Notes - Application Development - Client-side

Travel Notes - Part 3

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

- asynchronous query and update
- JSON loading and rendering
- reading and rendering app data
- …

AJAX and JSON The best way to consider AJAX is as a simple way to load data, often new or updated data, into a current page without having to refresh the browser window. One of the most common forms of data for work with AJAX is JSON, and this combination is often cited as helping in the rise of single page applications.

There are many common usage scenarios and examples for AJAX, including autocomplete in forms, live filtering of search queries, and real-time updates for content and data streams. We can also use AJAX to help us load data behind the scenes, in effect preparing content for our users before a specific request is received, thereby speeding up page responses and data load times.

As you might imagine from the name, AJAX uses an asynchronous model for processing requests. In essence, this means the user can continue to perform various tasks, queries, and work whilst the browser itself continues to load data. The inherent benefit should be a more responsive, intuitive usage and interface experience.

asynchronous model Compared to a more traditional synchronous model, which normally stops a page until it has loaded and processed a requested script, AJAX enables a browser to request data from the server without this pause in usage. Therefore, AJAX's **asynchronous processing model**, often known as **non-blocking**, allows a page to load data and process user's interactions.

As the server responds with the requested data, an event will be fired, which can then call a function to process the data. This allows us to simply update a single element, for example, with new data.

In essence, therefore, AJAX works as follows. The browsers submits a request for data from a given server. This server then responds with data, often JSON, XML, or simply HTML, which the browser processes as content to be added to the current page.

The browser will use an **XMLHttpRequest** object to help handle these AJAX requests. Once it submits a request to the server, it will not simply wait for a response. As mentioned, as the server responds to this request, the browser will use an event, often to trigger a JS function. This JS function will be used to process the returned data, which can then be used within the page.

Video example,

- AJAX requests & Twitter - UP TO 2:55
- Source - What is AJAX? - YouTube

JSON and jQuery Let us now try some AJAX with a JSON file. We'll test it first within a very basic HTML page.

initial setup First thing we need to do is create our simple JSON file as follows,

```
{
  "country":"France",
  "city":"Marseille"
}
```

and save this content to our `docs/json/trips.json` file. We'll be running these demos using a server. This is due to browser security restrictions for JavaScript. It's a legacy issue relating to what JavaScript can and cannot access on a local machine. It's there for a good reason, so if you want to test such code simply run examples using a local server. If not, you can also disable browser safeguards, for example in Chrome, and then load these files locally in a browser.

For local servers we can user XAMPP, for example, or simply load our pages using Python's **SimpleHTTPServer**,

```
python -m SimpleHTTPServer 8080
```

We may also run this simple server using Python 3,

```
python3 -m http.server 8080
```

The port number of the local server may be customised. For example, we might run the local server at port `4040` and so on.

We can also run a basic local server with Node.js, for example.

So, we now need to add our JavaScript function to help us access this content from our JSON file. We'll initially use the `getJSON()` function to test reading this content.

```
$.getJSON("docs/json/trips.json", function(trip) {
console.log(trip);
});
```

Console output is as follows,

```
Object { country: "France", city: "Marseille" }
```

We can now update our app's code to use the `trip` object, and output its name / value pairs.

test with site We can now use this return object to load our data as required within a site. For example, we can modify the JavaScript as follows,

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

  $.getJSON("docs/json/trips.json", function(trip) {
    //element for trip data
    var $tripData = $("<p>");
    //add some content from json to element
    $tripData.html(trip.city + ", " + trip.country);
    //append content to .note-output section
    $(".note-output").append($tripData);
  });
};

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

- DEMO - AJAX 1 - AJAX - demo 1

array for trips Whilst the previous example is useful, for our application we obviously need to store multiple trips. So, multiple countries, multiple cities, and so on. Therefore, we need to consider working with JSON arrays. We'll update our `trips.json` file as follows to test loading cities,

```
{
    "cities": [
        {
            "name": "Marseille",
            "region": "Provence-Alpes-Côte d'Azur"
        },
        {
            "name": "Paris",
            "region": "Île-de-France"
        }
    ]
}
```

load an array for trips We can update our JavaScript to load the array and set the data as required for the application.

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

  $.getJSON("docs/json/trips.json", function(trips) {
    //element for trip data
    var $cityData = $("<ul>");

    //iterate over cities array - trips.cities...
    var $cities = trips.cities;
    $cities.forEach(function (item) {
      var $city = $("<li>");
      $city.html(item.name + " in the region of " + item.region);
      $cityData.append($city);
    })
    //append list to .note-output
    $(".note-output").append($cityData);
  });
};

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

- DEMO - AJAX 2 - AJAX - demo 2

jQuery Deferred jQuery provides a useful solution to the escalation of code for asynchronous development.

This is known as the `$.Deferred` object. This **deferred** object effectively acts as a central despatch and scheduler for our events.

With the **deferred** object created, certain parts of the code indicate they need to know when an event completes, whilst other parts of the code signal an event's status. It's this **deferred** nature that coordinates these different activities, thereby enabling us to separate how we trigger and manage events from having to deal with their consequences.

using deferred objects We can now update our AJAX request with **deferred** objects.

