Notes - Application Development - Client-side

Travel Notes - Part 2

- Dr Nick Hayward

A brief outline of application development for client-side publication.

Contents

Intro *Travel Notes* is a basic application to help showcase development patterns and concepts for *client-side* applications.

We may consider its development using two comparative options for JavaScript implementation. This allows us to compare a popular JS front-end library, *jQuery*, and custom plain JavaScript.

This application allows a user to create text notes, organise and render them in a grid and flexible layout, and query a remote API for contextual information. In this example, the user may search for images, which may be associated with the text notes created in the app.

In *Part 2*, we may add the following functionality to this app

- delete option
  - including single note and all notes
- code abstraction and refactoring
- various style updates
- …

New features and add-ons We can now start to add some new features and options to our ongoing Travel Notes application.

The first thing we need to add is a delete option for users, and then modify the styling for our notes.

Delete option As a user adds their notes to the application, likewise they should have the option to delete them.

delete all notes We'll start by considering an option to **delete all** notes.

There are many different ways to achieve the same result, but in essence we may initially use jQuery's `remove` function with the specified elements.

For example,

```
$("p").remove();
```

We can add the **delete all** option as another button, for example, and perhaps create a new toolbar for such note options,

```
<section class="note-controls">
  <button id="notes-delete">Delete all</button>
</section>
```

Then we need to add some much needed initial styling for this new toolbar. We'll match our basic button styling to the earlier `add` button, and add a bottom border to help divide the toolbar from the note output section.

```
/* note controls */
.note-controls {
  margin: 10px 0 10px 0;
  padding: 2px;
  border-bottom: 1px solid #dedede;
  display: none;
}

/* simplify default button styles for note controls */
.note-controls button {
  padding: 2px;
  margin: 2px;
  border-radius: 0;
  border: 1px solid #dedede;
  cursor: pointer;
}
```

We may also add this functionality to our plain JavaScript version of the current *Travel Notes* app.

```
// delete all notes button
let deleteAll = document.getElementById('notes-delete');

// add event listener for delete all notes...
deleteAll.addEventListener('click', () => {
    // get notes from DOM
    let notes = noteOutput.querySelectorAll('p');
    // loop through notes and remove a single note per iteration...
    for (let note of notes) {
        note.remove();
    }
});
```

The HTML and CSS is not changed, of course, for the plain JS version.

check visibility As the note controls toolbar is hidden by default in the CSS, we need some way to check its visibility as we add our notes. If there are no notes, then the toolbar is not required.

So, we can now add a new function to our jQuery version of the app.

```
//check element visibility - expects single element relative to display:none
function checkVisible(element) {
  if (element.is(":hidden")) {
    element.fadeIn();
  }
}
```

For now, we are keeping it simple. We are checking a passed element to see if it is hidden. If the matched element is hidden, we can then simply fade it into the view. So, we can now call this function as we add new notes to the `.note-output` section,

```
checkVisible($(".note-controls"));
```

The option to show the element is currently using a default jQuery animation, `fadeIn` . However, we may define a custom animation for plain JS, which will likewise show or fade in the element.

For the plain JS version, we may consider a couple of comparative options for checking visibility.

The first option uses the `display` property to check the passed `node` ,

```
// check visibility of passed node
function checkVisible(node) {
    // check passed node's current visibility
    if (node.style.display != 'block') {
        // show in DOM to allow fadeIn...
        node.style.display = 'block';
        // call fadeIn for node in DOM
        fadeIn(node);
    }
}
```

and may be used in our existing app code as follows,

```
// define node to check...
let controls = document.getElementById('controls');
// call function
checkVisible(controls);
```

In effect, we grab the defined selector in the DOM, and check its current visibility relative to `display` .

