Session 8 exercises

Advanced JavaScript for Web Sites and Web Applications

Exercise 1 - Local storage

App overview

We are going to build an application that enhances a web form, allowing the user's input to be saved in the browser's **localStorage** so that they can resume where they left off in case they have closed their browser or lost their internet connection.

We will also provide an option for the user to clear the data we have stored in their browser.

You will find the form we will be working with in `exercise1.html` in the session 8 workshop files.

Note, we are not concerned with processing the data when the user submits the form (i.e. the *add user* button). You should assume that this will be taken care of by server-side scripts.

JavaScript's role in this application is to enhance the user experience, not to provide core functionality.

The FormRecall module

Our objective is to allow the user to add data to a form, and for our application to be able to recall that data at a later date/time.

We will use the *revealing module* pattern.

The module will:

- Store the data the user enters *as they type*
- Populate the form with stored data when the user clicks the *Restore* button
- Delete all stored data when the user clicks the *Clear* button

So, we will have 4 functions in the module:

- a function to store the input values in local storage
- a function to clear the local storage completely
- a function to populate the form with previously stored data
- an init function, to trigger everything (this is the only function we need to make public)

We will also need 3 event handlers to run when:

- User clicks the *Restore* button
- User clicks the *Clear* button
- User types something in a form field (keyup event)

Open *form-recall.js* where you will find a skeleton module for you to work with.

The HTML elements

In form-recall.js, the module variables have been defined for you:

- `form` : A reference to the form containing the input fields
  - we will attach the `keyup` listener to this element and use delegation.
- `clearButton` and `restoreButton` : References to the 2 buttons.
  - We will attach `click` listeners to these (we are not concerned with the *Add user* button)
- `nameInput` and `emailInput` : References to the input fields
  - We will both get data from these, and populate them with data

The Functions

The necessary functions have also been defined for you:

- `storeFieldValue` : This function is attached to the `keyup` event on the form fields. It accepts an argument (i.e. the `event` object). Currently, the function code retrieves the data that the user has typed in the field and displays it in the console.
- `clearStorage` : This function is attached to the `click` event on the *clear* button. It accepts an argument (i.e. the `event` object). Currently, the function code simply logs a debug message.
- `populateForm` : This function is attached to the `click` event on the *restore* button. It accepts an argument (i.e. the `event` object). Currently, the function code simply logs a debug message.
- `init` : This function sets up the event handlers for the module.

The event handlers

In the `init` function, the event handlers have been setup for you:

- `storeFieldValue` is attached to the `keyup` event on the `form`
- `populateForm` is attached to the `click` event on the `restoreButton`
- `clearStorage` is attached to the `click` event on the `clearButton`

Testing the module

Open exercise1.html in your browser and try typing in either of the form fields. You should see a debug message with the text you entered appear in the console.

Try clicking on the *Clear* and *Restore* buttons. Again, the console should display debug messages telling you which function is running.

Task 1 - Store the user's input in local storage

The `storeFieldValue` function has been attached to the form's `keyup` event.

Because of *bubbling*, this event will fire for both of the input fields.

This function currently:

- verifies that a text input triggered the event (using `.nodeName`)
- retrieves the value from the text input (using `.value`)

You should amend the code so that it also:

- retrieves the `name` of the input that has been typed in (using `getAttribute`)
- stores the entered value in `localStorage`

Each input's value will be stored in a separate storage *item*. You should use the input's *name* attribute as the *key*.

E.g.:

```
// Get the "name" attribute from field
var itemName = event.target.getAttribute("name");
// Get the value
var itemValue = event.target.value;
// Store "value" under "key"
window.localStorage.setItem(itemName, itemValue);
```

Task 2 - Retrieve data from local storage

The `populateForm` function will be responsible for retrieving data from localStorage and adding it to the form inputs.

However, there may not be any data in localStorage (user's first visit), so we need to allow for this. (remember, `getItem` returns `null` if the key we request is not present in localStorage)

For each of the *input* fields:

- check if a value is stored in localStorage
  - If there is data for that input in storage, add it to the form field (using `.value`)
  - If there is no data stored for that input, do nothing

E.g.:

```
var storedName = window.localStorage.getItem("userName"),
    storedEmail = window.localStorage.getItem("userEmail");

if(storedName !== null) {
    nameInput.value = storedName;
}

if(storedEmail !== null) {
    emailInput.value = storedEmail;
}
```

Task 3 - Removing stored data

The `clearStorage` function will be relatively simple. All it has to do is call the `clear` method of the `localStorage` object.

E.g.:

```
window.localStorage.clear();
```

Test the module

1. Open `exercise1.html` in your browser again and type something in the fields.
2. Close your browser
3. Open `exercise1.html` in your browser again and click on the *restore* button
    - The data you entered in step 1 should be displayed in the form
4. Click the *clear* button and then close your browser.
5. Open `exercise1.html` in your browser again and click on the *restore* button
    - This time, nothing should be displayed in the form.

Exercise 2 - Geolocation

App overview

We are going to build a module that displays a Google map with a marker depicting the user's current position.

Again, we will use the *revealing module* pattern.

