JavaScript Part 7: Forms, Validation and Local Storage

Forms are where visitors give us information, and good code treats every entry with care. Four steps handle a form from the first click to the next visit: capture, validate, store and restore. Each step answers one simple question, and the browser console is enough to test every line. In this part, Maya, a product developer at a corporate training provider, uses them to build the registration form of a workshop.

1. Capture

A form sends its data when the visitor submits it, and by default the browser sends the data away and loads a new page. The submit event lets our code take over, and the preventDefault method of the event object stops that default behaviour, so that our code decides what happens next. The value property of each field gives what the visitor typed, and it is always a string, even in a field meant for numbers. The question is: what exactly did the visitor enter, and who handles it now? We listen for submit on the form itself, not for clicks on the button, because a form is also submitted when the visitor presses the Enter key. Reading the values into one object keeps the rest of the code tidy.

Example: A receptionist hands each visitor a paper form, and when it comes back she reads it before it goes into the filing cabinet. She can check the name, ask about a missing line and set the form aside if something is wrong. Nothing is filed until she has looked at it. The submit event gives our code that chance to read first, and preventDefault is the hand that keeps the form out of the cabinet.

At work: Maya adds a submit listener to the registration form. She asks the Capture question, calls preventDefault and reads the name, the email, the course choice, the number of seats and the comment from the fields into one object. Because the values are strings, she converts the number of seats with the Number function, as in the first part of this series. The form data is now in her hands, ready to be checked.

2. Validate

Validation checks that the data is acceptable before we use it. HTML already offers attributes such as required, a type of email and a minimum length, and the browser checks them before the form is submitted. Our own code adds the rules that HTML cannot express, for example that the number of seats must be at least one and must not exceed what the room allows. The question is: what makes an entry acceptable, and how do we say so clearly? An error message should appear next to the field and tell the visitor what to fix, in plain words. Checks in the browser help visitors but they are not security, because anyone can bypass them, so the server must check the data again.

Example: A passport officer checks that every field is complete and legible before stamping, and tells the traveller exactly what is missing, for example that the date of birth is unreadable. A vague refusal sends the traveller away confused and wastes everybody's time. A clear message lets the traveller fix one thing and come back. Validation works the same way in a form, so each message sits next to the field and says what to fix.

At work: Maya relies on required and the email type for the basics. She asks the Validate question and adds a check that the number of seats lies between one and the limit of the room. When a check fails, she writes a message such as: please enter a whole number of seats, under the field with textContent. The server will validate again, because browser checks can be bypassed.

3. Store

The localStorage object keeps data in the visitor's browser, and the data stays there after the browser is closed. It stores pairs of a key and a value, and the value is always a string. The setItem method saves a

This document is a property of Asaseeyat Academy - © Copy Right Protected.

value, the getItem method reads it, and the removeItem method deletes one key. To store an object or an array, we first turn it into a string with JSON.stringify, and we turn it back later with JSON.parse. The question is: what is worth keeping, and is it safe to keep it here? For example, `localStorage.setItem("draft", JSON.stringify(draft))` saves the draft object as a string under the key draft. Local storage belongs to one website address, any script on that site can read it, and the visitor can edit it, so passwords and sensitive personal data do not belong there.

Example: A notepad on a desk keeps tomorrow's to-do list after the lights go off, and the next morning starts from the notes. It holds only written words, so a long list must first be copied out as plain text. Anyone who sits at that desk can read it, so a private password never goes into it. Local storage is that notepad inside the visitor's browser, and JSON.stringify does the copying out.

At work: Maya saves a draft of the course choice and the comment while the visitor types. She asks the Store question and turns the draft object into a string with JSON.stringify before calling setItem. She deliberately leaves the email and anything sensitive out of the draft. The visitor can close the tab by accident without losing the work.

4. Restore

Saving is only useful if the data comes back. When the page loads, we read the key with getItem, which returns null when nothing was saved, so we must check before using the result. If a string exists, JSON.parse turns it back into an object, and we put its values into the fields. The question is: what should the visitor see when they return? After a successful submission, we remove the draft with removeItem so that old data does not come back by mistake. A damaged or outdated draft can make JSON.parse fail, and careful code is prepared for that case too.

Example: A hotel keeps a guest's preferences on file, and when a returning guest arrives, the desk greets them with the usual room. A new guest gets a plain welcome because there is no file on them yet, and the file of a guest who has left is cleared. The hotel reads, uses and removes the information at the right moments. Our page does the same: it reads with getItem, parses with JSON.parse, removes with removeItem, and checks for null first.

At work: When Maya's page loads, she reads the saved draft. She asks the Restore question: what should the visitor see when they return? If a draft exists, she parses it and fills the course choice and the comment, and if not, the form stays empty. After a successful registration she removes the draft, so that the form starts fresh next time.

Conclusion

Maya's registration form now catches mistakes early and remembers unfinished work. Capture asks what the visitor entered and who handles it now, and validate asks what makes an entry acceptable. Store asks what is worth keeping and whether it is safe to keep it there, and restore asks what the visitor should see on returning. In the next part, we ask a server for data while the page keeps working.

Sources

- MDN Web Docs: Window.localStorage – https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage
- MDN Web Docs: Using the Web Storage API – https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API/Using_the_Web_Storage_API
- MDN Web Docs: Client-side form validation – https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Forms/Form_validation
- MDN Web Docs: HTMLFormElement submit event – https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement/submit_event
- MDN Web Docs: Storage.getItem() – https://developer.mozilla.org/en-US/docs/Web/API/Storage/getItem
- MDN Web Docs: JSON.parse() – https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse

This document is a property of Asaseeyat Academy - © Copy Right Protected.