## **Important Table**

| **Details** | **Links** |
| --- | --- |
| Live Working Application**\*** | https://js-capstone-kohl.vercel.app/ |
| Current State of Application | https://js-capstone-seven.vercel.app/ |
| GitHub Repo Link | https://github.com/prashantCuvette/JS-Capstone |
| Verify Git Installation | git --version (2 times hyphen) |
| Verify node Installation | node --version (2 times hyphen) |
| Verify json-server Installation | json-server --version ((2 times hyphen) |
| Command to Install json-server | npm i -g json-server |
| Command to Run JSON Server | json-server --watch db.json --port 3000 |

## **\*** It will only run if you start the json-server in your localhost and the port number is 3000 or else will give some kind of error. In order to install json-server, node js must be installed first.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## **Browsers Storage**

Browser storages are built-in ways for web browsers to save data on a user’s device. They help websites remember information, even if the page is closed or the internet connection is lost. Here are the main types:

### **Cookies**

Cookies are small pieces of data that a website stores on a user’s browser to remember information about that user. They help websites keep track of things like login sessions, user preferences, or items in a shopping cart.

When a user visits a website, **the server can send a cookie** along with the web page. This cookie gets saved in the browser and is **sent back to the server with every request** to the same site, allowing the server to recognize the user. Cookies usually have **a name, a value, an expiry date**, and optional settings like domain and path. Some cookies expire when the browser closes, while others can stay for days, months, or even years if they have a set expiry time.

Cookies can be created and managed in different ways. On the server side, they are usually set using the Set-Cookie HTTP header. For example, when a user logs in, the server sends a response header like Set-Cookie: sessionId=abc123; Expires=Wed, 01 Jul 2025 12:00:00 UTC; Path=/. This tells the browser to store the cookie with the given name and value until the expiry date. On the client side, developers can also create and read cookies using JavaScript with the document.cookie property. By setting document.cookie = "username=John; expires=Wed, 01 Jul 2025 12:00:00 UTC; path=/", a script can save a new cookie. To read cookies, JavaScript can simply access document.cookie, which returns all cookies as a single string, and then parse it as needed. Cookies are useful but have size limits and security considerations, so they should be used wisely for storing only small, non-sensitive data.

Cookies are best used for storing ***small pieces of data*** that the server needs to know with every request. The most common and best uses are:

**1. Session Management:** Cookies keep users logged in by storing session IDs. When you log in, a session cookie tells the server who you are each time you load a new page.

**2. User Preferences:** Cookies remember settings like language, theme, or items in your cart so that your choices stay the same when you come back

**3. Tracking and Analytics:** Cookies help websites know how users navigate the site. They store unique IDs to track visits, clicks, and actions for ads or stats.

Cookies are ***not good for large data or sensitive information*** like passwords because they are small in size (about 4 KB) and can be seen or changed by the user. So, the best use is to store simple IDs or preference flags that the server or site needs regularly.

### **Local Storage**

**LocalStorage** is a type of browser storage that lets a website save data in a user’s browser so that the data stays there even after the user closes the tab or browser. It works as a simple **key-value** store — you save data with a name (key) and a value (**which is always a string**). Unlike cookies, LocalStorage **does not send this data to the server with every request**; it stays only in the browser. It’s useful for storing larger data than cookies, like user settings, theme choices, or temporary form data. LocalStorage has a bigger size limit too — usually around 5–10 MB per website.

LocalStorage is very easy to use with JavaScript. It has built-in methods to add, read, update, and remove data. You use ***localStorage.setItem("key", "value")*** to save something. To read it later, you use ***localStorage.getItem("key")***.

If you want to remove a specific item, you use ***localStorage.removeItem("key")***. To remove everything at once, you use ***localStorage.clear()***. The data you save stays there until you or the user clears the browser storage or runs **clear()**. LocalStorage works only for the same domain, so other websites cannot access it.

It’s good for saving non-sensitive information because anyone can see it in the browser’s DevTools. It’s best used for things like dark mode, user preferences, or data you want to keep between visits but don’t need to share with the server.

### **Session Storage**

SessionStorage is another type of browser storage that lets a website save data in the user’s browser for **just one session**. Like LocalStorage, it stores data as **key-value pairs**, and the value is **always a string**. The main difference is that SessionStorage data lasts only as long as the **browser tab or window stays open**. When the user closes the tab, all SessionStorage data for that page is deleted automatically. This makes it perfect for storing temporary information that you don’t need after the session ends.

SessionStorage is easy to use in JavaScript. You use s**essionStorage.setItem("key", "value")** to save something. To get it back, you use **sessionStorage.getItem("key")**. If you want to remove just one item, you use **sessionStorage.removeItem("key")**.

To clear everything in the session storage, you use **sessionStorage.clear()**. SessionStorage does not send data to the server, so it stays only in the user’s browser and works only for the same page origin.

It’s useful for things like storing temporary form data, page state, or steps in a multi-page form that don’t need to last after closing the tab.

Just like LocalStorage, anyone can see the data in the browser’s DevTools, so it should not hold sensitive information. It’s best for short-lived data that helps improve the user’s experience during one visit.

### **Indexed DB**

IndexedDB is a more powerful type of browser storage that works like a mini database inside the browser. Unlike cookies, LocalStorage, or SessionStorage, which store only simple key-value pairs as strings, IndexedDB can store **large amounts of structured data**, including objects, arrays, and even files or images. It’s designed for complex web applications that need to save lots of data and work offline, like email clients, note-taking apps, or document editors.

IndexedDB works in a different way because it’s an **asynchronous** database. You don’t just call **setItem** and **getItem** — instead, you open a database, create an object store (like a table), and then add, read, update, or delete records using transactions.

It uses JavaScript events and promises to handle data without freezing the page. For example, you first open a database with **indexedDB.open("myDatabase", 1)**. If it’s a new database or version, you create object stores inside the **onupgradeneeded** event. To add data, you open a transaction and use methods like **add** or **put**. To read data, you use **get** or **getAll**.

IndexedDB keeps data until you or the user deletes it, so it’s **persistent** and can store much more than LocalStorage — often hundreds of MBs or more. It works only for the same origin and does not send data to the server. It’s very useful for storing things like user-generated content, offline files, or caching large API responses for fast loading. It’s a bit more complex than other storage types, but it’s very powerful for modern web apps that need to work smoothly without an internet connection.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## **JSON Server**

JSON Server is a simple tool that lets you create a **fake REST API** very quickly using a JSON file. It is mainly used for testing and prototyping front-end or full-stack applications when you **don’t have a real backend** yet. Instead of writing actual server code, you just prepare a **db.json** file with some sample data, and JSON Server turns it into a working API with routes for GET, POST, PUT, PATCH, and DELETE requests.

### Internal Working Mechanism

You install JSON Server (usually with **npm install -g json-server**), create a **db.json** file (which looks like a database with JSON objects and arrays), and then run the server using **json-server --watch db.json**. It listens on a port (default is 3000), and you can make API calls to it just like you would with a real server.

For example, if your **db.json** has a **users** array, you can fetch all users with **GET /users**, add a new user with **POST /users**, update a user with **PUT** or **PATCH**, and delete a user with **DELETE /users/1**.

### Why Do We Use?

We use JSON Server mainly because it saves time during development. It allows front-end developers to build and test the UI and make real API calls without waiting for the backend to be ready. It’s great for learning, prototyping, demos, or testing how your app handles API responses and errors.

JSON Server is lightweight, easy to set up, and requires no complex backend setup — just a simple JSON file. However, it’s not meant for production because it’s a mock server with no real authentication or complex business logic. It’s best for local development and quick testing when you need an API but don’t want to build one from scratch yet.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## **Markdown File**

A Markdown file is a plain text file written using the Markdown language. Its file extension is usually **.md** (for example, README.md).

Markdown is a lightweight markup language that lets you add formatting (like headings, bold text, lists, links, images) using simple, readable syntax — no complex HTML needed.

### Why is Markdown used?

* Easy to write and read: You don’t need to learn heavy syntax. It’s almost like writing normal text with a few special symbols.
* Converts to HTML easily: Markdown can be quickly converted to formatted HTML for web pages, blogs, or documentation sites.
* Great for documentation: It’s very popular for project README files on GitHub, writing notes, creating wikis, and blogs.

### Importance of Markdown

* Clear documentation: Most open-source projects use a README.md to explain what the project does, how to install it, usage examples, and how to contribute.
* Version control friendly: Because Markdown is plain text, it works perfectly with Git. Changes are easy to track.
* Supported almost everywhere: GitHub, GitLab, Bitbucket, and many note-taking apps (like Obsidian or Notion) support Markdown.
* Portable: One file works on multiple platforms.
* Lightweight: No extra formatting tools or editors are needed.

