Notes - Application Development - Client-side

Travel Notes - Part 1

- 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 1*, we may add the following functionality to this app

- basic app structure and logic
- add initial input options
- add event listeners for various input options
- create and add a note to the app
- various styles for the app, notes &c.
- add some initial code abstraction
- initial UI effects
- …

Initial Structure We're going to combine each of the three primary technologies for common client-side development, *HTML5*, *CSS*, and *JavaScript*, to create an example application with some simple interactive features.

We'll start with an outline of our project's basic directory structure, suitable for general websites and applications.

This is simply a boilerplate directory structure often used for starting to develop a website or application. This can, of course, be modified and updated to suit application requirements or personal design preferences.

We're not considering any build tools, such as Grunt, Gulp, or Webpack, at this stage.

```
.
|- assets
|  |- images //logos, site/app banners - useful images for site's design
|  |- scripts //js files
|  |- styles  //css files
|- docs
|  |- json //any .json files
|  |- txt //any .txt files
|  |- xml //any .xml files
|- media
|  |- audio //local audio files for embedding & streaming
|  |- images //site images, photos
|  |- video //local video files for embedding & streaming
|- index.html
```

Each of the above directories can, of course, contain many additional sub-directories.

For example,

- `|- images` may contain sub-directories for albums, galleries…
- `|- xml` may contain sub-directories for further categorisation..
- …

Home - `index.html` This is our initial template for our web application.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>travel notes - v0.1</title>
    <meta name="description" content="information on travel destinations">
    <meta name="author" content="ancientlives">
    <!-- css styles... -->
    <link rel="stylesheet" type="text/css" href="assets/styles/style.css">
  </head>
  <body>
    ...
    <!-- js scripts... -->
    <script type="text/javascript"      src="assets/scripts/jquery.min.js"></script>
    <script type="text/javascript" src="assets/scripts/travel.js"></script>
  </body>
</html>
```

We've set the initial metadata and title for the application, and added our required CSS files to the `<head>` element of our document. You'll also notice that we've now added our JS files to the foot of the *body*.

We've moved them to the end of the *body* element, instead of the *head* element, for a simple technical reason. In effect, as the browser renders the page, it works in a top-down hierarchical order. It creates DOM elements as it encounters them in the HTML document. Therefore, by placing our larger JS script files at the foot of the *body* element, these files become one of the last things to load.

An inherent benefit of this location for our JS files is that the user gains visual feedback from the rest of the loaded document as quickly, and efficiently, as possible.

The script files are stored within the local directory structure of our application, and will include the primary script code for our JS application. The `jquery.min.js` file is only required for the jQuery version.

We might also consider CDN references for various external JavaScript libraries.

The linked CSS file is also read from a local directory within the application.

document body We may update the initial *body* of the document for our app's home page as follows,

```
<body>
  <!-- document header -->
  <header>
    <h3>travel notes</h3>
    <p>record notes from various cities and placed visited...</p>
  </header>
  <!-- document main -->
  <main>
    <!-- note input -->
    <section class="note-input">
    </section>
    <!-- note output -->
    <section class="note-output">
    </section>
  </main>
  <!-- document footer -->
  <footer>
    <p>app's copyright information, additional links...</p>
  </footer>
  <!-- js scripts... -->
  <script type="text/javascript" src="assets/scripts/jquery.min.js"></script>
  <script type="text/javascript" src="assets/scripts/travel.js"></script>
</body>
```

We've added an app specific *header*, the *main* content area, and the app's *footer*.

The *main* content for this app initially includes two *sections*, the first for note input, and the second for the app's output of notes.

In effect, the first section will contain elements and tools to allow the user to create their notes, and the second section will organise and render those notes to the user.

initial styling We'll add the initial CSS styles to a stylesheet, `style.css` .

In this CSS file, we may add some basic styles for our site, keeping them as a simple base upon which we may continue to build our application. We've centred the layout of the basic frame for the *body*, and then we can later update this design to include a grid layout for additional content.

We've also set relative font-sizes to help us differentiate headings from *header* and *footer* paragraphs.

