Web Data Storage Example

by Sophia

WHAT'S COVERED

In this lesson, you will explore examples of the use of both sessionStorage and localStorage objects to store information locally on the user's browser. You will learn how to collect information from a web form and store it semipermanently on the user's system using localStorage. Next, you will learn how to implement sessionStorage to temporarily store shopping cart items. Lastly, you will learn how to connect the shopping cart code to the HTML user interface.

Specifically, this lesson will cover the following:

1. Local Storage Example
2. Session Storage Example
   - 2a. Implementing the Shopping Cart Modal
   - 2b. Implementing "Add to Cart" Buttons
   - 2c. Connecting Buttons to the Shopping Cart Modal

1. Local Storage Example

In this example, we will demonstrate using localStorage to store customer information on their local systems indefinitely. We will collect this information from a Contact Us form, load it into a JavaScript object, and then use JSON.stringify() to serialize the object data into a string for storage in localStorage.

MAKE THE CONNECTION

You will be expected to perform this type of operation in your final touchstone project.

First, we have the form itself located in the body of the webpage. This form includes a field for the user's name, email, phone number, and feedback message and a checkbox for whether or not this is a custom order. The "Reset" button is a default implementation of the input element using "reset" as the type. This creates a button that will simply reset the values back to their default. The "Submit" button will be a generic "button" input element with an onClicklistener, which points to a JavaScript function that will store the contents and produce a popup with data from the localStorage.

EXAMPLE The Contact Us form

```
<form autocomplete="on">
            <label>Name:
            <input type="text" name="name" id="name" /></label><br />
            <label>Email:
            <input type="email" name="email" id="email" /></label><br />
            <label>Phone:
            <input type="tel" name="phone" id="phone" /></label><br />
            <label>Feedback:
            <textarea name="feedback" id="feedback" ></textarea></label><br />
            <label>Custom Order:
            <input type="checkbox" name="customOrder" id="custom-order" /></label><br />
            <br />
            <input type="button" onclick="submitForm()" value="Submit" />
            <input type="reset" value="Clear Form" />
        </form>
```

Notice that the “Submit” button is an input element with a type of “button” and an onClick attribute pointing to the submitForm() function. The submitForm() function will extract the values entered into the fields and organize them into a JavaScript object.

EXAMPLE A partial view of the submitForm() function

```
<script>
function submitForm()
{
            const name = document.getElementById("name").value;
            const email = document.getElementById("email").value;
            const phone = document.getElementById("phone").value;
            const feedback = document.getElementById("feedback").value;
            const customOrder = document.getElementById("custom-order").checked;

            const customerInfo = {
                    name, email, phone, feedback, customOrder
            };

            const keyValue = name;
            //save customer information to localStorage
            localStorage.setItem(keyValue, JSON.stringify(customerInfo));

            //access and parse local data back out of localStorage.
            const who = JSON.parse( localStorage.getItem(name) );
            alert("Thank you for your message, "+ who.name +"!");
```

```
}
<script>
```

After we load the form data into the customerInfo object, we prepare a keyValue variable with the name value from the form to be used to set the local storage. The next line is a call to `localStorage.setItem( "key", "value" )`, wherein we pass in the keyValue of the user's name and the results of serializing customerInfo using `JSON.stringify()`.

At this point, the data from the form is loaded into the local storage on the user's system and is now accessible to the rest of the code. Remember that localStorage is accessible from other tabs and windows and will remain on the system until removed by the site's code or by the user clearing the browser cache.

The final two lines of the example function verify the use of localStorage to store the user's data by extracting the data back out of local storage and using part of the data in an alert message. To do this, we access `JSON.parse()` and pass in `localStorage.getItem(name)`, which retrieves the stringified user data and parses it into a JavaScript object called "who." We can then use the "who" object to access and use the data elsewhere, such as the alert thanking the user personally for the message.

---

2. Session Storage Example

In this example, we will use sessionStorage to store and manage the shopping cart's list of items. Since this is a function that would likely be used across the entire site, this would be a good opportunity to place the JavaScript code into an external .js file and simply link to it using a <script> tag. This way, each page will automatically attempt to look for and load cart items from sessionStorage and will have the ability to add new items to the cart.

EXAMPLE JavaScript sessionStorage checking for existing cart data

```
//Get the cart items from sessionStorage, or initialize as an empty array.
var cartItems = JSON.parse(sessionStorage.getItem("cartItems")) || [];
```

In this first line, we declare our cartItems array and attempt to fill it with any cart items that are currently in sessionStorage. The call to `sessionStorage.getItem("cartItems")` itself looks for a session object called "cartItems". If our code had previously stored cart items, it would have been under the key of "cartItems". If there are items already in sessionStorage, they will be parsed using `JSON.parse(sessionStorage.getItem("cartItems"))` and loaded into the new array.

If this call returns nothing or no "cartItems" key is found (because it had never been set), then the next section of code `|| []` will initialize the cartItems array as an empty array.

KEY CONCEPT

Assignment operations can be postfixed with the OR operator || (double pipe character) followed by a fallback value. The idea is that the system will attempt to evaluate the first expression to the right of the assignment operator =, and if this expression returns an error, the fallback value will be used. This is helpful when we need

to initialize a collection of possible preexisting data, and if that data does not exist, instead of an error, the fallback value will be used.

The above process will take place within the same browser tab and will occur each time the user navigates to another page that is linked to the shopping cart external script. Next will be the function used to add new items to the cart.

EXAMPLE JavaScript sessionStorage adding items to the cart

```
//function to add an item to the cart.
function addToCart(itemName)
{
    cartItems.push(itemName);
    sessionStorage.setItem("cartItems", JSON.stringify(cartItems));
    showMessage("Item added to the cart: " + itemName);
}
```

In the above section of code, the function is defined as “addToCart,” it accepts an item name as an argument, and it is stored in the itemName parameter. The first step within the function is to call the cartItems array’s push() function to add the name to the end of the array. Next, the cartItems session storage is updated using `sessionStorage.setItem("cartItems", JSON.stringify(cartItems))`. The updated array, cartItems, gets serialized and stored in sessionStorage under the key of "cartItems". The final step in the function is to show a message popup that confirms the item has been added to the cart. This message uses a custom-defined showMessage() function, which simply passes the message argument to a call to alert().

THINK ABOUT IT

What would be the purpose of wrapping a custom-defined function around a simple function such as alert()? Why not just call alert() directly when you need to?

There are multiple benefits to creating a wrapper function around alert(). The first benefit is that we can add logic, controls, and multiple optional parameters within the wrapper function to provide a single function that can carry out multiple types of messages for the user based on the presence or omission of arguments.

Secondly, this can help make our code more portable and somewhat **future-proof**. For example, what if we wanted to take this application project to a mobile platform that does not have an alert() function. This approach makes it very easy to adapt our application to another platform that may use a different function for popup system messages. Additionally, if JavaScript ever changed its alert() function, we would be able to easily update our code by modifying our showMessage() function.

The next section of code attaches an event handler function to any button that contains the id attribute value of “clear-cart-button.” The “click” event is used to trigger the anonymous function, which resets the cartItems array to an empty array and calls the `sessionStorage.removeItem("cartItems")` method to remove the cartItems stored in sessionStorage. Lastly, `displayCartItems();` is called, which clears the items from the screen and displays an empty cart. Lastly, showMessage is called to inform the user the cart has been cleared, and then the cart modal is closed. The modal functionality is not discussed in this tutorial section.

EXAMPLE JavaScript sessionStorage adding items to the cart

```
//event listener for the "clear cart" button.
var clearCartButton = document.getElementById("clear-cart-button");
clearCartButton.addEventListener("click", function () {
    cartItems = [];
    sessionStorage.removeItem("cartItems");
    displayCartItems();
    showMessage("Cart cleared");
    closeCartModal();
})
```

The last section of code is the custom-defined displayCartItems() function. This function's role is to populate the shopping cart with items. The idea here is that the shopping cart will have an HTML list element with an id value of "cart-items", which will be filled with newly created HTML list item elements <li>, each containing one of the cart item names.

EXAMPLE JavaScript sessionStorage adding items to the cart

```
//function to populate the shopping cart modal with items.
function displayCartItems()
{
    var cartList = document.getElementById("cart-items");
    cartList.innerHTML = "";

    if (cartItems && cartItems.length > 0)
    {
        for (var i = 0; i < cartItems.length; i++)
        {
            var listItem = document.createElement("li");
            listItem.textContent = cartItems[i];
            cartList.appendChild(listItem);
        }
    }
}
```

First, we get a handle to the HTML list using `getElementById("cart-items")` and storing it in the cartList variable. Next, we make sure this list's innerHTML is cleared. Then, we test if there even is a cartItems array and that it contains more than zero elements. Assuming we have the cartItems array and it contains items, we can then start a `for` loop to generate the list items. This loop creates a new HTML element using `document.createElement("li")`, fills its textContent attribute with one name from the cartItems array, and then appends the new HTML list item to the cartList, which effectively renders it to the screen.

With all of this code in place, the next steps will be to connect our shopping cart elements on the interface with the code in the external shopping cart code file.

One trick to understanding what needs to be done to the element on the webpage is to look for any function calls to document.getElementById and any variations of the “getElement” methods. Then, carefully add the proper id values to the correct buttons and other elements.

TERM TO KNOW

Future-Proof
A development approach or technique that helps ensure something will remain usable in the future and as other things change and evolve.

2a. Implementing the Shopping Cart Modal

In this tutorial section, we will demonstrate the implementation of a simple shopping cart modal screen and connect the various buttons to the JavaScript code to allow the user to add items to the cart, view and hide the cart, clear the cart, and process the order. We will start by adding the shopping cart HTML content to a page. Next, we will add CSS to the cart. This will ensure the cart is hidden when initialized and will set up the modal formatting so that the webpage gets a dark fade when the shopping cart is displayed. Lastly, we will tie it all together using JavaScript.

The HTML for the shopping cart includes an HTML division for the modal screen, and the second nested division will be for the shopping cart content itself. The content simply includes an X for the close button, a heading, an empty unordered list, and buttons for clearing the cart and processing the order.

EXAMPLE HTML modal content

```
<div id="cart-modal" class="modal">

      <div class="modal-content">
        <span class="close" id="close-cart-modal">×</span>
        <h3>Your Cart</h3>
        <br>
        <ul id="cart-items"></ul>
        <br>
        <br>
        <button id="clear-cart-button" class="cart-button">Clear Cart</button>
        <button id="process-order-button" class="cart-button">Process Order</button>
      </div>
</div>
```

Next, we will apply the CSS to the correct classes used above. We will use the outer division, with the “modal” class, to create the dark fade for the background. This div will take up the entire page with no margin and have a background color of black with 50% opacity.

EXAMPLE CSS for the modal background

```
/* The Modal (background) */
.modal {
  display: none;
  position: fixed;
  z-index: 1;
  padding-top: 100px;
  left: 0;
  top: 0;
  width: 100%;
  height: 100%;
  overflow: auto;
  /* Enable scroll if needed */
  background-color: rgba(0, 0, 0, 0.5);
  /* Black w/ opacity */
}
```

As you can see in the code above, the modal class is hidden using display: none; its position is fixed, flush to the left, and at the top, and it has a height and width of 100%. Also, overload is set to auto to allow for scrolling if needed, and the background-color uses the rgba() function to set a black color and 0.5 opacity. Notice the z-index of 1. The z-index is a CSS property that allows us to layer items behind or in front of other elements.

Now let's take a look at the .modal-content class; this class's styling is pretty straightforward.

EXAMPLE CSS for modal content

```
.modal-content {
  background-color: #fefefe;
  margin: auto;
  padding: 50px;
  border: 1px solid #888;
  width: 80%;
}
```

2b. Implementing "Add to Cart" Buttons

The last part in this setup is to program the "Add to Cart" buttons. Each of these buttons will have the same class attribute value of "add-to-cart-button" and a special attribute called the data-* attribute. This attribute allows us to attach additional data to an HTML element; furthermore, we get to customize the attribute by replacing the * with any name we desire. In this scenario, we will use the `data-item` at the attribute and give them values that can be used to identify the project chosen. This could be an id number assigned to the product or the product name. In this case, we are just going to use "Item 1," "Item 2," and so on to number the items.

KEY CONCEPT

When you call a function using an event, such as the click event for a button, the button element itself gets sent into the function as an argument. We can capture this element by putting a parameter in the called function. This is how we can then access the custom data element that we gave to the “Add to Cart” buttons.

EXAMPLE HTML code for the “Add to Cart” button

```
<button class="add-to-cart-button" data-item="Item 1">
```

EXAMPLE JavaScript code to add items to cart and attach the button

```
var addToCartButtons = document.querySelectorAll(".add-to-cart-button");

addToCartButtons.forEach(function(button) {
  button.addEventListener("click", function() {
    var itemName = this.getAttribute("data-item");
    addToCart(itemName);
  });
});

function addToCart(itemName) {
  cartItems.push(itemName);
  sessionStorage.setItem("cartItems", JSON.stringify(cartItems));
  showMessage("Item added to the cart: " + itemName);
}
```

We start with getting a handle to all of the “Add to Cart” buttons using document.querySelectorAll(".add-to-cart-button"), and we use the CSS class selector ".add-to-cart-button". This gets an array of handles to each of the buttons, so we can then attach the event listener to them using the forEach method. The forEach array method accepts a callback function as the only argument. The callback function takes each button as an argument and calls the button’s addEventListener() method to attach the button’s click event to an anonymous function. The anonymous function uses the this.getAttribute("data-item") to get the custom data from the button that was clicked and then passes the value to the addToCart() function. The addToCart() function pushes the value onto the cartItems array, updates the sessionStorage with the updated cartItems array, and then shows a message confirming the item was added to the cart.

One other command in the external JavaScript that is needed to initialize the list of cart items is to make a call to the displayCartItems(), which will either clear out the innerHTML of the cart list or initialize the list items from the cart items stored in sessionStorage. This command will be the last line in the external script file so that it gets executed after the above code is read and added to the DOM.

2c. Connecting Buttons to the Shopping Cart Modal

Next, we look at how the connection is made between the HTML content and the JavaScript code. Generally, the idea is that whether we write the HTML content first or the JavaScript, the connection can be made using id attributes and the getElementById() method. We can call the document.getElementById() and get a handle to and then attach an event listener to the element. The following buttons will be linked to the code:

- View Cart
- Add to Cart
- Clear Cart
- Process Order
- Close Cart

Each will have its own id value, which will be used to attach the event listener. Let's start with the "View Cart" button. This button element has the id of "view-cart-button", so the JavaScript will look like this:

EXAMPLE The JavaScript code for showing the cart modal

```
var viewCartButton = document.getElementById("view-cart-button");
var cartModal = document.getElementById("cart-modal");

viewCartButton.addEventListener("click", function() {
  openCartModal();
  displayCartItems();
});

function openCartModal() {
  cartModal.style.display = "block";
}

function displayCartItems() {
  var cartList = document.getElementById("cart-items");
  cartList.innerHTML = "";

  if (cartItems && cartItems.length > 0) {
    for (var i = 0; i < cartItems.length; i++) {
      var listItem = document.createElement("li");
      listItem.textContent = cartItems[i];
      cartList.appendChild(listItem);
    }
  }
}
```

Starting at the top, we get a handle to the "View Cart" button using the document.getElementById("view-cart-button") method and assign it to the viewCartButton variable. Next, we add an event listener to the button using the "click" event to call an anonymous function, which simply calls two different functions, openCartModal() and displayCartItems().

The openCartModal() function simply updates the CSS display property to "block," thus making the outer division for the cart visible. This causes the webpage to appear to fade dark thanks to the black background-color with 50% opacity and the shopping cart content being shown on top.

The displayCartItems() function is a utility function that gets a handle to the cart-items list element in the cart modal, clears the innerHTML attribute before testing if there are items in the cart array, and then starts a loop that fills the cart list with list items. The process of filling the list includes creating a brand-new list item element using document.createElement('li'), grabbing an item value from the array of cart items, and storing it in the textContent of the newly created list item before appending it to the cart list. This generates a list of the items that were added to the shopping cart.

Next, we handle the closeCart button:

EXAMPLE Setup of the “Close Cart” button

```
//Get the handle.
var closeCartButton = document.getElementById("close-cart-modal");

//Function to close the modal by changing CSS property.
function closeCartModal() {
  cartModal.style.display = "none";
}

//Attach event listener to trigger closeCartModal().
closeCartButton.onclick = function() {
  closeCartModal();
}
```

Again, we start by getting a handle to the close button, defining the function that will close/hide the modal, and then attaching the event listener by assigning an anonymous function to the button’s onclick attribute. The anonymous function simply calls the closeCartModal() function.

TRY IT

You will write the JavaScript code to clear the shopping cart of items by clearing the cartItems array and removing the items from session storage.

Directions: Open the script.js file that contains the JavaScript code for the site. Go to the bottom of the file and declare a new variable called clearCartButton; assign a handle to the element with the id of "clear-cart-button".

Next, use the clearCartButton to add an event listener. The event will be “click,” and the called function can be an anonymous function or a call to a traditional function.

The function will not take any arguments, and the algorithm for the body of the function will be as follows:

- Check IF the length of the cartItems array is not equal to 0.
- Then:
  - Set cartItems to an empty array.
  - Call the session storage removeItem() method and remove the artItems.
  - Update the shopping cart item list by calling the displayCartItems() function.

    - Show a message to the user indicating the cart has been cleared.
- Else:
    - Show a message to the user indicating that there are no items to clear.
- Close the shopping cart modal by calling the closeCartModal() function.

At this point, you should be able to test out the functionality by adding items to the cart, viewing the cart, and then clicking the “Clear Cart” button. Make sure you get the message that the cart has been cleared. Next, reopen the cart and make sure that the items are indeed gone.

- What to do if the message doesn’t appear stating that the cart was cleared?
    - Double-check that you used the correct id value when you called getElementById() and that the value is surrounded in quotes.
    - Double-check your syntax of the addEventListener function. The first argument should be “click” and not “onclick” and surrounded with quotes.
- What to do if the message appears, but the items still remain in the shopping cart modal or reappear after reopening the shopping cart?
    - If the message appears and the items do not go away, double-check that you are overwriting the contents of the correct cartItem variable with an empty set of square brackets.
    - If the message appears and the items disappear but then come back after refreshing the page, then double-check the call to the sessionStorage.removeItem() method and make sure that the argument is surrounded in quotes and that it matches the key "cartItems" used when creating and updating the session storage.

REFLECT

By completing this exercise, you practice real-world development by having to identify the steps that need to be carried out (in this case, these steps were given). Then, recall how to do each step from memory, look back to previous code that you wrote, or conduct research on sites like w3schools.com or developer.mozilla.org for guidance on how to accomplish the tasks in JavaScript. This is also a good opportunity to practice troubleshooting should something not work as expected.

The “Process Order” button is one that will be different depending on the payment system used by the organization. The base will be a button that shows a message if there are items in the cart to be purchased. It then clears the array of items, clears the cartItems from sessionStorage, and then makes a call to displayCartItems() and then closeCartModal().

SUMMARY

In this lesson, you learned how to utilize the **localStorage** object to capture and store user information on the user’s system. In particular, you saw how to take the data from a web form and load it into localStorage so that the site can use the data. You then learned how to manage shopping cart data using **sessionStorage**. Additionally, you saw how placing JavaScript code into an external file can then easily be applied to the multiple pages and provide a continuity of functionality across multiple pages.

Finally, you saw how to connect your HTML interface to the **shopping cart modal** code contained in the external file.

Source: This Tutorial has been adapted from "The Missing Link: An Introduction to Web Development and Programming " by Michael Mendez. Access for free at https://open.umn.edu/opentextbooks/textbooks/the-missing-link-an-introduction-to-web-development-and-programming. License: Creative Commons attribution: CC BY-NC-SA.

TERMS TO KNOW

Future-Proof

A development approach or technique that helps ensure something will remain usable in the future and as other things change and evolve.