As such, our intention is to separate the initiation of the event, the AJAX request, from having to deal with its consequences, essentially processing the response.

With this separation in logic, we no longer need a success function acting as a callback parameter to the request itself. Instead, we can now rely on the simple fact that the `.getJSON()` call returns a **deferred** object.

The function, effectively, returns a restricted form of this **deferred** object, which is known as a **promise**.

So, we're now able to store the returned object in a variable,

```
deferredRequest = $.getJSON (
  "file.json",
  {format: "json"}
);
```

Then, we can indicate our interest in knowing when the AJAX request is complete and ready for use,

```
deferredRequest.done(function(response) {
  //do something useful...
});
```

The key part of this logic is the `done()` function. Effectively, it is specifying a new function that we want to execute each and every time the event, our AJAX request in this example, is successful and returns complete.

The **deferred** object is able to handle the abstraction within the logic to make our lives a lot easier. In particular, if the event is already complete by the time we register the callback via the `done()` function, our **deferred** object will execute that callback immediately. If not, it will simply wait until the request is complete.

handling errors with deferred objects In addition, we can also signify interest in knowing if the AJAX request fails. So, instead of simply calling `done()` , we can use the `fail()` function. This will still work with JSONP, as the request itself could fail, and therefore be the reason for the error or failure.

```
deferredRequest.fail(function() {
  //report and handle the error...
});
```

Travel Notes - example usage We'll now add the option to read and write from a JSON file. Naturally, we'll use AJAX for these requests.

So, initially we can consider our application as follows,

- read data from JSON file
- load initial data to application

We can add edit functionality and options as we move our data to a DB.

JSON So, let's start by testing that we can actually read and load the data from our JSON file.

We'll be ignoring the standard AJAX pattern, which is normally based upon passing two callbacks, one for success and another for error, and instead we will be using the `deferred` and `promise` option we've just looked at.

Our initial basic JSON will be as follows,

```
{
  "travelNotes": [{
    "created": "2015-10-12T00:00:00Z",
    "note": "a note from Cannes..."
  }, {
    "created": "2015-10-13T00:00:00Z",
    "note": "a holiday note from Nice..."
  }, {
    "created": "2015-10-14T00:00:00Z",
    "note": "an autumn note from Antibes..."
  }]
}
```

deferred We'll start off by simply submitting a query for the required JSON file, and then retain the deferred object we're using for tracking.

We can then indicate our interest in knowing when this AJAX request is complete.

```
//load main app logic
function loadApp() {
  "use strict";

    var $deferredNotesRequest = $.getJSON (
      "docs/json/notes.json",
      {format: "json"}
    );

    $deferredNotesRequest.done(function(response) {
        console.log("tracking json...");
    });

};
$(document).ready(loadApp);
```

So, at the moment, our application will simply output to the console. As mentioned earlier, the `done()` method is the key part. It helps us specify the required logic to execute when the request completes. Also, if the given event has already completed when we register the callback via the `done()` method, our deferred object will execute the required callback immediately. If not, it will simply wait until the request is complete.

We can also now see how to respond to a failure, in a similar manner to the `done()` method. So, we can add a `fail()` method to the handler and report errors.

work with data - part 1 For the returned data, our response will naturally return an object containing an array with our notes. We could then simply extract the required notes, and append them to the DOM.

```
$deferredNotesRequest.done(function(response) {
    //get travelNotes
    var $travelNotes = response.travelNotes
    //process travelNotes array
    $travelNotes.forEach(function(item) {
      if (item !== null) {
        var note = item.note;
        //create each note's <p>
        var p = $("<p>");
        //add note text
        p.html(note);
        //append to DOM
        $(".note-output").append(p);
      }
    });
});
```

- DEMO - ajax & json basic loader

We end up with each note being injected into the DOM for our test page.

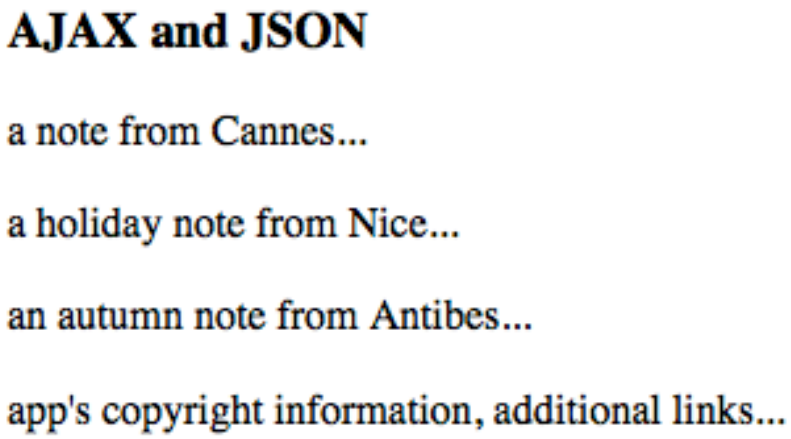

Figure 1: AJAX & JSON - basic loader

As we're working with known local data, and not much of it at the moment, we can get away with simple deferred requests. We'll update and improve these examples later.

However, if we were to call a remote API, which may have staggered API calls to data, we would need to use a slightly modified approach. In effect, we would be digging through the data layer by layer, submitting a request as one layer returned successfully.

For this type of processing with remote APIs, we could create a second deferred object, which we could use to track additional processing requests. Effectively, this allows us to stagger our requests to the API, thereby ensuring that we only request certain data as and when it is needed or available. We can also create multiple deferred objects to handle our requests and returned data, which again allows us to respond accordingly within the application.

jQuery resolve and promise methods So, let us now consider how to use explicit `resolve()` and `promise()` methods. First, though, what is the difference between these methods?

- `resolve()`

We can use this method with the deferred object to change its state, effectively to complete. As we resolve a deferred object, any **doneCallbacks** that may have been added with the `then()` or `done()` methods will be called. These callbacks will then be executed in the order they were added to the object. Another thing to note, any arguments supplied to the `resolve()` method will subsequently be passed to these callbacks as well.

- `promise()`

The `promise()` method is useful for limiting or restricting what can be done to the deferred object.

```
function returnPromise() {
  return $.Deferred().promise();
}
```

In effect, this method returns an object with a similar interface to a standard `deferred` object, but now it only has methods to allow us to attach callbacks. Importantly, it does not have the methods required to resolve or reject the deferred object. Therefore, we are restricting the usage and manipulation of the deferred object. For example, we may offer an API or other request the option to subscribe to the deferred object, but they won't be able to resolve or reject it as standard. Therefore, if we now return a deferred object with the `promise()` method, our logic will fail if we try to resolve that object.

```
...
returnPromise().resolve("test");
...
```

work with data - part 2 However, we can still use the `done()` and `fail()` methods as normal.

As mentioned, we can use additional methods with these callbacks including the `then()` method. We can use this method to return a new promise, which we can use to update the status and values of the deferred object. We can use this method to modify or update a deferred object as it is resolved, rejected, or still in use.

We might also want to combine, for example, our promises with the `when()` method. This method allows us to accept many promises, and then return a sort of master *deferred*. This updated `deferred` object will now be resolved when all of the promises are resolved, and it will likewise be rejected if any of these promises fail. We then use the standard `done()` method to be able to work with the results from all of the promises.

For example, we could use this pattern to combine results from multiple JSON files. We might also need to call different layers within an API based upon returned results, or send staggered requests for pages results.

So, we can update our application to test this pattern.

We can now start to update our test AJAX and JSON application. We'll begin by simply abstracting our code a little,

```
function buildNote(data) {
  //create each note's <p>
  var p = $("<p>");
  //add note text
  p.html(data);
  //append to DOM
  $(".note-output").append(p);
}

//get the notes JSON
function getNotes() {
  //.get returns an object derived from a Deferred object - do not need explicit deferred object
  var $deferredNotesRequest = $.getJSON (
  "docs/json/notes.json",
  {format: "json"}
  );
  return $deferredNotesRequest;
}
```

We can add new functions to handle fetching the data, and then rendering the returned data.

- DEMO - ajax & json abstract loader

As we're requesting our JSON file using `.getJSON()`, we get a returned promise for the data. This means, as mentioned earlier, we are only able to use the deferred object's method required to attach any additional handlers or determine its state. Therefore, we can work with

- `then`, `done`, `fail`, `always` …

However, we can't work with any deferred methods that change the state, including

- `resolve`, `reject`, `notify` …

When working with AJAX and JSON, one of the benefits of using **promises** is the ability to load one JSON file, wait for the results, and then issue a follow-on request to another file, for example.

So, for our example, we could now use one JSON file for countries and places, and another to hold the notes. As we load the countries, and places, we can then search the notes for those results (the countries and places etc…).

First things first, let's see how we can now chain some `then()` methods.

```
getNotes().then(function(response1) {
  console.log("response1="+response1.travelNotes[2].note);
  $(".note-output").append(response1.travelNotes[2].note);
  return getPlaces();
}).then(function(response2) {
  console.log("response2="+response2.travelPlaces[2].place);
  $(".note-output").append(response2.travelPlaces[2].place);
});
```

We're only outputting a test result to the DOM and the console, and a limited result at that, but at least we can now see how chained `then()` works.

As we chain our `then()` methods, we pass along any returned results from the previous method as an argument for the next `then()` method. So, our returned result from the `getNotes()` method is output in the first `then()` method, we return a result from the `getPlaces()` method, and pass this along to the second `then()` method. We can use this returned result to manipulate, search, output, and so on.

- DEMO - ajax & json deferred .then()

Demos

- DEMO - AJAX 1 - AJAX - demo 1
- DEMO - AJAX 2 - AJAX - demo 2
- DEMO - Travel Notes - ajax & json basic loader
- DEMO - Travel Notes - ajax & json abstract loader
- DEMO - Travel Notes - ajax & json deferred .then()

Resources

- jQuery
  - jQuery
  - jQuery API
- JS
  - MDN - JS
  - MDN - JS Objects
  - W3 Schools - JS
- What is AJAX? - YouTube