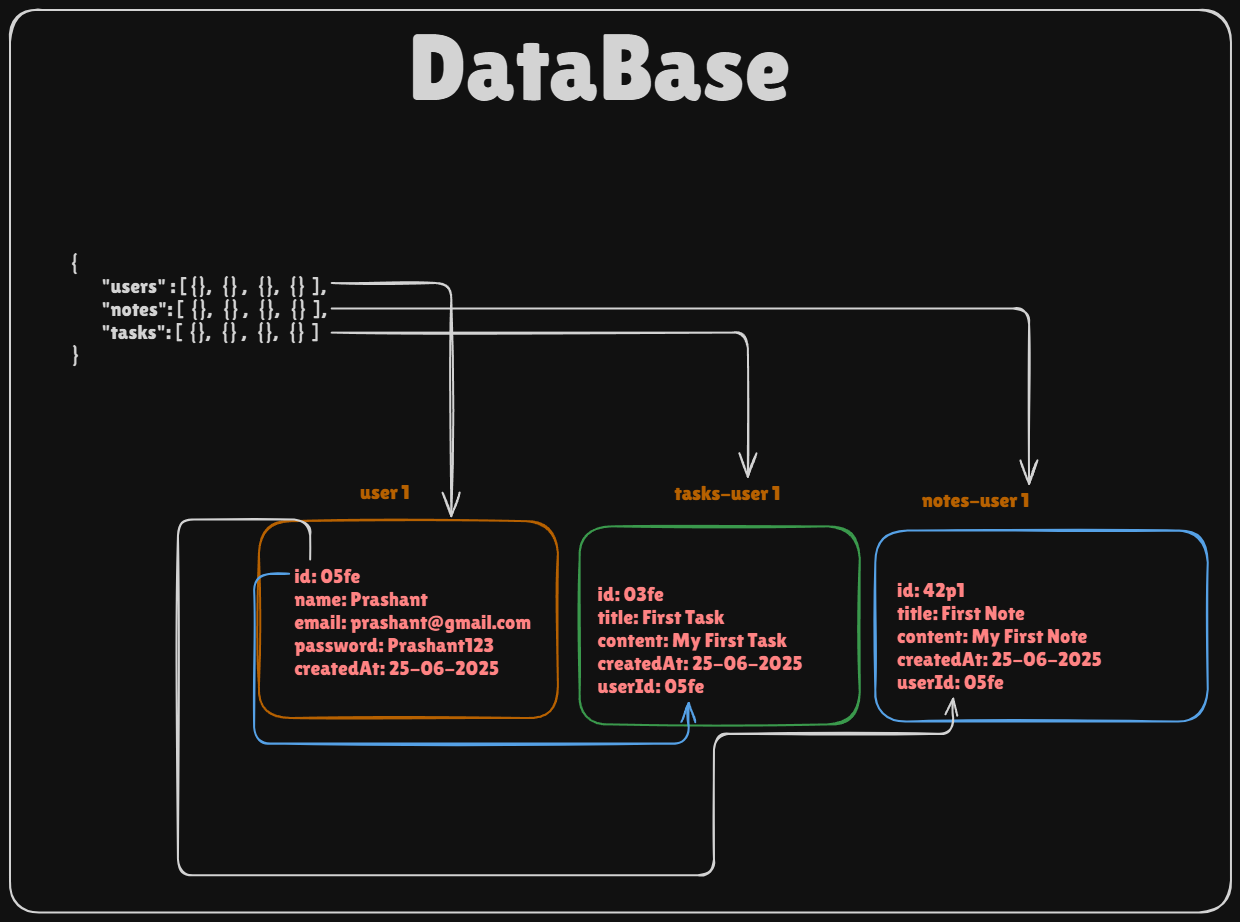
### Basic Syntax Examples

| # Heading 1  ## Heading 2  ### Heading 3  \*\*Bold text\*\*  \*Italic text\*  - Bullet list item  - Another item  1. Numbered list item  2. Second item  [Link to Google](https://www.google.com)  ![Image alt text](image\_url)  `Inline code` |
| --- |

### **Markdown CheatSheet** [ [Link](https://www.markdownguide.org/cheat-sheet/) ]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## **Structure of the DataBase**