```
body {
  width: 850px;
  margin: auto;
  background: #fff;
  font-size: 16px;
  font-family: "Times New Roman", Georgia, Serif;
}

h3 {
  font-size: 1.75em;
}

header {
  border-bottom: 1px solid #dedede;
}

header p {
  font-size: 1.25em;
  font-style: italic;
}

footer p {
  font-size: 0.8em;
}
```

initial JavaScript As mentioned above, we may consider two alternative options for implementing our JavaScript structure and logic for this app. The first option will use the *jQuery* library, and the second will use plain JavaScript.

For the initial JavaScript file, `travel.js`, we've created a simple JS function to hold the basic logic for our app. We can call this function any reasonable, logical name, on a scale from `main` to the name of the app itself in *camelCase*. In effect, it should be something that represents the overall app name, concept, or a generic standard, again such as `main`.

Within this initial function, we set the `strict` pragma and add an example call to the jQuery function, `html()`, which sets some initial note content. This function will then be loaded each time the DOM is rendered, for example as the page loads in the browser, using the jQuery function `ready()`. There are many different ways to achieve this basic loading of JS, and it will often depend upon how we are using and working with JS relative to our app.

```
// overall app logic and loader...
function travelNotes() {
    "use strict";

    $(".note-output").html("<p>first travel note for Marseille...</p>");

};

$(document).ready(travelNotes);
```

We may also implement this logic and structure using plain JavaScript.

For example,

```
function travelNotes() {
  "use strict";

  // get a reference to `.note_output` in the DOM
  // n.b. these can be combined as well...
  let noteOutput = document.querySelector('.note-output');
  noteOutput.innerHTML = '<p>first travel note for Marseille...</p>';

}

// load app
travelNotes();
```

We may already see the similarities and differences in code usage and implementation between a JS library, and its various existing methods, and custom plain JS.

Our first working example is basic, but the app has started to take shape.

- DEMO 1 - travel notes - series 1

Add a note We've now tested the initial JavaScript, dynamically added some initial content, added some default styling, and rendered to the document. We can now create some additional functionality, and update the app's logic and styling.

If we consider the app's underlying structure, it should be clear that there are three semantic divisions of the app's content, `<header>` , `<main>` , and `<footer>` . Within the `<main>` content category, we can create and add our notes for our application.

To allow a user to create a new note, we'll obviously need some way for them to enter some brief text, and then set it as a note. At the moment, this note will, effectively, simply resemble a heading or brief description. We can expand, and add further options later.

initial note creation For this initial note creation, we can use the HTML element `<input>` , which includes some interesting new attributes in HTML5 such as *autocomplete*, *autofocus*, *required*, *width*, and so on. We can now add this element, and some associated text, to our app.

```
<h5>add note</h5>
<input>
```

```
<input type="text" value="add a note...">
```

tidy up the styling We also need to add some additional styles to create a correct, logical separation of visual elements and content.

We can add a border to the top of our *footer*, perhaps matching the *header* in style, and update the box model for the `<main>` element.

If we simply add a border to the top of the *footer*, without adjusting the `<main>` element accordingly, we will see an example of disappearing content. The *footer* and *main* will seem to merge into one, and so on, as we might expect with the logic of the box model.

So, we can update our CSS as follows,

- add h5 styles
- update `<main>`
- update `<footer>`

```
h5 {
  font-size: 1.25em;
  margin: 10px 0 10px 0;
}

main {
  overflow: auto;
  padding: 15px 0 15px 0;
}

footer {
  margin-top: 5px;
  border-top: 1px solid #dedede;
}
```

input update We may also update the CSS styles for the *input* field, and the associated *button*.

For example,

- remove button's rounded borders to match style of `input`
- match border for button to basic design aesthetics
- set cursor appropriate for a link style…

```
<input><button>add</button>
```

```
.note-input input {
  width: 40%;
}

.note-input button {
  padding: 2px;
  margin-left: 5px;
  border-radius: 0;
  border: 1px solid #dedede;
  cursor: pointer;
}
```

Our second working example with a few basic updates,

- DEMO 2 - travel notes - series 1

Interaction We've now added our input and associated button, and styled it with some general aesthetics. However, it still doesn't allow a user to create a new note, and add it to the output.

For this task, we need to add some JavaScript to handle the click event on the button. So, when a user clicks on the **add** button, they should be able to see the text entered in the input field rendered in the note output. We'll need to modify our `travel.js` script file to handle this event.

handle user event We'll start with the act of responding to a user clicking the button, which will output the result to the console for testing.

The following code snippet is for an event listener in jQuery. This example has attached a listener for a *click* event on the defined selector in the document. In effect, for the selected button, we are listening for a *click* event, such as a user clicking on the button.

