Creating an In-Browser Editor

One thing that blew my mind was a web-based slide deck that changed a style's CSS when you edited the CSS block shown as text in the slide. Think about it... We can change the CSS of our content as we need to just by editing the CSS inline, in the same browser that is displaying the content we are styling. Awesome!

Later on I changed my mind a little. CSS is too much of a pain but what if we want to change a document or create a new one and then save it or do some further processing with it.

In this post I'll cover things like `contenteditable`, `execCommand`, other ways to create and add content to our editable regions and how to package an editor into a web component.

What do we want to do

Before diving into the technology let's talk about what we want to do for a second, It's always a good idea to have a spec before we start doing the work so we won't loose sight of the final objective.

I want to build an inline editor that will let me edit CSS, Javacript and HTML. Think of a limited version of Netscape Composer but inline in the browser rather than being a separate application.

I want to be able to type content directly in the editor and have buttons that will let me insert some HTML tags on their own or over highighted content.

Some functionality

At it's simplest we can make a section editable by adding the `contenteditable="true"` attribute to the element we want to make editable.

```
<div class="editor-container" contenteditable="true">
 <p>This is the editor</p>
```

```
</div>
```

We can change the content of the editor, but that's about the extent of it. However this will change if we start working with styles. Take the following style element placed inline in a page's body.

```
<style>
  body {
    font-size: 1.5em;
    font-color: black;
  }
</style>
```

In your master stylesheet make the style sheet visible by changing its display to block.

```
style {
  display: block
}
```

This will make the stylesheet visible in your document but will still work styling the content. One last change to make, make the style element editable.

```
<style contenteditable="true">
  body {
    font-size: 1.5em;
    font-color: black;
  }
</style>
```

In this 3-step process we've created a basic CSS editor for the page it is running on. It will only work on the page we place the styles and, right now, we can't save the styles.

We'll create a button to edit the content and wire it to enable and disabling content editable; When we make the content non editable we make sure we save

the content in local storage.

```
const editBtn = document.getElementById("editBtn");
const editable = document.getElementById("editor-content");

editBtn.addEventListener("click", function(e) {
  if (!editable.isContentEditable) {
    editable.contentEditable = "true";
    editBtn.innerHTML = "Save Changes";
    editBtn.style.backgroundColor = "#6F9";
  } else { // Disable Editing
    editable.contentEditable = "false";
    editBtn.innerHTML = "EDIT CONTENT";
    editBtn.style.backgroundColor = "#F96";
    saveContent();
  }
});
```

The save function is simple, is creates an item in local storage with key of the id for our editor (`content-editor`) and the data in the editor (the editor's innerHTML) as it's value.

```
function saveContent() {
  if (typeof(Storage) !== "undefined") {
  // Save the data in localStorage
  localStorage.setItem(editable.id, editable.innerHTML);
  }
}
```

If you save the changes and then reload the editor you will find that the editor will revert to the last time you saved it. That's because there is a way to save the content not not to retrieve it.

> To handle loading content from local storage wedo three things:

1. We check that local storage is supported
2. If it's suppoorted then we check that there is a value that matches what we want to load (`content-editor`)

3. We load the data inside the editor's body

Yes, I'm being overtly cautious in checking if Local Storage is supported, both when saving and loading content. We could make a function out of the test but I'm comfortable doing it this way.

```
if (typeof(Storage) !== "undefined") { // 1
  if (localStorage.getItem('editor-content') !== null) { // 2
    editable.innerHTML = localStorage.getItem('editor-content'); // 3
  }
}
```

Right now we have a bare bones functional editor. We can write on it, we can save the data and when we return to it, we'll be able to resume editing from where we left off. This is a good starting point but we can definitely do more.

Adding functionality

This is the most interesting part and also the most difficult to get working across browsers. `document.execCommand()` allows you to run a list of predefined commands insie your text editor. It takes three parameters:

- The name of the command to run
- Whether to show the default UI for the command (not supported in Firefox so we leave it as false throughout)
- The parameter for the command, if needed. For example the URL for the image if we are inserting one

This is not a trivial undertaking and most times I would just turn to a canned solution like the ACE or TinyMCE editor systems but for the purpose of this article I Want to be able to add the functionality manually.

Because I want to add several commands we'll have to get creative and use JSON as the data sourceand some creative use of JAvascript to generate the buttons and add them to the page.

The JSON file takes three elements:

- The name of the command
- The command itself

- Any parameters for the command or null if there is none

```
const commands = [
  {
    "name": "p",
    "command": "formatBlock",
    "param": "p"
  },
  {
    "name": "h1",
    "command": "formatBlock",
    "param": "h1"
  },
  {
    "name": "h2",
    "command": "formatBlock",
    "param": "h2"
  },
  {
    "name": "h3",
    "command": "formatBlock",
    "param": "h3"
  },
  {
    "name": "h4",
    "command": "formatBlock",
    "param": "h4"
  },
  {
    "name": "h5",
    "command": "formatBlock",
    "param": "h5"
  },
  {
    "name": "h6",
    "command": "formatBlock",
    "param": "h6"
  },
```

```
  {
    "name": "Bold",
    "command": "bold",
    "param": null
  },
  {
    "name": "Underline",
    "command": "underline",
    "param": null
  },
  {
    "name": "Strike Through",
    "command": "strikeThrough",
    "param": null
  },
  {
    "name": "Remove Formatting",
    "command": "removeFormat",
    "param": null
  }
]
```

We then run the commands through `array.map` to perform the following actions:

1. Create variables to hold each part of the command
2. Create a button element
3. Give it the name of the command
4. Assing an ID attribute of the command itself
5. Create an `onclick` event handles and use it to add a `document.execCommand` command for the appropriate element

```
var tb1 = document.getElementById('toolBar1');

commands.map(function(item) {
  var name = item.name; // 1
  var command = item.command;
  var param = item.param;
```

```
  var newButton = document.createElement('button'); // 2
  newButton.textContent = name; // 3
  newButton.setAttribute('id', command); // 4

  newButton.addEventListener('click', () => {
    document.execCommand(command, false, param) // 5
  });

  tb1.appendChild(newButton);
});
```

This method will provide a basic set of commands but will not handle some of the more complex cases like inserting images or hyperlinks.

I've chosen not to address these cases and leave them as exercise for the reader. There are commercial solutions like ACE or TinyMCE editor systems that do a much better job than I've done in this project and I'mm ok with that. This is a proof of contcept and will probably not go into much further development.

Links and resources

- Basic support for content editable
- Design Mode
- execCommand
- Content Editable Spec @ W3C
- Input Events Spec @ W3C
- Rich Text Editing in Mozilla