### Single Entry in DataBase



## **Working of Important Operators**

### Falsy Values

*false, 0, -0, 0n, “”, null, undefined, NaN*

### Ternary Operator

Syntax: testCondition ? if true : if false;

### && Operator Examples

| true && true // true  true && false // false  false && true // false  false && false // false | 'hello' && 42 // 42  0 && 'something' // 0 (stops at first falsy)  true && 'done' // 'done'  undefined && 'ok' // undefined  null && 10 // null |
| --- | --- |

### || Operator Examples

| true || false // true  false || false // false  true || true // true | '' || 'fallback' // 'fallback'  false || 5 // 5  0 || null || 'hi' // 'hi'  null || undefined // undefined |
| --- | --- |

### && and || Operator Combined Examples

| true || false && false // true  false && true || true // true  false || (false && true) // false  (true && false) || 'ok' // 'ok' |
| --- |

### Use Cases

#### **1. Default Value with ||**

let name = inputName || 'Guest'; // fallback if inputName is falsy

#### **2. Conditional Execution with &&**

isLoggedIn && showDashboard(); // only runs if isLoggedIn is true

## **Understanding Promise Flow**

| const myPromise = new Promise((res, rej) => {  console.log("Promise Body Start");  let flag = true;  if(flag) {  res("Resolved");  } else {  rej("Rejected");  }  console.log("Promise Body End");  });  function handleMyPromise1() {    console.log("Before Resolved");    myPromise  .then((data) => console.log(data))  .catch((err) => console.log(err))    console.log("After Resolved");  }  // handleMyPromise1();  async function handleMyPromise2() {  console.log("Before Resolved");    const data = await myPromise;  console.log(data);    console.log("After Resolved");  }  // handleMyPromise2(); |
| --- |

Copy the code and try to run it in any compiler. First time use *handleMyPromise1()* to resolve and second time use *handleMyPromise2()* to resolve the same promises by commenting and un-commenting.

Take a closer look at the output screen in both cases you will find a difference. Explore by yourself why is it so?

You will find that async-await is a bit more than just syntactic sugar.

## **Display Inline vs Display Block in CSS**

| **display: inline** | **display: block** |
| --- | --- |
| Takes only as much width as the content needs.  Does NOT start on a new line.  Cannot set width or height.  Examples: <span>, <a>, <strong> | Takes the full width of its container.  Starts on a new line.  You can set width and height.  Examples: <div>, <p>, <h1> |

## **Local Storage Methods**

*JSON.parse(): Converts from String to original format*

*JSON.stringfy(); Converts to String format*

| **Methods (All String Values)** | **Syntax** |
| --- | --- |
| localstorage.setItem(key, value) | localstorage.setItem(“name”,”Prashant”) |
| localstorage.getItem(key) | localstorage.getItem(“name”) |
| localstorage.removeItem(key) | localstorage.removeItem(“name”) |
| localstorage.clear() | Clears entire storage |
| localstorage.key(index) | localstorage.key(0) |

## **List of API’s End Points**

| fetch(`http://localhost:3000/users?email=${encodeURIComponent(email)}`) | To check whether a user exists or not based on email |
| --- | --- |
| fetch('http://localhost:3000/users', {  method: 'POST',  headers: { 'Content-Type': 'application/json' },  body: JSON.stringify({ username, email, password })  }) | Submit the data to backend if it is a new user |
| fetch(`http://localhost:3000/users?email=${encodeURIComponent(email)}` | Get the response for an existing user |

In the given code, the fetch() function is used to send a POST request to the URL http://localhost:3000/users. This is commonly done when we want to send data to a server, such as registering a new user. The first part of the code specifies the URL where the request should go, which in this case is the /users route on the local server running at port 3000.

The method: 'POST' part inside the fetch options tells the server that we want to send data to create something new, like saving a new user's details. POST is one of the HTTP methods, and it is used when the client wants to submit or store some data to the server. Unlike GET requests that only fetch data, POST requests send data in the body of the request.

Next is the headers section. Here, we use 'Content-Type': 'application/json' to inform the server that the data we are sending is in JSON format. This is important because the server needs to know what kind of data is coming so it can correctly parse and handle it. If we don’t specify this, the server might not understand the format of the incoming data and respond with an error.

Then we have the body section, which contains the actual data being sent. The data is first converted into a JSON string using JSON.stringify({ username, email, password }). This converts the JavaScript object into a string format that can be transmitted over the network. The object holds user details like username, email, and password that we want to save on the server.