

ĆWICZENIE 5

Aplikacja AJAX (klient dla serwisu REST)

Author: Mariusz Fraś

Objectives of the exercise

The purpose of the exercise is:

1. Acquaintance with architecture of AJAX based Web applications.
2. Mastering the technique of building AJAX client for REST service.

Development environment

The developing of applications can be performed using various platforms. Here is considered the following environment:

- Windows 7 (or higher) Operating System (eventually MacOS).
- Java Software Development Kit (JDK 11 or newer) and Java application development platform (here: IntelliJ IDEA).
or
- Visual Studio 2017/2019/2021
- Any editor for HTML doc and Web Browser

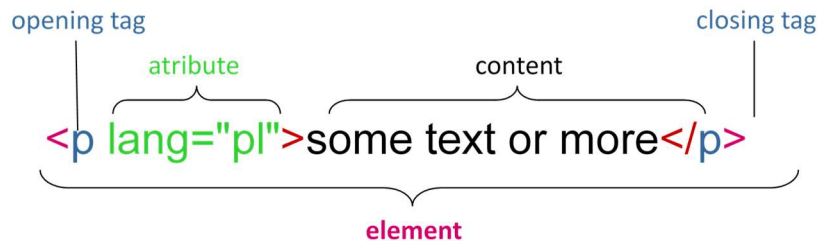
Wrocław 2024

1 Basic elements of creating an AJAX based web application (web browser client)

The chapter briefly presents the basic programming elements important for AJAX technique based application client.

1.1 Elements of HTML essential for the implementation of AJAX mechanisms

HTML (HyperText Markup Language) is a language for describing hypermedia content / web documents. The basic components of HTML documents are **HTML elements** defined by tags:



Elements can be nested and tags can contain attributes that define various properties of the elements (e.g. colors, identifiers, etc.).

For the implementation of operations as a result of AJAX requests, the selection of HTML elements is important. The most common uses are:

- Selection based on element ID. One of the attributes can be a unique identifier **id** (in the example for a paragraph of the text (specified by the <p> tag)):

```
<p id="my-id"> here is some text </p>
```

- Selection based on the class to which many elements can be assigned. The **class** attribute can be assigned to multiple elements and multiple classes can be defined (in the example for a text paragraph and 2nd-order heading):

```
<p id="my-id" class="my-first-class"> ... </p>
```

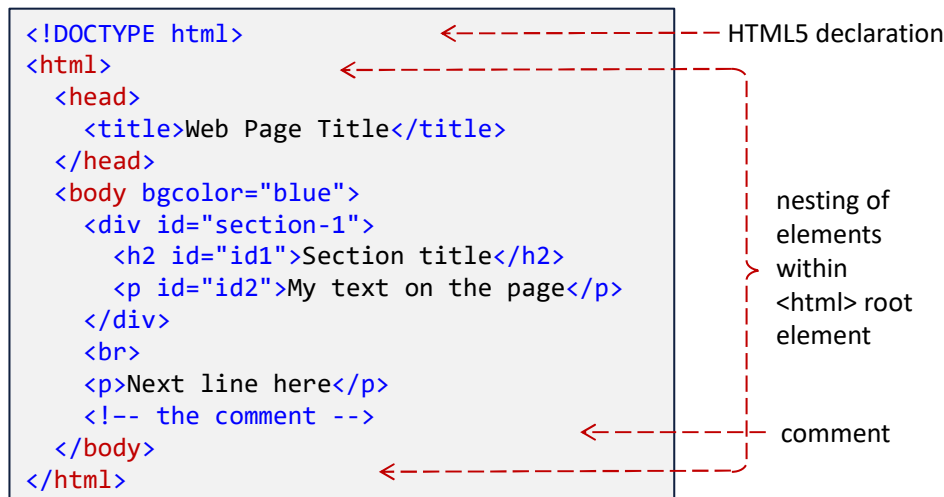
```
<h2 id="other-id" class="my-first-class my-second-class"> ... </h2>
```

- Selection by element type (tag name).

Some of the most basic tags for defining the structure and content of HTML docs are:

Tag	Description	Tag	Description
<html>	Defines an HTML document		an unordered list
<head>	The header of the document		an ordered list
<body>	Defines the document's body		a list item
<p>	Defines a paragraph	<table>	a table
<hx>	HTML heading of level x=1...6	<tr>	a row in a table
<a>	Defines a hyperlink	<td>	a cell in a table
	Defines an image	<input>	an input element to enter data
<button>	Defines a clickable button	<div>	a section in a document

The `<html>`, `<head>`, and `<body>` tags define the required basic document structure:.