```
//handle user event for `add` button click
$(".note-input button").on("click", function(e) {
  console.log("add button clicked...");
});
```

We may see the same functionality implemented using plain JavaScript.

```
let addNoteBtn = document.getElementById('add-note');
addNoteBtn.addEventListener('click', () => {
  console.log('add button clicked...');
});
```

initial output We can update the current jQuery code to better handle and output the text from the input field.

```
//handle user event for `add` button click
$(".note-input button").on("click", function(e) {
  $(".note-output").append("<p>sample note text...</p>");
});
```

So, what is this handler actually doing.

The jQuery code has attached an event listener to an element in the DOM, which is referenced in the selector option at the start of the function. It uses standard CSS selectors to find the required element, sometimes by name, class, ID, and so on. Therefore, jQuery allows us to easily select and target elements within the DOM using standard CSS selectors. We can then manipulate them, as required, using JavaScript.

To output some text to our `note-output` section, we can update our earlier jQuery. Each time a user clicks on the `add` button, at least they can now see some rendered text in the note output. However, it's static, and keeps outputting the same text.

We may update our plain JS code to match this functionality,

```
function travelNotes() {
  "use strict";

  // get a reference to `.note_output` in the DOM
  let noteOutput = document.querySelector('.note-output');
  // add note button
  let addNoteBtn = document.getElementById('add-note');

  // add event listener to add note button
  addNoteBtn.addEventListener('click', () => {
    // create p node
    let p = document.createElement('p');
    // create text node
    let noteText = document.createTextNode('sample note text...');
    // append text to paragraph
    p.appendChild(noteText);
    // append new paragraph and text to existing note output
    noteOutput.appendChild(p);
  });

}
```

At the moment, the code example for plain JavaScript seems a tad verbose in comparison. However, the difference is in the way we are adding the content to the document. In jQuery, this functionality is abstracted to a method `append()` . In plain JS, we define the logic in our own custom code.

However, we may later abstract this code as well to a custom function. At least, we can see and control exactly how we are adding content to the document.

We may see the latest update in the following working example,

- DEMO 3 - travel notes - series 1

get user input Whilst we may now listen for a user click on the `add` button, and output some text to `note-output` in the document, it's still not very useful as an application.

We need to abstract the code further, get some actual note text, and then render it in a meaningful way to the `note-output` section.

So, we can now define our shell for a rendered note, and then capture the content entered in the input field. Combine the two, and then output to the `note-output` section.

In our updated code, the first thing we do is create a new jQuery object to store the value of our note HTML wrapper. We use the `$` to help us define a variable as a jQuery object. This is not required, but it has become a useful option to help easily recognise jQuery specific logic from plain JS in such code.

Next, we need to get the text from the input field. We can then use this value to add to our new jQuery object, which becomes the formatted note for our `note-output` .

```
//overall app logic and loader...
function travelNotes() {
    "use strict";

  //handle user event for `add` button click
  $(".note-input button").on("click", function(e) {
    //object for wrapper html for note
    var $note = $("<p>");
    //get value from input field
    var note_text = $(".note-input input").val();
    //set content for note
    $note.html(note_text);
    //append note text to note-output
    $(".note-output").append($note);
  });

};

$(document).ready(travelNotes);
```

Likewise, we may see the same logic and structure used for our current plain JS example.

```
function travelNotes() {
  "use strict";

  // get a reference to `.note_output` in the DOM
  let noteOutput = document.querySelector('.note-output');
  // add note button
  let addNoteBtn = document.getElementById('add-note');
  // input field for add note
  let inputNote = document.getElementById('input-note');

  addNoteBtn.addEventListener('click', () => {
    // create p node
    let p = document.createElement('p');
    // get value from input field for note
    let inputVal = inputNote.value;
    // create text node
    let noteText = document.createTextNode(inputVal);
    // append text to paragraph
    p.appendChild(noteText);
    // append new paragraph and text to existing note output
    noteOutput.appendChild(p);
  });