Our second option uses the `visibility` property to check the passed `node` ,

```
// check visibility of passed node
function checkVisible(node) {
    // check passed node's current visibility
    if (node.style.visibility = 'hidden') {
    // show in DOM to allow fadeIn...
    node.style.display = 'block';
    node.style.visibility = 'visible';
    // call fadeIn for node in DOM
    fadeIn(node);
    }
}
```

The following video shows the difference between these properties, *display* and *visibility*.

- CSS Display and Visibility - YouTube - up to 1:46

delete all - usage As we add a note, the `.note-controls` toolbar is shown, and the **delete all** button now becomes available to our users.

To delete all of the notes, we may use the following jQuery handler,

```
//handle deletion of all notes
$("#notes-delete").on("click", function(e) {
  var $note = $(".note-output p");
  $(this).parent().hide();
  $note.remove();
});
```

In this code, we are creating a new handler for the click events on the `#notes-delete` button, which then hides its own container, the notes toolbar, and then removes all of the notes from the `.note-output` section. As there are no notes visible, we may safely hide the note toolbar.

So, our jQuery code for the *delete all* notes options is currently as follows,

```
//check element visibility - expects single element relative to display:none
function checkVisible(element) {
  if (element.is(":hidden")) {
    element.fadeIn();
  }
}
...
//handle deletion of all notes
$("#notes-delete").on("click", function(e) {
  var $note = $(".note-output p");
  $(this).parent().hide();
  $note.remove();
});
```

and the plain JS version is as follows,

```
// delete all notes button
let deleteAll = document.getElementById('notes-delete');

// add event listener for delete all notes...
deleteAll.addEventListener('click', () => {
    // hide parent controls node...
    deleteAll.parentNode.style.display = 'none';
    // get notes from DOM
    let notes = noteOutput.querySelectorAll('p');
    // loop through notes and remove a single note per iteration...
    for (let note of notes) {
        // remove single node
        note.remove();
    }
});
```

We may test the latest update in the first example of the second part of this app,

- DEMO 1 - travel notes - series 2

check element exists However, whilst this update works, we are still making an assumption, rightly or wrongly, that the notes exist in the `.note-output` section.

So, just to be safe, we may add an additional function to quickly check whether an element exists in the DOM or not.

For this check, we can use the `length` property,

```
$("p").length
```

So, our new function for checking elements in the DOM is as follows,

```
//check elements exists
function checkExist(element) {
  if (element.length) {
    return true;
  } else {
    return false;
  }
}
```

We may then check the returned boolean, and proceed accordingly.

Whilst this, and the similar `checkVisible()` function, may seem simple and unnecessary abstraction, it allows us to add any further programmatic options, as needed, as we continue to develop our code.

We may also use this abstracted functionality with the jQuery and plain JS versions of the app.

delete all functionality So, our updated *delete all* functionality is as follows,

- updated delete all notes option to include check for notes
- call `checkExist()` function in conditional statement

The current jQuery handler is as follows,

```
//handle deletion of all notes
$("#notes-delete").on("click", function(e) {
  //set note selector
  var $note = $(".note-output p");
  //check $note exists
  if (checkExist($note) === true) {
    //hide note-controls
    $(this).parent().hide();
    //remove all notes
    $note.remove();
  }
});
```

and the plain JavaScript version,

```
// add event listener for delete all notes...
deleteAll.addEventListener('click', () => {
    // get notes from DOM
    let notes = noteOutput.querySelectorAll('p');
    // check notes in DOM
    if (checkExist(notes) === true) {
        // hide parent controls node...
        deleteAll.parentNode.style.display = 'none';
        // loop through notes and remove a single note per iteration...
        for (let note of notes) {
            // remove single node
            note.remove();
        }
    }
});
```

