url: 'https://github.com/HinaJadav',
* //   followers\_url: 'https://api.github.com/users/HinaJadav/followers',
* //   following\_url: 'https://api.github.com/users/HinaJadav/following{/other\_user}',
* //   gists\_url: 'https://api.github.com/users/HinaJadav/gists{/gist\_id}',
* //   starred\_url: 'https://api.github.com/users/HinaJadav/starred{/owner}{/repo}',
* //   subscriptions\_url: 'https://api.github.com/users/HinaJadav/subscriptions',
* //   organizations\_url: 'https://api.github.com/users/HinaJadav/orgs',
* //   repos\_url: 'https://api.github.com/users/HinaJadav/repos',
* //   events\_url: 'https://api.github.com/users/HinaJadav/events{/privacy}',
* //   received\_events\_url: 'https://api.github.com/users/HinaJadav/received\_events',
* //   type: 'User',
* //   user\_view\_type: 'public',
* //   site\_admin: false,
* //   name: 'Hina Jadav',
* //   company: null,
* //   blog: '',
* //   location: null,
* //   email: null,
* //   hireable: null,
* //   bio: 'Perusing Computer Engineering From DDU  \r\n',
* //   twitter\_username: null,
* //   public\_repos: 20,
* //   public\_gists: 0,
* //   followers: 2,
* //   following: 3,
* //   created\_at: '2022-05-17T16:42:38Z',
* //   updated\_at: '2024-10-20T21:51:05Z'
* // }

**Types of Caching:**

1. Local / Memory Cache:
   * This type of caching stores data in the memory of the local machine where the application is running. It provides fast access but is limited by the available memory on the machine.
2. Distributed / External Cache:
   * In this type, the cache is stored on an external system, which may be shared among multiple application servers. This is used in distributed environments where scalability and fault tolerance are required.
3. Application Server Cache:
   * This involves caching data that is fetched from the database or API/network calls at the application server level. This reduces the need to access the database or make API calls repeatedly.
4. CDN (Content Delivery Network):
   * CDNs are used to cache data on a global scale, especially for delivering content (like images, videos, or static files) quickly to users across different regions. They store data on servers closer to the end-users to reduce latency when accessing content internationally.

Caching == Memosation

**OOJS:**

* JS is basically “Prototype based language”
* The class feature is introduced in advance JS part but at the end JS works as prototype-based language.
* **Object:** collection of properties and methods.
* Part of OOP:
  + Construction function
  + Prototype
  + Classes
  + Instance (new, this)
* 4 Pillers of OOP:
  + Abstraction
  + Encapsulation
  + Inheritance
  + Polymorphism
* Why use OOP?
  + Using OOP, we can enhance and use more advance features of JS
* Classes:
  + Special type function in JS.
  + Two ways to define classes:
    - 1) class expression
    - 2) class declaration
  + The body of a class is executed in [strict mode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode) even without the "use strict" directive.
  + **Constructor** a special method for creating and initializing an object created with a class.
    - **SystaxError:** when one class has more than one occurrence of constructor keyword
    - It uses “super” keyword to call the constructor of super class.
  + **Static initialization Blocks:**
    - Flexible initialization of static properties
  + We can declare static fields and properties using “static” key word.
  + For create private fields and methods we use: #
    - We need to create private properties in initial phase of their declaration.
* **Callbacks** 
  + A **callback** is a function that is passed as an argument to another function and is executed after that function completes. It allows asynchronous behaviour, where code can continue executing while waiting for a particular task to finish.
* **Issues with callback**
  + **Callback Hell**
  + Callback hell occurs when multiple asynchronous functions are nested inside each other in such a way that the code becomes difficult to read and maintain. It happens when you need to perform multiple asynchronous operations one after the other, and each operation depends on the result of the previous one.

### ****Inversion of Control (IoC) in JavaScript****

* + Inversion of Control (IoC) refers to a design principle where the control of flow (or logic) is reversed compared to traditional procedural programming. In simpler terms, instead of a calling function directly dictating what happens step-by-step, control is passed to an external function, framework, or library. This principle is often used with **callbacks**, **event listeners**, **dependency injection**, and **frameworks**.
  + **Callback Example and IoC:**
    - In JavaScript, **callbacks** introduce a form of IoC because we pass the control of a function to another function, typically provided by the environment (like the browser or Node.js). The callback function's execution is no longer directly under the control of the original calling code.
* Promises:
  + It is use to manage async operation in JS

Async & Await:

* **async**: It is a keyword which use before create asynchronous function.
* Async function always returns a **promise.** If you return any other data types then this function wraps that value into promise object and then return.