1.2 DOM and JavaScript

Web browsers display the content on the screen (browser window) based on the DOM (Document Object Model) tree (including HTML DOM). The DOM tree is a tree of objects representing the elements and structure of the HTML document to display, built by the browser based from HTML code and CSS styles. DOM tree objects are accessed through the DOM API. **The ability to modify the DOM tree allows you to directly impact what is displayed in the browser (i.e. on the client side).**

The API for JavaScript provides objects and methods that allow you to operate on the DOM tree. Many of the actions performed are initiated using the **document** object. The objects and methods available include, for example:

- searching for HTML elements - for example:

Method	Description
document.getElementById(id)	Find an element by element id
document.getElementsByTagName(name)	Find elements by tag name
document.getElementsByClassName(name)	Find elements by class name

- adding/removing elements - e.g.:

Method	Description
document.createElement(element)	Create an HTML element
document.removeChild(element)	Remove an HTML element
document.appendChild(element)	Add an HTML element
document.replaceChild(new, old)	Replace an HTML element
document.write(text)	Write into the HTML output stream

- change of HTML elements (attributes) and CSS styles - for example:

Property	Description
<i>element.innerHTML</i> = <i>new html content</i>	Change the inner HTML of an element
<i>element.attribute</i> = <i>new value</i>	Change the attribute value of an HTML element
<i>element.style.property</i> = <i>new style</i>	Change the style of an HTML element
Method	Description
<i>element.setAttribute(attribute, value)</i>	Change the attribute value of an HTML element

where *element* is a selected specific html element (e.g. a paragraph of text), *attribute* is a specific attribute of the element, *style.property* is a specific style of a given element.

Example:

For HTML document::

```
<html>
  <body bgcolor="blue">
    <p id="id1">First text</p>
    <p id="id2" class="c2">Second text</p>
    <p id="id3">Third text</p>
  </body>
</html>
```

The Javascript code:

```
<script type="text/javascript">
  document.getElementById("id1").innerHTML = "New Text";
  document.getElementsByClassName("c2").innerHTML = "New Text 2";
  document.getElementsByTagName("p").innerHTML = "New Text 3";
  document.getElementById("id1").align = "center";
  document.getElementById("id2").style.color = "red";
  document.getElementById("id3").setAttribute("align","center");
</script>
```

The given JavaScript code (e.g. defined in a function) is executed as a result of events, including interface **input events** (user events) - e.g. the **onClick** event - clicking a given element. JavaScript allows you to define event handling statically and dynamically.

The assignment of code execution to a given event is defined statically by specifying an attribute of the element with **name of event** and assigning it a code to be executed (e.g. assigning the name of a function defined in the script):

```
<html>
  <head>
    <script type="text/javascript">
      function myFunction() { [...here script code...] }
    </script>
  </head>
```

```
<body>
  <button id="id1" onClick = "myFunction()">...</button>
  ...
</body>
</html>
```

1.3 JSON methods

For handling AJAX requests, among many other JavaScript objects and functions worth to know are those that are useful for converting Json objects to text and vice versa:

JSON.parse(text) – returns Json object

JSON.stringify(object) – returns text representing Json object

2 Creating an AJAX based web application

In the exercise the implemented AJAX based application consists of:

- REST service with CRUD operations
- Web client built with AJAX technology.

2.1 HTTP-based REST service

Basic HTTP-based REST service characteristics

- The HTTP REST service should adhere among the others to the following rules:
 - the standard HTTP methods GET, POST, PUT (or PATCH), and DELETE should be used to perform CRUD operations.
 - standard HTTP status codes (response status) should be used in responses,
 - to support AJAX Web clients the CORS mechanism should be used.

In order to be able to execute all AJAX requests in the browser, the service must support the CORS mechanism (for *preflighted requests*), since most AJAX requests by definition are not allowed in the browser without the control of this mechanism.