The latest updates may be seen in the second example,

- DEMO 2 - travel notes - series 2

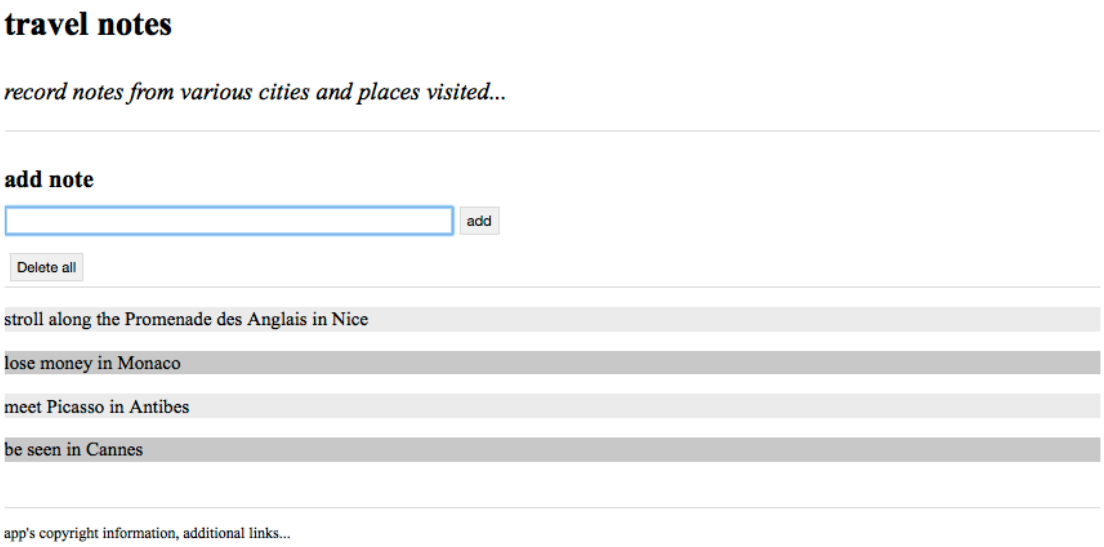

Figure 1: Travel Notes - Series 2 - Demo 2

delete a single note Let us now consider adding a single delete option per note. In effect, allowing a user to selectively delete their chosen note, regardless of hierarchical position within the `.note-output` section.

The first decision we need to make is how we present this option to our users. Do we offer a selection option, such as checkboxes, to select one or more delete items, or perhaps a single delete button per note, or a drag and drop to delete option. There are many different ways we may present and use this option.

However, programmatically we may follow a similar pattern for the deletion of the note.

We'll now take a look at three possible options, three jQuery functions that can help us remove elements from a document for the jQuery version of the app.

- `remove()`
- `detach()`
- `replaceWith()`

jQuery options Each of these functions has a different way of handling such removals or deletions, and inherent benefits depending upon how we want to handle our elements, in this case our notes.

We've already encountered the `remove()` function in jQuery but, of course, this was relative to clearing our `.note-output` section.

The notable feature of the `remove()` function is that it should, effectively, be used to remove or delete elements *permanently* from a document. This function will also **unbind** any attached event handlers for the elements being removed. This function will return, if necessary, a reference to these removed elements, but not the original bound events.

The second option, `detach()` , is often used for any temporary removal requirements. For example, if we need to update many nodes in the DOM, it might make more sense to detach the affected elements, and then insert later as required. In contrast to the `remove()` function, `detach()` retains its event handlers, and we may add these elements later.

For example,

```
$("p").detach();
```

We could use, for example, the jQuery function `appendTo()` to insert these elements in the DOM.

```
var $detachP = $("p").detach();
$detachP.appendTo("#detached");
```

Our third option, the function `replaceWith()` , allows us to replace an element, or group of elements, with the passed element. Similar to the `remove()` function, event handlers for the replaced elements are unbound.

```
var $replacedP = $(".note-output p").first().replaceWith("<p>replaced...</p>");
```

single note deletion with jQuery So, which function to use for a single note deletion with jQuery.

At the moment, we simply need to delete the selected note. We'll use the same `remove()` function as the *delete all* notes feature.

Next, we'll add an option per note to allow our user to delete a required note. This is similar to many options now available on mobile devices where a user swipes each list item to delete. However, they'll still need to click a button to delete.

create buttons We'll need to add a delete button for each note. This will need to be added programmatically, as each note is also being added dynamically.

