WEB STORAGE

Excerpt from:
Learning Web Design, 6e

by Jennifer Robbins
with contributions by Aaron Gustafson

by Aaron Gustafson

AUTHOR's NOTE: This article originally appeared in **Chapter 24, Working with the DOM and Events** *in* Learning Web Design, 6e *and is written in a way that assumes you have read the beginning of that chapter. The exercises it refers to use JavaScript to place a button on the page that toggles between light to dark visual themes. However, even if you haven't read the chapter, this article is still a useful introduction to web storage.*

One drawback of the theme toggle button we've built so far is that refreshing the page in the browser resets the theme. That's not ideal. If a user chooses a particular theme, we should honor that and display their chosen theme when they return to the page (or, to use developer lingo, their choice should **persist**). In this article, we'll explore how to do that.

One of the long-time challenges of designing for the web is that, in isolation, web pages are **stateless**. That means when you request an HTML page from a given website, it's just an HTML page. It doesn't know who you are, what you added to your shopping cart, or what you've done previously. Each page load is a blank slate.

Of course, this isn't our experience of the web today. In fact, I'd argue the web would not have become what it is without us overcoming that fundamental challenge. On today's web, the state of the website—whether you're logged in, your account details if you are, the items in your shopping cart, etc.—is maintained in a few different ways.

Important things like your shopping cart are often maintained in a database of some sort on the **server side** of things, which is to say a computer run by the website's owner. Typically, the website puts a little bit of information in your browser—the **client side**—to connect your browsing experience to the information in the database. Often, that bit of information is stored in some-

thing called **cookies**. Cookies are small text files associated with the website's domain name that get sent to the server when you request a web page so the server knows who you are and how to retrieve your personal information. You've probably dismissed at least 200,000 cookie banners at this point in your life, so I'm betting you're at least somewhat familiar 🙃.

Whereas cookies work as a bridge between the client side and the server side, there are ways to store data purely in the browser. The easiest to work with—the `localStorage` and `sessionStorage` objects—are collectively referred to as **web storage**. You can think of these two behind-the-scenes browser features as shelving on which you can store labeled buckets containing values. The items we store in web storage aren't technically variables, but they act in a similar way—we'll get to what that means shortly.

These two storage options are nearly identical. The only difference between them is how long the data we put on their shelves sticks around. As you might expect, data in `sessionStorage` only exists during a user's session. As soon as the user closes their browser, it disappears. Data stored in `localStorage`, however, has staying power.

Web storage is great for persisting information the server really doesn't need access to. That could be something simple, like tracking whether a particular notification was dismissed so you don't show it again, or it could be complex, like storing form field values as the user types so you can repopulate the form if they accidentally close the tab before they hit "Submit." You could also use it to save which theme a user has chosen.

MEET LOCALSTORAGE

Working with `localStorage` is a bit like creating JavaScript variables. When we declare a variable, we give it a name and assign it a value. `localStorage` stores a collection of **key/value pairs**. The **key** is akin to a variable name, but it's a string. The **value** is the value you want to associate with that key. It's analogous to the value you assign to a variable, but it must also be a string (see the sidebar **"Storing Non-String Values in localStorage"**). Let's look a bit more closely at how `localStorage` works.

We store values using the `localStorage.setItem()` method. Here, I'm storing the string value "Hi!" for the key "greeting":

```
localStorage.setItem( "greeting", "Hi!" );
```

When I want to retrieve the value again, I call `localStorage.getItem()` and pass it the key (see **Note**):

```
localStorage.getItem( "greeting" ); // "Hi!"
```

NOTE

If the key you try to get is not found, the method returns a `null` *value.*

Storing Non-String Values in localStorage

With **localStorage**, all values are stored as strings. That means if you store a number value, it will come back to you as a string containing that number:

```
localStorage.setItem( "amount", 100 );
localStorage.getItem( "amount" ); // "100"
```

If you're not aware of this, it can trip you up. Don't fret, though; you can easily convert the string that was stored back into the proper data type. In the case of numbers, the built-in **parseInt()** function will convert a string to a number for you:

```
let amount = parseInt(
  localStorage.getItem( "amount" ), 10
);
```

While the requirement to store string values may seem limiting, it's not too hard to work around. It just requires a little bit of effort to convert certain value types to strings and back again. Thankfully, JavaScript provides all the tools you need to do that. For a detailed discussion of this, read "How to Store Objects or Arrays in Browser Local Storage" by Dillion Megida (*freecodecamp.org/news/how-to-store-objects-or-arrays-in-browser-local-storage/*).

If you're curious to see the data stored in **localStorage** for your site or any others, you can access it using your browser's developer tools. In Chrome and related browsers, it's part of the Application tab (see FIGURE A). In Safari and Firefox, it's under Storage.

