ADVANCED JAVASCRIPT FOR WEB SITES AND WEB APPLICATIONS

SESSION 5 - AJAX PART 1

Building a product listing page with filters and pagination - part 1

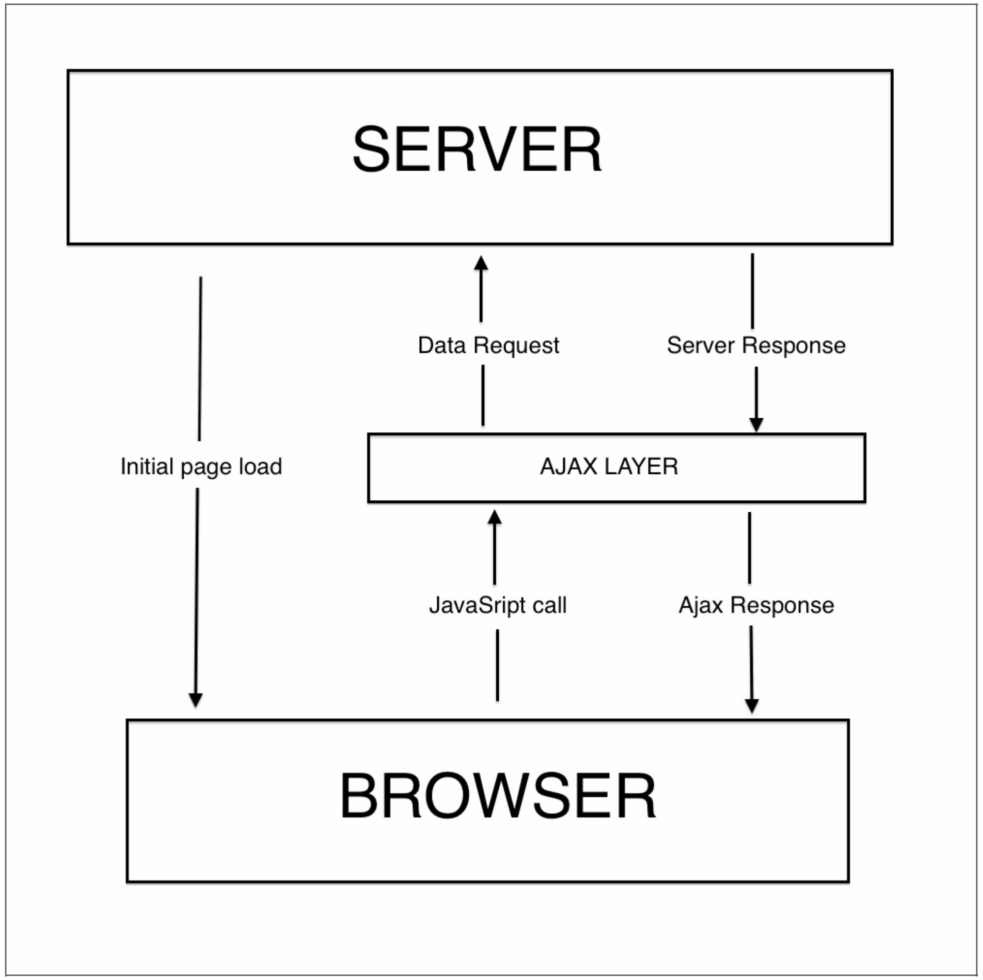
A BRIEF DESCRIPTION

Ajax lets you make calls to the server directly from a web browser without needing a page reload.

This can be usefull to make quick and small calls to the server and display the results, without waiting for a page refresh each time. For example on an autocomplete input field.

You can also use it to load content after the initial page load, when a user interacts with the page, to minimise the initial load time.

EXAMPLE DIAGRAM



Ajax works by using the `XMLHttpRequest` object. It can retrieve data through http, ftp or file protocol, in different formats (xml, html, json, text etc.).

This is how the object works:

- create a new instance of the object
- pass this new object the URL to query and the method (GET, POST etc.)
- do something with the result

```
function reqListener () {
    console.log(this.responseText);
}

var oReq = new XMLHttpRequest();

oReq.onload = reqListener;

oReq.open("get", "yourFile.txt", true);

oReq.send();
```

Browser quirks and differences in the implementation of `XMLHttpRequest` make it hard to use writing simple code.

We'll look at an example using jQUery, but more information on the native JavaScript object can be found on the **Mozilla Development Network**.

Ajax calls with jQuery can be made the following way:

```
$.ajax({

  url: "url_as_string",

  dataType: "data_type_as_string",

  success: function (data) {
    // do something here once the call has finished
  }

});
```

`url` will be the api url, passing the correct parameter.

The data type we use is `json`.

And the `success` function will run once the call has been made and we have the data from the call.

That data can be passed as an argument of the `success` function (in the previous example, with the argument name "data").

PRODUCT LISTING - PART 1

Let's build a simple product list, like Amazon do, that we can filter by type of products.

We'll list the categories: books, phones, laptops and toys.

We'll filter the list and display only the products we are interested in by making an ajax call to a products api.

You can find **an example** of the final working script.

Here is how the application needs to work:

- when a user clicks on one of the filter button, make an ajax call to the api, passing the category we want
- when the result comes back, loop through the JSON and template the data in HTML
- add the HTML to the page

We'll get the data from this api:

http://staff.city.ac.uk/~sbbh718/api/products-list/products.php?callback=&category=phone

In the category parameter, you can pass "phone", "laptop", "book" or "toy".

Here is one way to break down the code:

- use the revealing module pattern
- use an `init` function that will be public, all other functions can be private to the app
- in the `init` function, declare the event handlers for the filters
- create one "ajax" function to take care of the call
- create one "json to html" function that takes data as json and returns the data as html
- create one "html append" function that takes the html created and adds it to the dom

Part 1

Start with the following revealing module pattern:

```
var PRODUCTLIST = (function () {

  var

    init = function () {
    };

  return {
    init: init
  };

}());

PRODUCTLIST.init();
```

Part 2

Declare the three functions we need. Only declare them for now, in the single `var` declaration.

We can call them `jsonTOhtml`, `addHTMLtoDOM`, `callAjax`.

We will later use the `callAjax` function as the event handler function.

```
var jsonTOhtml = function () {
    },

    addHTMLtoDOM = function () {
    },

    callAjax = function () {
    },

    init = function () { [...]
```

Part 3

Declare the event handlers for the 5 filters.

You can declare one event handler per filter, or run a `for` loop to do this more efficiently.

Or declare one event handler and use event delegation!

The event handler(s) call the function `callAjax`.

```
init = function () {

    filter[0].addEventListener(
      "click", callAjax
    );

    filter[1].addEventListener(
      "click", callAjax
    );

    filter[2].addEventListener(
      "click", callAjax
    );

    filter[3].addEventListener(
      "click", callAjax
    );

    filter[4].addEventListener(
      "click", callAjax
    );
};
```

```
var filters = document.querySelectorAll(".filters")[0];

init = function () {
    filters.addEventListener(
      "click", callAjax
    );
};
```

Part 4

In the `callAjax` function, make a call using jQuery's ajax method. The url will be the api url and the dynamic category. The category we want to pass is stored in the `data-category` attribute of each html filter element.

The `dataType` will be `json`.

Remember that this function is the event handler function, and so creates an event object argument. This object holds information aobut the element that was just clicked (the filter), using the "target" property.

Once you have the filter element itself, you can read its attribute using `getAttribute("data-category")`.

We'll then use the `success` method of the ajax call to call the `jsonTOhtml` function, passing the json data from the api.

In the case of this api, console.log the data object that the ajax call returns. In the object, we are interested in the property `responseJSON.products`.

```
callAjax = function (event) {

  var
    category = event.target.getAttribute("data-category");

  $.ajax({
    url: "http://staff.city.ac.uk/~sbbh718/api/products-list/products.php
    dataType: "json",
    success: function (data) {
      jsonTOhtml(data.products);
    }
  });

},
```

If you have used event delegation:

```
callAjax = function (event) {

var
  filter = event.target,
  category = filter.getAttribute("data-category");

  if (filter.classList.contains("filter")) {

    $.ajax({
      url: "http://staff.city.ac.uk/~sbbh718/api/products-list/products.p
      dataType: "json",
      success: function (data) {
        jsonTOhtml(data.products);
      }
    });
  }

},
```

Part 5

We now need to populate the `jsonTOhtml` function.

This function takes one argument (the JSON data passed from the Ajax call). It needs to loop through the object (you can use a normal `for` loop).

For each item it finds in the JSON, we need to build a html string like this:

```
<li class="phone"><a href="url">Item name</a> - <span>Price: Item price</
```

We'll also need to add a `ul` with a class of "products" around all the `li's`.

Once our html string is stored in an array, we can join the array and pass the final html string to our `addHTMLtoDOM` function.

```
jsonTOhtml = function (JSONdata) {

  var i, productsInHtml, currentItem, finalHtml, productHtml = ""
    productsHtml = [];

  productsHtml.push("<ul class=\"products\">");

  for(i = 0; i < JSONdata.length; i ++) {
    currentItem = JSONdata[i];
    productHtml = "<li class=\"" + currentItem.category
            + "\"><a href='" + currentItem.url + "'>" + currentItem.n
            + "</a> - <span>Price: " + currentItem.price + "</span>";
    productsHtml.push(productHtml);
  }

  productsHtml.push("");

  finalHtml = productsHtml.join("\n");

  addHTMLtoDOM(finalHtml);
},
```

Part 6

We can now populate the `addHTMLtoDOM` function.

This function takes one argument (the html string passed from the `jsonTOhtml` function), and add it inside the `div` with the id of "products-list".

Since this html will be added every time we need we use one of the buttons we need to remove any `ul` previously added, if it exists, before adding anything.

Do a simple `if` statement to check if any `ul` with a class of "products" exists under the `div` and remove it!

```
addHTMLtoDOM = function (html) {
  var exisitingProducts = listOfProducts.querySelectorAll(".products");

  if (exisitingProducts.length !== 0 ) {
    exisitingProducts[0].parentNode.removeChild(exisitingProducts[0]);
  }
  listOfProducts.insertAdjacentHTML("afterbegin", html);
},
```

Bonus part 2

You will notice that when we click on one of the fliters, we have no feedback at the moment of what is happening. The html is being updated eventually, but the user doesn't know that this is happening behind the scene.

We can do the following:

- as soon as someone clicks one of the filters, before doing anything else, remove any `ul` product list (previously, we were doing this at the very end)
- add a small loading image to indicate that something is happening. Add a class of "loading" to the "products-list" div
- once the new html is then ready to be added to the page, remove the image (by removing the class) and add the html in as before!

```
callAjax = function (event) {

  var
    category = event.target.getAttribute("data-category"),
    exisitingProducts = listOfProducts.querySelectorAll(".products");

  if (exisitingProducts.length !== 0 ) {
    exisitingProducts[0].parentNode.removeChild(exisitingProducts[0]);
  }

  listOfProducts.classList.add("loading");

  $.ajax({ ...
```

```
addHTMLtoDOM = function (html) {

  listOfProducts.insertAdjacentHTML("afterbegin", html);

  listOfProducts.classList.remove("loading");

},
```

Bonus part 3

At the moment, if the api returns an error, the script will not handle it.

Two typical errors can be that:

- the category isn't passed correctly to the api (although it shouldn't happen...)
- the ajax calls fails and returns a 404 or time out (network issues)

For the first case, we can check the data object returned in the success method. If it contains an "error" property, the category passed was unexpected

For the second case, we can add an "error" method to our ajax call. Check the **jQuery documentation**!

To make sure we trigger an error, add a filter that doesn't exist:

```
<p class="filter" data-category="television">Television</p>
```

```
success: function (data) {

  if(data.error) {
    listOfProducts.classList.remove("loading");
    listOfProducts.insertAdjacentHTML(
      "afterbegin",
      "<p class=\"products\">There was an error in the call<\/p>"
    );
  }

  jsonTOhtml(data.products);
}
```