The module will:

- create an instance of a Google Map on the page
    - To start, we will specify a low zoom level and center the map around Europe
- Listen for clicks on the *Find me* button
    - When clicked, retrieve the user's position (using `getCurrentPosition()`)
- Provide callbacks for the geolocation object's *success* and *error* events
    - On *success*: center the map on the new position, add a marker to the map, adjust the zoom level
    - On *error*: display an `alert()` with the *message* contained in the error object

So, we need a total of 5 functions for our module:

- A function to initialise the map
- A function to get the user's position from the geolocation object
- A function to run when the user's position has been retrieved
- A function to run when an error is encountered retrieving the user's position

- An init function, to trigger everything (this is the only function we need to make public)

We also need 2 event handlers to run when:

- the window has loaded
  - this is when we initialise the map
- the user clicks the *Find me* button
  - this is when we will attempt to retrieve their location

Open `mapper.js` where you will find a skeleton module for you to work with.

The module variables

The module variables have already been defined for you in `mapper.js`:

- `gMap` : This has no initial value. We will store the map object in it once it has been initialised.
- `button` : This is a reference to the *Find me* button
- `mapWrapper` : This is a reference to the HTML element we want to put the map in
- `mapOptions` : This is the options object that we pass to the Map method while initialising the map.

For the map options object, when we load the map we want to display it zoomed out considerably, centered on Europe. To achieve this, we have set **45, 5** as the coordinates and **5** as the zoom level

The module functions

In `mapper.js`, The necessary functions have also been defined for you:

- `initializeMap` : this will be attached to the window's load event. It will accept no arguments.
- `getPosition` : this will be attached to the `click` event on the `button` element, so should accept an argument (i.e. the `event` object)
- `positionSuccess` : this will be the *on success* callback for the geolocation object, so should accept an argument (i.e. the `position` object)
- `positionError` : this will be the *on error* callback for the geolocation object, so should accept an argument (i.e. the `error` object)
- `init` : this is where you will attach the event handlers. This is the only *public* function.

Task 1 - Add the event handlers

In the `init` function, set up the event listeners that we will need.

1. Attach `initializeMap` to the window's load event (via the google maps event interface)
2. Attach `getPosition` to the `click` event on `button`

Task 2 - Initialise the Google Map

In the `initializeMap` function, we will instantiate a new Google Map.

- We will do this with the API's `Map` constructor, passing it:
  - the element we want to place the map in (`mapWrapper`)
  - our `mapOptions` object
- We will store the resulting Map object in the `gMap` variable:

E.g.:

```
gMap = new google.maps.Map(mapWrapper, mapOptions);
```

You can now preview your page and you should see a Map embedded in it, centered on Europe.

Task 3 - The `getPosition` function

The `getPosition` function runs when the user clicks the *Find me* button. It should accept an `event` argument.

Inside the function, we need to:

- Prevent the default behaviour (`preventDefault`)
- make a call to `getCurrentPosition`
  - passing it the names of our *callback* functions

E.g.:

```
event.preventDefault();
navigator.geolocation.getCurrentPosition(positionSuccess, positionError);
```

Task 4 - The `positionSuccess` function

The `positionSuccess` function runs when we have successfully retrieved the user's position. It receives a `position` object as it's argument.

This function needs to:

1. extract the user's coordinates from the position object
2. add a new Google Map marker to the map
   - using the extracted coordinates
3. change the zoom level of the map
   - set it to 16
4. center the map on the user's location

Remember, we can get the user's coordinates from the `coords` property of the position object.

```
positionSuccess = function (position) {

    var userLat, userLng, newLocation, marker;

    // (1) Get user's coords from "position" object
    userLat = position.coords.latitude;
    userLng = position.coords.longitude;

    // Create LatLng object for marker
    newLocation = new google.maps.LatLng(userLat, userLng);

    // (2) Create marker and add to map
    marker = new google.maps.Marker({
        position: newLocation,
        map: gMap
    });

    // (3) Adjust the zoom
    gMap.setZoom(16);

    // (4) Center the map on user's location
    gMap.setCenter(newLocation);

};
```

Task 5 - The `positionError` function

How we handle errors will vary, depending on our application's needs. Also, as we have seen, we will not always be notified of the user's response.

For now, we can simply call `alert`, passing it the *message* property from the error object:

```
positionError = function (err) {
    alert(err.message);
};
```

Test the app

Now, you should be able to test the app fully.

Load exercise2.html in your browser and click the *Find me* button. Assuming you allow your location to be shared with the page, a marker should be displayed on the map and the zoom level should change.

Bonus!

At the moment, while we are retrieving the user's location, the user is not made aware that something is happening. This may cause them to believe their is a problem with the system.

To rectify this:

- When the call to `getCurrentPosition` is made, add the class `loading` to the HTML element that holds our map (`mapWrapper`)
- When the call to `getCurrentPosition` is complete, remove the `loading` class from the map element.
  - Remember, the *callbacks* run when the call is complete

Note, there are CSS rules in place to display a loading graphic on the page, assuming the `loading` class has been added to the correct element.

Bonus 2!

Consider the scenario where neither our success nor error callbacks get executed.

- You can trigger this in Firefox by viewing the page locally and either ignoring the popup or selecting the *not now* option

How might you deal with this in your code?

You might use setTimeout to monitor things, as we saw in the presentation, but what will the application do if no response is received?