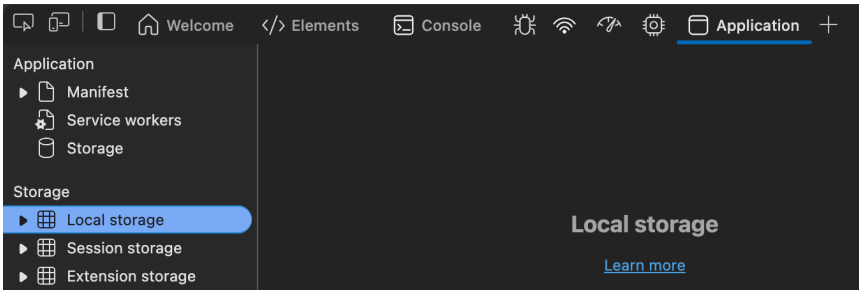

FIGURE A. Accessing local storage in Chrome DevTools

For more on **localStorage**, including how to store complex values like arrays, check out Benjamin Semah's article "How to Use LocalStorage in JavaScript" (*freecodecamp.org/news/use-local-storage-in-modern-applications/*).

It should be easy to see that **localStorage** is the right tool for storing and retrieving the user's current theme in the theme-toggle project. We have one more challenge to overcome, though: if we retrieve the stored value in an external JavaScript file (referenced via the **script** element in our HTML), users will need to wait for that file to be downloaded and processed *before* they'll see the theme change. That could take anywhere from a few hundred milliseconds to several seconds, which is not a great user experience. We'll discuss why that's the case in the next section.

HOW AND WHEN OUR JAVASCRIPT IS PROCESSED

At a basic level, when you navigate to a web page, the browser downloads the HTML first. It then parses the HTML to create the structure of the page we call the DOM. At this point, the browser begins sending out requests for assets like CSS, JavaScript, images, and so on, starting with requests at the top of the HTML source (see FIGURE B). Browsers prioritize requesting certain resources over others to try to render the page as quickly as possible (see **Note**).

NOTE

For a detailed discussion of the browser rendering process, check out the article "How Browser Rendering Works—Behind the Scenes" by Ohans Emmanuel (blog.logrocket.com/how-browser-rendering-works-behind-scenes/).

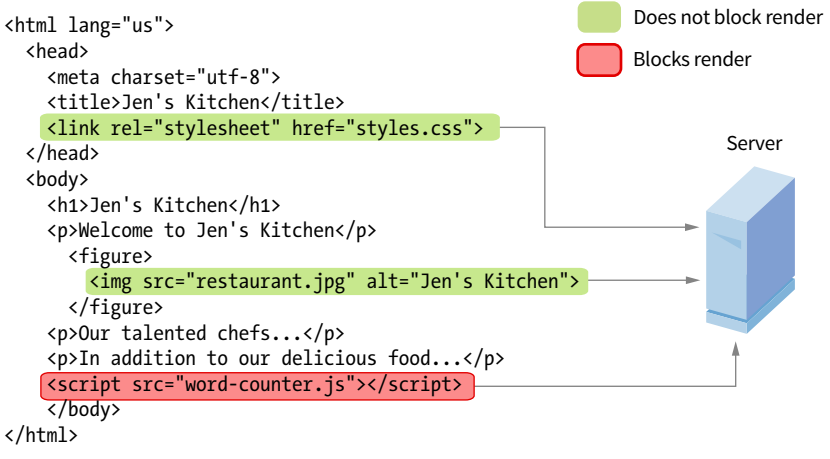

FIGURE B. Browsers must request external resources like CSS, JavaScript, and images. Some resources, like JavaScript, will stop the browser from rendering the page until the file is downloaded and run.

Browsers request CSS as they go, and the CSS will be downloaded and parsed while the DOM is being constructed. JavaScript is different, though. When the browser hits a `script` element, it stops everything until it has downloaded and run the JavaScript code. While the browser is busy doing that, it's not building the DOM. It's not processing CSS. Nothing. That's why we refer to JavaScript as a **render-blocking** resource.

Browsers do this because JavaScript can alter any elements that come before it in the source. The browser doesn't want to continue rendering the document if it might need to go back and rework something further up the page. Developers generally put `script` elements just before the closing `body` tag (`</body>`) to avoid blocking rendering in this way, as you have been doing in the exercises.

That brings us to the problem we need to solve. If a user has chosen the dark theme and we've stored it in `localStorage`, the code to retrieve that information isn't being run until *after* the page has fully rendered. That means the page has the default light theme when the page loads, and it will switch to the dark theme suddenly when the JavaScript runs. That's jarring.

How do we overcome that? Because our theme toggle controls the theme via a **class** on the **html** element, we can put a tiny amount of JavaScript in the **head** of the document to get the theme and apply the **class**, as needed, to the **html** element.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Page Title</title>
    <script>
      // code here can alter the html, head,
      // meta[charset], and title elements
    </script>
  </head>
  <!-- code continues below -->
```

It'll still block rendering, but if the JavaScript is inline—within the **script** element rather than in an external file—it will run immediately. And because it's so high up in the DOM, it will run *before* any rendering has happened, so a user who prefers the dark theme won't see the light theme for even a millisecond.

Armed with this knowledge, let's put a final bit of polish on the theme toggle in EXERCISE A. EXERCISE B, immediately following, is a bonus challenge.

EXERCISE A. Saving the theme

In this exercise, you'll use **localStorage** to save the user's preferred theme and retrieve it again whenever the page is loaded. You'll also use an inline script in your HTML to deliver the best possible user experience.

The JavaScript and HTML documents needed for this exercise are provided with the materials for the book, available at *learningwebdesign.com/materials*. The *theme-toggle.js* file contains the script for placing the theme toggle button and is the starting point for this exercise (FIGURE C).

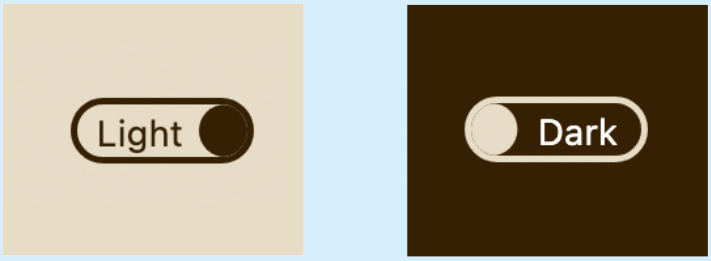

FIGURE C. The theme toggle button

EXERCISE 1. Continued

NOTE

Variables and functions that I have created and named appear in orange*. You can assume that keywords, operators, variables, and functions not in orage are part of the built-in functionality of JavaScript.*

Step 1: Create a function that saves the chosen theme to localStorage

1. Open *theme-toggle.js* in a text or code editor.
2. Above **toggleTheme()**, create a new function called **saveTheme()** that we'll use to save the current theme to **localStorage**:

```
function saveTheme() {

}
```

3. Within the function, declare a variable (**theme**) to track the theme name. Set its value to "light" since that's the default theme for the page:

```
function saveTheme() {
  let theme = "light";
}
```

4. Next, write a conditional **if** statement that checks if the dark theme is currently applied to the **html** element (already available as **$html**), using **classList.contains()**. We did this same step in **EXERCISE 24-4**. Since the **classList.contains()** method returns a Boolean **true** or **false**, we can insert it directly into the conditional statement, like this:

```
function saveTheme() {
  let theme = "light";
  if ( $html.classList.contains("dark") ) {

  }
}
```

5. Inside the body of the conditional—remember, this code will be run if the current theme is dark—assign a new value of "dark" to **theme**:

```
function saveTheme() {
  let theme = "light";
  if ( $html.classList.contains("dark") ) {
    theme = "dark";
  }
}
```

6. Finally, store the theme name, using **localStorage.setItem()**. As with variable naming, it's a good idea to choose a brief and descriptive key. We'll use "saved_theme". Putting it all together, your function should look like this:

```
function saveTheme() {
  let theme = "light";
  if ( $html.classList.contains("dark") ) {
    theme = "dark";
  }
  localStorage.setItem( "saved_theme", theme );
}
```

Step 2: Call that function when the button is clicked

7. Now that we have the **saveTheme()** function ready to store the current theme to **localStorage**, we need to call it. Do that from within the toggle button's event handler. Now, when the user clicks the button, the button gets updated and that theme is stored:

EXERCISE 1. Continued

```
function toggleTheme() {
  $html.classList.toggle("dark");
  updateButton();
  saveTheme();
}
```

Step 3: Create a function to handle initializing the theme and button

8. We also need to update the button when the page loads with the stored theme preference. To do that, we're going to create a new function called **`initialize()`** that runs all the things we want to happen when the page loads and our interface is initialized.

   Create the new function just after the statement that toggles the theme when the user clicks (**`onclick`**):

```
$toggle.onclick = toggleTheme;
function initialize() {

}

updateButton();
document.body.appendChild( $toggle );
```

9. We want the button to update and to be inserted on the page right away, so those existing statements get moved into **`initialize()`**:

```
function initialize() {
  updateButton();
  document.body.appendChild( $toggle );
}
~~updateButton();~~
~~document.body.appendChild( $toggle );~~
```

10. When the interface is initialized, we also want to load the stored theme value from **`localStorage`**. If the browser has stored a value for the "saved_theme" key, we'll get the string value back ("light" or "dark"). If it has not yet been set, the value will be **`null`**, and the user will see the default light theme.

    At the top of the function, use **`localStorage.getItem()`** to get the value by passing in the "saved_theme" key we used:

```
function initialize() {
  let preferred = localStorage.getItem("saved_theme");
  updateButton();
  document.body.appendChild( $toggle );
}
```

11. Now we'll use an **`if`** conditional to check whether the stored theme value is "dark". If it is, we'll add the value "dark" to the **`class`** list in the **`html`** element, and the user will see their preferred dark theme:

```
function initialize() {
  let preferred = localStorage.getItem("saved_theme");
  if ( preferred === "dark" ) {
    $html.classList.add("dark");
  }
```

EXERCISE 1. Continued

```
  updateButton();
  document.body.appendChild( $toggle );
}
```

12. Finally, call the **initialize()** function just after its declaration. This will ensure the function is run when the page loads:

```
function initialize() {
  // function contents
}
initialize();
```

13. Save the JavaScript file and refresh the web page in your browser. Toggle the theme to dark and hit refresh. Did you notice how the light theme is loaded first, then the dark theme is applied? That's because the light theme is the one applied by default. Let's fix that.

Step 4: Fix the flash of the default light theme

14. Open *toast.html* and find the **script** element in the **head** of the document, just after the **title**.

15. Delete the comment and declare a new variable named **saved_theme**. Assign it the value of the stored theme using **localStorage.getItem()**. Remember to use the same key name you used in the JavaScript file: "saved_theme". It's important to recognize that the variable **saved_theme** and the **localStorage** key "saved_theme" are two distinct things. Aligning their names, however, can make it easier to mentally associate the two and how they are being used.

```
<script>
  const saved_theme = localStorage.getItem("saved_theme");
</script>
```

16. Write a conditional (**if**) that checks to see if the value you got back from **localStorage** (**saved_theme**) equals "dark". Use the === (identical to) comparison operator for the check:

```
const saved_theme = localStorage.getItem("saved_theme");
if ( saved_theme === "dark" ) {

}
```

17. If the stored theme is the dark theme, we can apply the "dark" **class** to the **html** element. Inside the body of the conditional, use **classList.add()**. Since this code is running before *theme-toggle.js* is downloaded and run, the **$html** variable we declared it in doesn't exist yet. That's not a problem, though; you can use **document.documentElement** to access the **html** element:

```
const saved_theme = localStorage.getItem("saved_theme");
if ( saved_theme === "dark" ) {
  document.documentElement.classList.add("dark");
}
```

18. Save the HTML file and go back to the browser. Refresh the page, and you should see that the dark theme is already applied when the page loads. No waiting!

19. Okay, it's time for the final test. The moment of truth! Close your browser entirely (Quit if you're on a Mac or Exit if you're on a Windows machine), then start it up again and open *toast.html*. You should see that the dark theme is still applied. Success!

EXERCISE B. Additional challenge—Store an array

This challenge gives you the opportunity to write code more independently.

In the console, create an array. If you still have the one you made for the second challenge in **Chapter 23**, you can use that, or you can create a new array from scratch (say, using the colors of the rainbow). Now I want you to store that array in **`localStorage`** and then retrieve it back. As I mentioned earlier, **`localStorage`** only stores strings, so the challenge is to make the array a string to store it and then turn it from a string back into an array when you retrieve it. The key is using the **`split()`** method to convert a string of words into an array:

```
"one two three".split( " " ); // [ "one", "two", "three" ]
```

To go from an array to a string, you use the array's **`join()`** method.

```
[ "one", "two", "three" ].join( " " ); // "one two three"
```

As long as you keep the separator you use (a space, in this case) consistent, you can go back and forth with ease. Just be aware that if your array items are strings that contain spaces, you'll want to use a different separator. I tend to use a series of characters that rarely appears in my string values: three vertical pipes ("|||").

This is a challenging one, but you've got this!

TEST YOURSELF

Here are a couple of questions to test your knowledge of web storage.

1. Which of the following are methods you can use to store or retrieve data in **`localStorage`**?
   a. **`getValue()`**
   b. **`setValue()`**
   c. **`setItem()`**
   d. **`getItem()`**

2. When it comes to rendering a page, what happens when the browser encounters the following **`script`** element?

   ```
   <script src="some.js"></script>
   ```

   a. It downloads the JavaScript in the background and runs it after the page is finished rendering.
   b. It stops rendering the page.
   c. It stops applying CSS.
   d. It downloads and runs the JavaScript immediately.