For example,

```
function createButton(buttonClass, buttonText) {
  var $button = $('<button class="'+buttonClass+'">'+buttonText+'</button>');
  return $button;
}
```

This new function allows us to create simple buttons as required, including a specified class and button text passed as parameters. We can then use this function to build our required delete button in the `createNote()` function.

For example,

```
//create delete button
var $delete_button = createButton("note-delete", "delete");
```

We may implement the same logic for a button function with plain JS,

```
// create button element - pass class and text
function createButton(btnClass, btnTxt) {
    // create button node
    let btnNode = document.createElement('button');
    // create button text node
    let btnTxtNode = document.createTextNode(btnTxt);
    // set attribute on button node
    btnNode.setAttribute('class', btnClass);
    // append text to button
    btnNode.appendChild(btnTxtNode);
    // return new button node with text and attribute...
    return btnNode;
}
```

and then call as required,

```
// create delete button for note
let delButton = createButton('note-delete', 'delete');
```

using dynamic button creation Now, we could append this to the `.note-output` section, after creating each note, but then we'd have to check for the note and delete button each time we tried to delete all notes or even a single note.

This isn't difficult, it's pretty trivial in fact, but it's an unnecessary additional complication for our design. So, instead we can append the delete option to the note itself before adding that note to the DOM in the `createNote` function.

Again, there are many different ways we could achieve this same, basic result.

This is one possible solution with jQuery,

```
function createNote() {
...
//set content for note
$note.html($note_text.val());
//append delete button to each note
$note.append($delete_button);
...
}
```

and a working update for the `createNote()` function with plain JS,

```
function createNote(input, output) {
  // get value from input field for note
    let inputVal = input.value;

    // check input value
    if (inputVal !== '') {
    // create p node
    let p = document.createElement('p');
        // create delete button for note
        let delButton = createButton('note-delete', 'delete');
        // prepend button to note
        p.prepend(delButton);
        // create text node
        let noteText = document.createTextNode(inputVal);
        // append text to paragraph
        p.appendChild(noteText);
        // append new paragraph and text to existing note output
        output.appendChild(p);
        // call custom animation for fade in...
        //fadeIn(p);
        // clear input text field
        input.value = '';
    }

    let controls = document.getElementById('app-controls');
    checkVisible(controls);
}
```

delete button and click event With jQuery, we need to bind a click event to the dynamically created delete note button. This is slightly different than before, simply due to the fact that the delete button is being added to the DOM dynamically.

We can get around this by programmatically adding our handler for the single note deletion event to the parent `.note-output` , which does exist in the DOM.

```
$(".note-output").on("click", "button.note-delete" , function() {
  //delete parent note
  $(this).parent().remove();
  //set note selector
  var $note = $(".note-output p");
  //check for empty notes, and then remove note-controls
  if (checkExist($note) === false) {
    //hide note-controls
    $(".note-controls").hide();
  }
});
```