2.1.1 CORS support in Spring

In Spring adding basic CORS support is very easy with annotation:

```
@CrossOrigin(origins="http://localhost:8080", methods="...", ...)
```

May be used for each method in controller separately or for the whole class (then applies to all methods in the class. If used without any specification (only **@CrossOrigin** keyword) it permits for all actions, from any location. The annotation permits to configure among the others:

- **origins** – domains that are eligible to send requests,
- **methods** – allowed HTTP methods (include also OPTIONS if specifying methods),
- **allowedHeaders** – allowed HTTP headers (i.e. not allowed by default),
- **maxAge** – maximum age (in seconds) for pre-flight response's (default is 1800 sec.).

2.1.2 CORS support in WCF Web App.

In WCF applications, support for preflight requests can be done by adding an additional **Global.asax** class to the project - a class that contains the code of methods called when handling requests to the service.

- Add a class with the option:

Add → New Item → Web\General\Global application class

- Add the code for the **Application_BeginRequest** method, in which we allow requests for appropriate HTTP methods and headers from each location (e.g. for a period of 2 hours):

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e) { }
    protected void Session_Start(object sender, EventArgs e) { }
    protected void Application_BeginRequest(object sender, EventArgs e) {
        HttpContext.Current.Response.AddHeader("Access-Control-Allow-Origin", "*");
        if (HttpContext.Current.Request.HttpMethod == "OPTIONS")
        {
            HttpContext.Current.Response
                .AddHeader("Access-Control-Allow-Methods", "POST, PUT, DELETE");
            HttpContext.Current.Response
                .AddHeader("Access-Control-Allow-Headers",
                    "Content-Type,
                    Accept");
            HttpContext.Current.Response.AddHeader("Access-Control-Max-Age", "7200");
            HttpContext.Current.Response.End();
        }
    }
    protected void Application_AuthenticateRequest(object sender, EventArgs e) { }
    protected void Application_Error(object sender, EventArgs e) { }
    protected void Session_End(object sender, EventArgs e) { }
    protected void Application_End(object sender, EventArgs e) { }
}
```

2.2 AJAX Web Client

The Web client of HTTP-based REST service in the form of Web page with appropriate event handling, can be implemented with pure **JavaScript** code, or any ready to use library like **jQuery**.

2.2.1 Using pure JavaScript

The JavaScript code uses **XMLHttpRequest** (XHR for short) object to define/prepare and send Ajax request, and define handling of asynchronous response.

- The basic **XHR methods** to perform basic operations are:

`open(method, url)` – to create a query/request (asynchronous by default), or:

`open(method, url, async)`

`open(method, url, async, username, password)`

where: *method* – is one of HTTP method, *url* – is URL to send the request to, *async* – the Boolean parameter indicating whether or not to perform the operation asynchronously.

`send()` – sends the request to the server,

`send(body)` – method accepts an optional parameter which lets to specify the request's body

where: *body* – can be a *Blob*, *BufferSource*, *FormData*, *URLSearchParams*, or string object.

`abort()` – aborts the request if it has already been sent.

- The basic **object properties** used to support operations are:

- **readyState** – the status if request execution

The following states are possible:

Value	State	Description
0	UNSENT	Client has been created. <code>open()</code> not called yet.
1	OPENED	<code>open()</code> has been called.
2	HEADERS_RECEIVED	<code>send()</code> has been called, and headers and status are available.
3	LOADING	Downloading; <code>responseText</code> holds partial data.
4	DONE	The operation is complete.

- **responseType** – specifies the type of the response.
- **response**
responseText
responseXML – the response of given type: (1) an *ArrayBuffer*, a *Blob*, a *Document* (XML, HTML,...), a JavaScript object, or a string, depending on the value of `responseType`, or (2) *plain text*, or (3) a *Document* containing the HTML or XML
For `responseXML` usually, the response is parsed as "text/xml". If the `responseType` is set to "document" and the request was made asynchronously, instead the response is parsed as "text/html".
- **status** – HTTP response code (e.g. 200, 404, etc.)
- **statusText** – contains status as a string (e.g. "OK" or "Not Found")

- The basic **object events** used to support operations are:
 - **readystatechange** – fired whenever the readyState property changes. Also available via the onreadystatechange event handler property.
 - **load** – fired when an XHR transaction completes successfully. Also available via the onload event handler property,
- The basic **object events handlers** used to support operations are:
 - **onreadystatechange** – reference to the function invoked by the readyState change.
 - **onload** – reference to the function invoked when request is completed.

Code examples

- 1) A simple example (a pattern) to implement the GET request (enclosed in **doRequest** function) and reaction to completion of the request with execution of **doJob** function:

```
function doRequest()
{
  var xhr = new XMLHttpRequest();
  xhr.onreadystatechange = doJob; // callback
  xhr.open("GET", url, true);
  xhr.send();
}

function doJob()
{
  if (xhr.readyState==4) {
    if (xhr.status==200) {
      [...] // process received data and/or do anything you want
    }
    else { alert("Data query/response error"); }
  }
}
```

- 2) A snippet with using **onload** property:

```
function doRequest()
{
  var xhr = new XMLHttpRequest();
  xhr.onload = function doJob() { // callback
    [...] //process received data and/or do anything you want
  };
  xhr.open("GET", url, true);
  xhr.send();
}
```


3) An example with POST method:

```
var xhr = new XMLHttpRequest();
xhr.open("POST", url, true);

//Send the proper header information along with the request
xhr.setRequestHeader("Content-Type", "application/json");

xhr.onreadystatechange = function() { // when the state changes.
    if (this.readyState === XMLHttpRequest.DONE &&
        this.status === 200) {
        [...] //process received data and/or do anything you want.
    }
}

xhr.send("{foo:bar , lorem:ipsum}");
```

2.2.2 Using Fetch API

...soon to complete – sorry ☹

2.2.3 Using jQuery

The jQuery library supports AJAX requests with **\$.ajax()** method.

2.2.3.1 \$.ajax() method

The scheme of using **\$.ajax()** function is the following:

```
$.ajax({
    url          : "http://host.domain.org/service",
    method       : "POST",
    dataType     : "json",
    contentType  : "application/json",
    data         : {
        name      : "Mariusz",
        country   : "Polska"
    }
})

.done( function(data) {
    [...] //process received data
})

.fail( function(xhr, status, error) {
    [...] //handle the error
})

.always( function() {
    [...] //perform some processing
});
```

Where properties (method parameters) are:

`url` – targeted address (where to send request),
`method` – connection type/HTTP method (GET by default),
`dataType` – the MIME type of data expected in response,
`contentType` – the type of the sent data,
`data` – data to be sent (e.g. JSON object),

The defined methods called in the effect of the request are:

- .done** – called when the request is successfully completed,
- .fail** – called after an error,
- .always** – code called in both cases.

Unused method can be omitted.

The other useful methods (some selection) are:

- `$.load()` – loads data from a server and puts the data into the selected element
- `$.get()` – loads data from a server using an AJAX HTTP GET request
- `$.getJSON()` – loads JSON-encoded data from a server with HTTP GET request
- `$.getScript()` – loads (and executes) a JavaScript from a server using an AJAX HTTP GET request
- `$.post()` – loads data from a server using an AJAX HTTP POST request
- ...and a few more.

2.2.3.2 Examples of implementing Ajax Web client using jQuery

- Let's assume the REST service that:
 - performs CRUD operations on the list of the following item data:
 - id** – the item identifier,
 - name** – the item name,
 - intValue** – the item integer value .
 - the base address of resources is **`http://example.com/items`**, (for local tests e.g.: **`http://localhost:8080/items`**).
 - It uses JSON format for transferred data.

In JavaScript code variables for this data (string and JSON objects) can be:

```
<script type="text/javascript">
  var myUrl = "http://example.com/items"
  var Item = {
    "id":0,
    "name":"",
    "intValue ":0
  }
  ...
</script>
```

- The HTML code of the web page that support operation with the service and presents results of operation contains:

- Input type elements that allow to enter some data:

```
<input id="id_val" type="text" />
<input id="name" type="text" />
<input id="intvalue" type="text" />
```

For each field the unique identifier attribute is set. These elements will be used to enter data to be sent or data read from server.

- The buttons that after clicking call a Javascript function that send ajax request:

```
<button type="button" id="read" onclick="getItem()">Read</button>
<button type="button" id="add" onclick="addItem()">Add</button>
```

- The table with identifier **itemtable** where the read data list is presented:

```
<table id="itemtable"> </table>
```

Note: at the beginning the table is empty.

An example view can be the following:

Above is table with inputs, below the table to present data initially empty).

- To activate/run Ajax request with input data the input fields and the buttons are to be included in `<form>` tag e.g. as follows (also positioned in table):