}
```

The latest update may be seen in the following working example,

- DEMO 4 - travel notes - series 1

clear the input field Our example is steadily improving, but it still needs some finishing touches, and a couple of standard features that a user would expect for such interactions. These include an input field that clears as focus is applied, and the option to simply press the return key after typing a new note, and then see it loaded in the `note-output` .

So, let's start by clearing the input field, and then we can add an event listener for the return key.

Clearing the input field is, thankfully, straightforward in principle. However, we need a conditional check to ensure we do not simply clear the input field each time. That is both a waste of resources, and bad design.

In our conditional statement, we simply check whether the input field is empty or not, handle the text if present, and then clear it after appending it to the `note-output` .

We may update our jQuery example as follows,

```
//overall app logic and loader...
function travelNotes() {
    "use strict";

  //handle user event for `add` button click
  $(".note-input button").on("click", function(e) {
    //object for wrapper html for note
    var $note = $("<p>");
    //define input field
    var $note_text = $(".note-input input");
    //conditional check for input field
    if ($note_text.val() !== "") {
    //set content for note
    $note.html($note_text.val());
    //append note text to note-output
    $(".note-output").append($note);
    //clear the input field
    $note_text.val("");
  }
  });

};

$(document).ready(travelNotes);
```

and the plain JS version as follows,

```
function travelNotes() {
  "use strict";

  // get a reference to `.note_output` in the DOM
  let noteOutput = document.querySelector('.note-output');
  // add note button
  let addNoteBtn = document.getElementById('add-note');
  // input field for add note
  let inputNote = document.getElementById('input-note');

  // add event listener to add note button
  addNoteBtn.addEventListener('click', () => {
    // create p node
    let p = document.createElement('p');
    // get value from input field for note
    let inputVal = inputNote.value;

    // check input value
    if (inputVal !== '') {
      // create text node
      let noteText = document.createTextNode(inputVal);
      // append text to paragraph
      p.appendChild(noteText);
      // append new paragraph and text to existing note output
      noteOutput.appendChild(p);
      // clear input text field
      inputNote.value = '';
    }
  });

}
```

This update may be seen in the following working example,

- DEMO 5 - travel notes - series 1

add a keyboard listener As noted, we also need to consider how to handle keyboard events, in particular listening and responding to a user hitting the return key after entering text into the input field for a new note.

In essence, the pattern is similar to the way we handled a listener for the earlier click event on the **add button**.

We could repeat the code used for handling the button click, but there is an inherent repetition and waste in such a code design.

One of the issues with adding a listener for keyboard events is that we naturally need to be selective with regard to keys pressed. Therefore, we can add a conditional check to our listener for a specific key, in this example our required **return** keypress.

By using the local variable from the event itself, in this example `e` , we can get the value of the key pressed, and then check against the required logic of the app. We simply select the required keycode, and compare against the keys pressed by the user until we find the one we want.

The rest of the function can work in the same underlying manner as the click handler on the **add** button.

The jQuery event handler for a *keypress*,

```
$(".note-input input").on("keypress", function (e) {
  if (e.keyCode === 13) {
    ...
  }
});
```

and the plain JS example,

```
// add event listener for keypress in note input field
inputNote.addEventListener('keypress', (e) => {
  // check key pressed by code - 13 - return
  if (e.keyCode === 13) {
    console.log('return key pressed...');
  }
});
```

Notice that the underlying logic to check for the key pressed,

```
// check key pressed by code - 13 - return
if (e.keyCode === 13) {
  console.log('return key pressed...');
}
```

is the same for both listeners, jQuery and plain JS.

We may also see this functionality used to track user inputs in an example text editor. The output in the right sidebar, for example, shows full details for each key press and user action.

- example recording keypresses
- Demo Editor

Abstract code We now need to create a new function that will simply abstract the creation and output of a new note, and manage the input field for our note app.

In essence, we're moving the logic from our button click function to a separate, abstracted function. We can then call this function as needed, for example in response to a button click or keyboard press, and then create and render the new note.

Our new function is as follows for our jQuery version,

```
//manage input field and new note output
function createNote() {
  //object for wrapper html for note
  var $note = $("<p>");
  //define input field
  var $note_text = $(".note-input input");
  //conditional check for input field
  if ($note_text.val() !== "") {
  //set content for note
  $note.html($note_text.val());
  //append note text to note-output
  $(".note-output").append($note);
  //clear the input field
  $note_text.val("");
  }
}
```

The current working code for our jQuery version may now be updated as follows,

```
//overall app logic and loader...
function travelNotes() {
  "use strict";

  //manage input field and new note output
  function createNote() {
    //object for wrapper html for note
    var $note = $("<p>");
    //define input field
    var $note_text = $(".note-input input");
    //conditional check for input field
    if ($note_text.val() !== "") {
    //set content for note
    $note.html($note_text.val());
    //append note text to note-output
    $(".note-output").append($note);
    //clear the input field
    $note_text.val("");
    }
  }

  //handle user event for `add` button click
  $(".note-input button").on("click", function(e) {
    //call custom function
    createNote();
  });

  //handle user event for keyboard press
  $(".note-input input").on("keypress", function(e){
    if (e.keyCode === 13) {
      //call custom function
      createNote();
    }
  });
};
$(document).ready(travelNotes);
```

The plain JS version of the app's current code may also be updated with an abstracted `createNote()` function.

For example,

```
function travelNotes() {
  "use strict";

  // get a reference to `.note_output` in the DOM
  let noteOutput = document.querySelector('.note-output');
  // add note button
  let addNoteBtn = document.getElementById('add-note');
  // input field for add note
  let inputNote = document.getElementById('input-note');

  // add event listener to add note button
  addNoteBtn.addEventListener('click', () => {
      //call custom function & pass required arguments
      createNote(inputNote, noteOutput);
  });

  // add event listener for keypress in note input field
  inputNote.addEventListener('keypress', (e) => {
    // check key pressed by code - 13 - return
    if (e.keyCode === 13) {
      //call custom function & pass required arguments
      createNote(inputNote, noteOutput);
    }
  });

}

// create a note
// - input = value from input field
// - output = DOM node for output of new note
function createNote(input, output) {
    // create p node
    let p = document.createElement('p');
    // get value from input field for note
    let inputVal = input.value;
    // check input value
    if (inputVal !== '') {
      // 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);
      // clear input text field
      input.value = '';
    }
}

// load app
travelNotes();
```

An example of the current version of the app is as follows,

- DEMO 6 - travel notes - series 1

Add some animation One of the things jQuery is well-known for is its simple ability to animate elements, including *show* and *hide*. There are many built-in effects available in jQuery and, of course, we can always create our own as needed.

So, let's finish off this set of JS code with some simple fade animations.

To be able to `fadeIn` an element, effectively it needs to be hidden first. So, the first thing we do is hide our newly created note, and then we can set it to `fadeIn` when ready.

There are many additional parameters for jQuery's `fadeIn` function, and we can customise a callback, for example, change the speed of the animation, and so on.

```
//manage input field and new note output
function createNote() {
  //object for wrapper html for note
  var $note = $("<p>");
  //define input field
  var $note_text = $(".note-input input");
  //conditional check for input field
  if ($note_text.val() !== "") {
  //set content for note
  $note.html($note_text.val());
  //hide new note to setup fadeIn...
  $note.hide();
  //append note text to note-output
  $(".note-output").append($note);
  //fadeIn hidden new note
  $note.fadeIn("slow");
  $note_text.val("");
  }
}
```

The following working example shows this animation effect for a new note,

- DEMO 7 - travel notes - series 1

Update some styles Now that we have some new notes in our app, we can add some styling to help improve the look and feel of our basic app.

We can set background colours, borders, font styles, and so on.

For this example, as we have the option of many new notes, we can set differentiating colours for each alternate note. This also allows us to try some pseudoclasses in the CSS for specified paragraphs in the `note-output` section.

```
.note-output p:nth-child(even) {
  background-color: #ccc;
}
.note-output p:nth-child(odd) {
  background-color: #eee;
}
```

Our update styles can be tested in the following example,

- DEMO 8 - travel notes - series 1

These are initial styles, which we may update in a later version to use CSS Grids and Flexbox.

Additional options to consider We now have a basic app that records simple notes. There are many additional options we can add, and some basic functionality is needed to make it useful.

For example,

- autosave - otherwise we lose our data each time we refresh the browser
- edit a note
- delete a note
- add author information
- additional functionality might include
  - save persistent data to DB, name/value pairs…
  - organise and view collections of notes
  - add images and other media
    - local and APIs
  - add contextual information
    - again, local and APIs
  - structure notes, media, into collection
  - define related information
  - search, sort…
  - export options and sharing…
- security, testing, design patterns
- …

Suffice to say, there's a lot we might consider adding to this type of app.

Resources

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