These updates may be seen in the following working example,

- DEMO 3 - travel notes - series 2

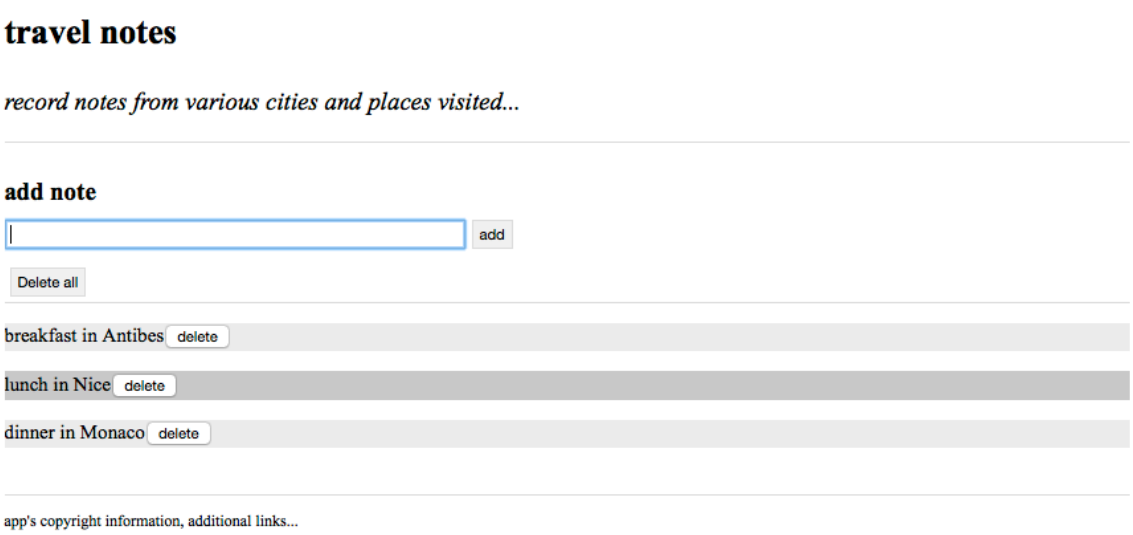

Figure 2: Travel Notes - Series 2 - Demo 3

select a note to delete So, we can now allow our users to delete a single note for all of their notes.

However, the single note option is awkward at the moment. Why do we need to show the delete button for each note, all of the time. Perhaps we could simply allow a user to either mouseover or select a note to show additional options. At the moment, this naturally means showing the available delete button.

To enable a user to select their note of choice, we need to bind a click event to a note, which should now look familiar with jQuery,

```
// handle click event per note
$(".note-output").on("click", "p", function() {
  ...
});
```

However, even if we allow a user to select a given note, and then show the delete button, we still have a design issue. As a user selects a note, we do not have a check for previous visible delete buttons.

For example, a user selects note 1 and then note 3. Both of these notes will now have a delete button, as shown in the following image.

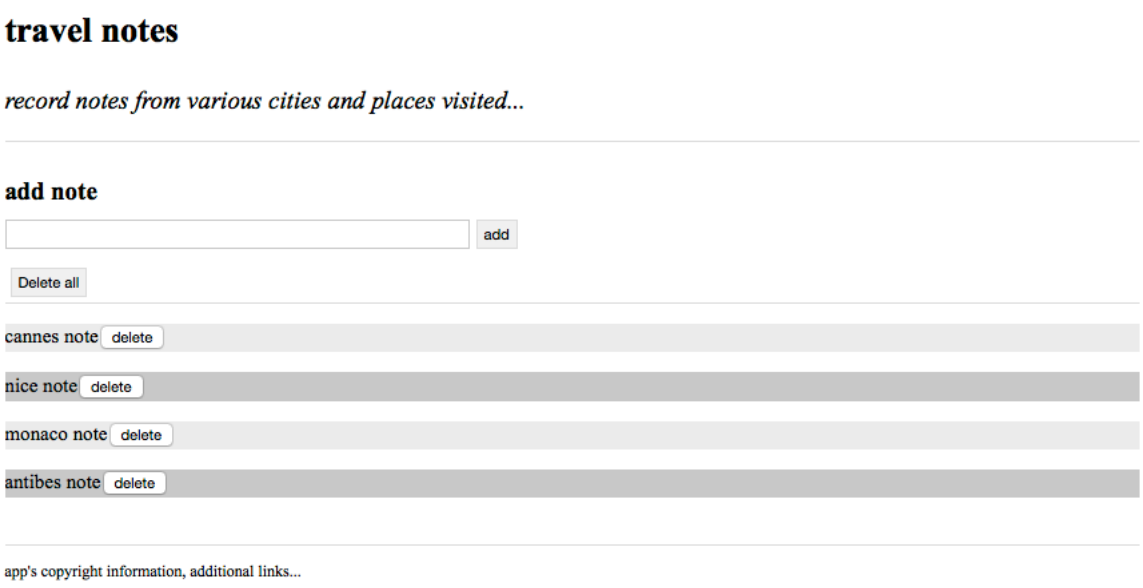

Figure 3: Travel Notes - Series 2 - Too many delete buttons

update check for visibility We'll now return to our earlier function, the `checkVisible()` function, and slightly modify to allow better abstraction and usage.

We can modify this function to test for visibility, and then simply return a boolean value. This returned result, true or false, can then be used to update the application accordingly.

```
//check element visibility - expects single element relative to display:none
function checkVisible(element) {
  //check if element is hidden or not
  if (element.is(":hidden")) {
    return true;
  } else {
    return false;
  }
}
```

As we update this function, we'll also need to modify our check for the `.note-controls` in the `createNote()` function,

```
...
//check visibility of note controls
if (checkVisible($(".note-controls")) === true) {
  $(".note-controls").fadeIn();
}
...
```

We may now also update the logic and functionality in the plain JS version of the app.

The first update is for the note delete button,

```
// add delete button for current note
// use anonymous FN instead of arrow FN
// this binds to clicked DOM node
delButton.addEventListener('click', function () {
    console.log('note delete...', this.parentNode);
    this.parentNode.remove();
});
```

We may update this note delete button further with a check for notes. For example, no notes in the app, we may then hide the *delete all* option to the user,

```
// add delete button for current note
// use anonymous FN instead of arrow FN
// this binds to clicked DOM node
delButton.addEventListener('click', function () {
    console.log('note delete...', this.parentNode);
    this.parentNode.remove();
    // get notes from DOM
    let notes = output.querySelectorAll('p');
    if (checkExist(notes) === false) {
        controls.style.display = 'none';
    }
});
```

Our updated plain JavaScript app is as follows,

- DEMO 3 - travel notes - series 2 - plain JS

update handler for note selection Our updated handler for note selection in the jQuery version can now check for visible delete buttons, and respond correctly.

For example,

```
//handle click event per note
$(".note-output").on("click", "p", function() {
  //check if other delete buttons visible
  if (checkVisible($("button.note-delete")) === true) {
    $("button.note-delete").hide();
  }
  $(this).children("button.note-delete").show();
});
```

So, after binding the handler for the user clicking on a note, we can check whether other delete buttons are visible on any other notes. If visible, we can simply hide these delete buttons, and only show the delete option for the currently selected note.

We may also later abstract this function to handle other options associated with each note. Perhaps an edit option, for example.

In the plain JS version, we may update this functionality as follows,

- check for current delete buttons per note

- hide each delete button
- then, show delete button for current note…

```
// click listener for note
p.addEventListener('click', function() {
    // get notes delete buttons from DOM
    let delBtns = output.querySelectorAll('.note-delete');
    if (checkExist(delBtns) === true) {
        for (let btn of delBtns) {
            btn.style.display = 'none';
        }
    }
    this.querySelector('.note-delete').style.display = 'inline';
});
```

We may test these updates in the latest examples for jQuery and plain JS,

- DEMO 4 - travel notes - series 2
  - jQuery
  - plain JS

Update app styles We also need to add some additional styling to our notes.

style notes We'll start with some changes to the design of each note, and then consider the overall `.note-output` section.

We'll remove the styling for alternating notes, and instead create a uniform output for each note.

```
/* note paragraph output */
.note-output p {
  margin: 10px;
  padding: 10px;
  border: 1px solid #b1c4b1;
  cursor:pointer;
}
```

Now, there are many possible ways we could style these notes. For now, we'll keep them as block elements, and allow the text in each note to simply wrap from one line to another.

style delete button Then we need to add some styling for our delete button, and position it within each note.

```
/* note delete button */
.note-output p button.note-delete {
  display: block;
  padding: 5px;
  margin: 5px 5px 10px 0;
  border-radius: 0;
  border: 1px solid #dedede;
  cursor: pointer;
}
```

We'll also add some styling for the button's hover pseudo-class, which acts as useful feedback to the user that the button is an active element.

```
.note-output p button.note-delete:hover {
  background-color: #aaa;
  color: #fff;
}
```

add feedback styles In the same vein, we might also consider adding some similar feedback to our note, simply as a sign of *active focus* as the user moves their mouse cursor over each note.

```
/* note paragraph output hover */
.note-output p:hover {
  border: 1px solid #1a3852;
}
```

So, we now have double feedback for our user with the change in cursor to indicate the note can be clicked, and a sense of active focus by updating the border colour.

We can also update this feedback as we continue to build our notes, images, and other content as a simple, but effective, way to nudge our users into usage patterns within the site or application.

We may test these updates styles in the following working example,

- DEMO 5 - travel notes - series 2

fix style issues There are, however, a couple of issues that still need to be fixed at this point.

The first is the lack of consistency in styling our buttons, and associated feedback for interaction. This first issue can be fixed by abstracting our CSS styling for a default button, and then including specific styles for each button as required.

```
/* default button style */
button {
  padding: 2px;
  margin: 2px;
  border-radius: 0;
  border: 1px solid #dedede;
  cursor: pointer;
}
```

We've now also removed the need for a ruleset to style the button for adding a note, delete all notes, and the single delete button per note.

Likewise, we can create a default ruleset for a button hover pseudo-class, again reducing our need for repetition in the stylesheet.

```
/* default button hover style */
button:hover {
  background-color: #aaa;
  color: #fff;
}
```

As you continue to develop your application or website, you'll come across many examples where it is possible to retrospectively abstract a style or some aspect of your application's logic. This is good practice, and you'll find it is not always possible to anticipate every aspect of your application that needs abstracting when you begin your design and development.

fix logic issues The second issue is the expected interaction with each note.

A user can currently click on a note, and view each available option. This is currently the single option to **delete** a note, but the issue is simply that a user cannot choose to remove this option.

In effect, they should be able to click on the note to show the options, and then click on the note again to hide these options.

We can update this interaction by modifying the handler for the click event on a note.

```
//handle click event per note
$(".note-output").on("click", "p", function() {
  //check if other delete buttons visible
  if (checkVisible($("button.note-delete")) === true) {
    //set all siblings to active=false to ensure checks are correct
    $(this).siblings().attr("active", "false");
    $("button.note-delete").hide();
  }
  //then handle click event for current note
  if (!$(this).attr("active") || $(this).attr("active") === "false") {
  $(this).attr("active", "true");
  $(this).children("button.note-delete").show();
  } else if ($(this).attr("active") === "true") {
  $(this).attr("active", "false");
  $(this).children("button.note-delete").hide();
  }
});
```

In essence, what are we now doing with this handler.

The first thing we do is check whether any of the other notes are currently showing their delete buttons. If other delete buttons are visible, we can either set or update a new attribute on each note as an indicator of active status for the options, our delete button at the moment.

Then, we perform a conditional check against this active attribute, again checking either for its existence, or value set to false. If this side of the conditional statement is correct, then we can set the value for our active attribute to true for the current note, and show the delete button. Otherwise, if the value of the active attribute is true, then we update it to false, and hide its delete button.

We may test these updates in app logic and functionality in the following working example,

- DEMO 6 - travel notes - series 2

A few extras to consider As we continue to develop this application, we may consider many extra options and features.

For example, we might add the following to our app

- alternative layouts
  - grid
  - squares
  - snippet view
  - table
  - lists…
- notifications
- snippets with expansion
- split views
  - note snippet with contextual/media per note…

- drag and drop delete
- filters
- sort options
- tags
- ...

Resources

- jQuery
  - jQuery
  - jQuery API
- JS
  - MDN - JS
  - MDN - JS Grammar and Types
  - MDN - JS Objects
  - W3 Schools - JS