**CLIENT-SIDE STORAGE:**

Client-side storage is an excellent way to quickly add performance gains to an application. By storing data on the browser itself, you can skip fetching information from the server every time the user needs it. While especially useful when offline like it allow users to save sites or documents for offline use. Client-side storage can be done with cookies, Local Storage (technically “Web Storage”), IndexedDB, and WebSQL (a deprecated method that should not be used in new projects).

**LOCAL STORAGE:**

Local Storage uses a key/value system for storing data. It is limited to storing only simple values but complex data can be stored if you are willing to encode and decode the values with JSON. In general, Local Storage is appropriate for smaller sets of data you would want to persist, things like user preferences or form data. Larger data with more complex storage needs would be better stored typically in IndexedDB.

Let’s begin with a simple form based example:

**<div id="app">**

**My name is <input v-model="name">**

**</div>**

This example has one form field bound to a Vue value called name. Here’s the JavaScript:

**const app = new Vue({**

**el: '#app',**

**data: {**

**name: ''**

**},**

**mounted() {**

**if (localStorage.name) {**

**this.name = localStorage.name;**

**}**

**},**

**watch: {**

**name(newName) {**

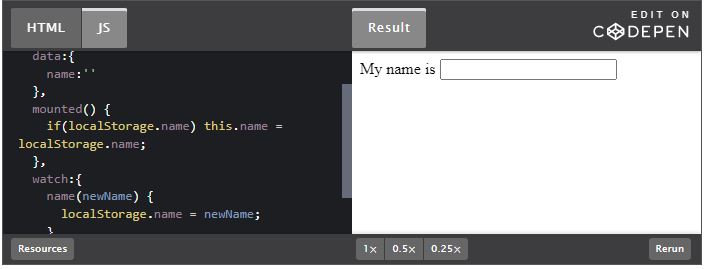
**localStorage.name = newName;**

**}**

**}**

**});**

Focus on the 'mounted' and 'watch' parts. Term 'mounted' is used to handle loading the value from localStorage. To handle writing the data base, we watch the 'name' value and on change, immediately write it.



Type something in the form and then reload this page. You’ll note that the value you typed previously will show up automatically.

**W**[**ORKING WITH COMPLEX VALUES**](https://v2.vuejs.org/v2/cookbook/client-side-storage.html#Working-with-Complex-Values)**:**

As mentioned above, Local Storage only works with simple values. To store more complex values, like objects or arrays, you must serialize and deserialize the values with JSON. Here is a more advanced example that persists an array of cats (the best kind of array possible).

**<div id="app">**

**<h2>Cats</h2>**

**<div v-for="(cat, n) in cats">**

**<p>**

**<span class="cat">{{ cat }}</span>**

**<button @click="removeCat(n)">Remove</button>**

**</p>**

**</div>**

**<p>**

**<input v-model="newCat">**

**<button @click="addCat">Add Cat</button>**

**</p>**

**</div>**

This “app” consists of a simple list on top (with a button to remove a cat) and a small form at the bottom to add a new cat. Now let’s look at the JavaScript.

**const app = new Vue({**

**el: '#app',**

**data: {**

**cats: [],**

**newCat: null**

**},**

**mounted() {**

**if (localStorage.getItem('cats')) {**

**try {**

**this.cats = JSON.parse(localStorage.getItem('cats'));**

**} catch(e) {**

**localStorage.removeItem('cats');**

**}**

**}**

**},**

**methods: {**

**addCat() {**

**// ensure they actually typed something**

**if (!this.newCat) {**

**return;**

**}**

**this.cats.push(this.newCat);**

**this.newCat = '';**

**this.saveCats();**

**},**

**removeCat(x) {**

**this.cats.splice(x, 1);**

**this.saveCats();**

**},**

**saveCats() {**

**const parsed = JSON.stringify(this.cats);**

**localStorage.setItem('cats', parsed);**

**}**

**}**

**})**

In this application, we’ve switched to use the Local Storage APIs versus “direct” access. Both work but the API method is generally preferred. ‘mounted’ now has to grab the value and parse the JSON value. If anything goes wrong here we assume the data is corrupt and delete it. (Remember, any time your web application uses client-side storage, the user has access to it and can modify it at will.)

You can play with this version below:

