Lesson 03.03

- forms and form elements
- event.preventDefault()
- form attributes: action and method
- "GET" vs. "POST"

We have a Login form for which the html and css are all ready. We will look at two different ways to process the form:

- on the page where the form itself is located
- on a separate processor page; this is the default

We don't have a database, so we can't compare the username and password typed into the form to any actual credentials in a database. Instead, we will just use local variables.

1. Open the html page in the browser; it's a typical login form, where the user enters their username and password and clicks a button to submit the form. The html page is currently using FINAL.js, so the form should work.

2. Fill out the form. The correct username is "Amy". The password is "abc".

3. Click the Log In button. This calls a function that compares what you entered to the correct username and password. If they match, you get a "Welcome" message; else the message is "Login failed".

4. In the html, notice that :

- `<form>` tag has no attributes. This is fine for now, but later, we will add attributes to it.
- `<input>` tags have id's for JS to get them.
- `<button>` inside a form submits the form by default on click.

5. Switch to using PROG.js, so that we can code all the functionality from scratch.

6. In PROG.js, declare variables for the username and password. Use any values you like. We'll use uppercase consts, since these values will not be changed:

```
const USER = 'amy';
const PSWD = 'abc';
```

7. Get the DOM elements: button, username and password inputs, and the h2:

```
const btn = document.querySelector('button');
const userBox = document.getElementById('username');
const pswdBox = document.getElementById('password');
const h2 = document.querySelector('h2');
```

8. Have the button call the login function on click:

```
btn.addEventListener('click', login);
```

event object

The **event** object is a default parameter passed to all functions. It is there even if not explicitly added as a parameter. The default name is **event**, but you can call the event object anything you like. Common aliases are **evt** and **e**, but renaming the event object requires that it be explicitly passed in as an argument of the function.

event.preventDefault()

When a button inside a form is clicked, it submits the form by default. This causes the page to redirect to the url specified by the form tag's **action** attribute, where a processor script is expected. If there is no action attribute, the page will reload in place. which resets the form and clears all its elements. We want to process the form on this page, so we need to prevent the default behavior which reloads the page and erases the username and password. To prevent the default, use the **event.preventDefault()** inside the login function.

9. Define the **login()** function, passing in the **event** argument and calling the **event.preventDefault()** method:

```
function login(event) {
    event.preventDefault();
```

10. Get the **username** and **password** from the form input fields. These are the **values** of the input objects:

```
 let user = userBox.value;
 let pswd = pswdBox.value;
```

11. Compare the username and password from the form to the correct username and password. There are two conditions to evaluate, so use the **&&** operator. If the credentials match, output a "Welcome" message and hide the input elements; else output "Login Failed":

```
    if(user === USER && pswd === PSWD) {
        h2.textContent = "Welcome " + USER;
        userBox.style.display = "none"; // hide username field
        pswdBox.style.display = "none"; // hide password field
        btn.style.display = "none"; // hide Log In button
    } else {
        h2.textContent = "Login Failed";
    }
} // end login function
```

leaving out event.preventDefault()

12. Comment out the **event.preventDefault()** line, and fill out the form again. It doesn't work, because the page reloads, which clears the username and password fields before JS has a chance to compare the values to the correct credentials.

form action: send form variables to a processor page

A form action attribute:

- has as its value the page where the form vars are sent: action="form-processor.html"
- since we are redirecting to another page on button click, the button no longer needs a click event to call a function

13. Comment out the listener for the button so that it no longer calls the login function on click.

14. In the form tag, add the action attribute: `**<form action="login-processor.html">**`

form method attribute

15. Also add a method="GET" attribute. This means that the form variables will travel via the URL to the login-processor.html processor page.
```
<form action="login-processor.html"
method="GET">
```

16. Fill out the form, and submit it again. We are redirected to login-processor.html.

17. Check the URL up in the browser address bar. Notice that the url ends with a question mark **"?"**. The part of the url that begins with the **?** is called the **querystring**.

name attribute of form elements

There are supposed to be name=value pairs following the question mark, specifically: **"?username=amy&password=abc"**. The reason these are absent is because these only occur for form elements that have a name attribute.

18. Go back to the html form and add name attributes to the two input elements.
    The name values can be anything, but it makes sense that they match the id's:
```
<input type="text" id="username" name="username" placeholder="username">
<input type="password" id="password" name="password"
placeholder="password">
```

19. Run the form again. This time up in the browser address bar at login-processor.html, we should see that the url ends in the querystring: **"?username=amy&password=abc"**

The code for accessing querystring variables involves arrays and loops--topics we have not covered yet--so instead, we will use another way:

sessionStorage for saving Session Variables

Session Variables are variables that we store in the browser, as oppposed in the current script page. This makes the vars available to any page in the application that is opened in the browser. Over in the form page,

we will save the username and password as session variables. This way, we will be able to access them on the processor page.

mouseover event

The **mouseover** event occurs whenever the cursor hovers over some DOM element, such as a button, div or image. As with any event, **mouseover** can call a function. The reason we want mouseover rather than click is because we want to make the session variables before the form is submitted on click.

20. Below the commented-out listener that calls the login function on click, add a listener that invokes a function called setSessionVars and runs on mouseover:

```
btn.addEventListener('mouseover', setSessionVars);
```

21. Define the **setSessionVars()** function:

```
function setSessionVars() {
    // 20. Call the setItem() method on the sessionStorage property,
passing it two arguments: a variable name in quotes and a value, with
the values being the username and password from the input boxes:
    sessionStorage.setItem("user", userBox.value);
    sessionStorage.setItem("pswd", pswdBox.value);
    console.log("Session Vars: user", userBox.value, "pswd",
pswdBox.value);
}
```

22. The JS for checking the login credentials is now going to be in the processor page, where we are redirected on submit. Switch to that page.

Form Processor Page

The following JS code is imported by **form-processor.html**, which processes the form, that is, checks the user log in.

- it gets the session variables set in the form page
- compares the session variables vars to the "real" username and password
- since we do not have a real database in which to look up the correct username and password, we have hard-coded those as constants, USER and PSWD

23. Get the session variables, and save them to "regular" vars:

```
let user = sessionStorage.getItem("user");
let pswd = sessionStorage.getItem("pswd");
console.log("session var user:", user);
console.log("session var pswd:", pswd);
```

24. Hard-code the correct username and password:

```
const USER = 'amy';
const PSWD = 'abc';
```

25. Get the h2 that displays the login response message:

```
const h2 = document.querySelector('h2');
```

Compare the session vars to the correct username and password:

26. First make sure the session vars exist; if they do not, the form was submitted with one or both inputs left blank in which case the login fails

```
if(user && pswd) {
    // 27. If the vars are set, next check to see if they match the
correct login credentials
    if(user == USER && pswd == PSWD) {
        // 28. Output a personalized welcome message on login success
        h2.textContent = `Welcome back, ${user}`;
    } else {
        // 29. Else, utput a "Login failed"
        h2.textContent = 'Login failed.';
    }
// 30. If no session vars exist, login also fails
} else {
    h2.textContent = 'Login failed.';
}
```

27. Back in the html log in page, in the form, change the method from "get" to "post": `<form action="login-processor.html" method="post">`

28. Re-run the form. Notice that now in the url of the processor page, there is no querystring, that is no **"?username=amy&password=abc"** This is because the post method transmits the form variables to the processor invisibly.

same-page processing vs. processor-page

To fully appreciate the difference between a form that is processed on the same page, as opposed to a form that redirects to a processor page, lets' go back to the same-page processin version.

33. Comment out the form tag with the action and method attributes and add replace it with an empty

    tag.

34. In the JS file, reactivate the button listener that calls the login function on click: **btn.addEventListener('click', login)**

35. In the **login(event)** function, reactivate the **event.preventDefault()** line.

36. Fill out and sumbit the form. This time, you stay on the same page.

form methods: GET vs POST

When a form is submitted, from variables as name=value pairs are sent to a script for processing. In the case of a login form the from variables would be something like username and password, or perhaps user and pswd, he values of which would be whatever the user entered into the login form.

"GET" : if a form has **method="GET"**, it means that:

- the form variables are transmitted to the processing page vias the URL querystring.
- right after the url of the page itself, there will be a question mark **?**, followed by one or more **name=value** pairs, connected by ampersands **&**:
- the **?** separates the url of the page itself from the querytstring
- **"GET"** is commonly used for HTTP requests that GET or request something, such as search results

"POST" : if a form has **method="POST"**, it means that the form variables are transmitted to the processing page invisibly -- they do not appear in the URL. **"POST"** is commonly used for HTTP requests that POST or write something, such as a registration form that logs a new user to the database.

"GET" and "POST" will become enormously important concepts when you get into **Node.JS**, where you will constantly be making "GET" and "POST" requests as you build server-side applications with database connectivity.