```
<h2 style="margin-top:10px;margin-bottom:25px;">Operacje</h2>
```

```
<form>
  <table id="itemoperation" cellpadding="1">
    <tr>
      <td>ID</td>
      ...
    </tr>
    <tr>
      <td><input id="id_val" type="text" /></td>
      ...
    </td>
```

```

        <button type="button" id="read"
            onclick="getJSONItem()">Wczytaj</button>
        ...
    </td>
</tr>
</table>
</form>

```

- The code to read and present the item data in fields can be the following:

```

function getItem() {
    var id = $("#id_val").val();

    $.ajax({
        type: "GET",
        url: myUrl + "/" + id,
        contentType: "application/json; charset=utf-8",
        dataType: "json",
    })

    .done(function(msg) {
        document.getElementById("name").value=msg.name;
        document.getElementById("intvalue").value=msg.intValue ;
    })

    .fail(function(errMsg) {
        alert(JSON.stringify(errMsg));
    });
}

```

Here:

- the item **id** (the value) is read from input element of identifier **id_val**,
- next Ajax GET request is sent to **http://example.com/items/id** (where **id** is entered id in input element),
- on request completion the received data in the form of JSON (in **msg** parameter) is put into the rest of input elements (of identifier **name** and identifier **intvalue**).

Similarly the data may be inserted in other HTML element – e.g. text paragraph `<p id=pid1></p>` – with the instruction:

```
document.getElementById("pid1").innerHTML = msg.name;
```

- The example code to add an item data with Ajax request can be the following:

```

function addItem() {
    var item = Item;

    item.id = $("#id_val").val();
    item.name = $("#name").val();
    item.intVal = $("#intvalue").val();

    var id = $("#id_val").val();

```

```

$.ajax( {
  type: "POST",
  url: myUrl + "/", // "/" + id,
  data: JSON.stringify(item),
  contentType: "application/json; charset=utf-8",
  dataType: "json",
})
.done(function(msg) {
  [...] // here may be some action to show that data are stored
        // e.g.: alert('the item is stored');
        // or reloading data list, etc.
})
.fail(function(errMsg) {
  alert(JSON.stringify(errMsg));
});
}

```

Here:

- the data (values) are read from HTML `<input>` elements and saved in JSON *item* variable,
 - the Ajax POST request is sent ...
 - ... with data stringified to text.
- The example code to read whole data and present it with use of iterating JSON data can be the following:

```

function getAllItems()
{
  [...]
  url: myUrl,
  [...]
  .done(function(msg) {
    showItems(msg)
  });
}

```

- The function presents read data in `<table>` element (initially empty) in browser window) will:
 - remove previous content of table (tbody) – in case something was put previously,
 - iterate received data – all elements of received JSON data containing item list with use of **\$.each** method,
 - add table rows with data taken from JSON objects into table `<tbody>` element.

```

function showItems(msg)
{
    $("#itemtable tbody").remove(); //remove previous data
    $.each(msg, function (id, item) { // iterate by all JSON elements
        // (items)

        // First check if a <tbody> tag exists and add one if not
        if ($("#itemtable tbody").length == 0) {
            $("#itemtable").append("<tbody></tbody>");
        }
        // Append table row to <table> (in <tbody> part)
        $("#itemtable tbody").append("<tr>" +
            "<td>" + item.id + "</td>" +
            "<td>" + item.name + "</td>" +
            "<td>" + item.intVal + "</td>" +
            "</td>" +
            "</tr>");
    });
}

```

The rest of CRUD operations and presentation of results in browser window is implemented similarly.

The view with results of operations can be similar to the following:

The screenshot shows a web application with a light green background. At the top, there's a header "Operacje". Below it, there's a form with three input fields: "ID" (containing "2"), "Name" (containing "Second item"), and "Value" (containing "20"). To the right of these fields are four buttons: "Wczytaj", "Aktualizuj", "Dodaj", and "Usun". Below these buttons is a button labeled "Wyswietl liste".

Below the form is a section titled "Items" containing a table with three columns: "ID", "Name", and "Value". The table has three rows of data:

ID	Name	Value
1	First item	10
2	Second item	20
3	Third item	30

Red arrows and dashed boxes indicate the state of the application:

- A dashed box labeled "po wczytaniu" (after loading) has two arrows pointing to the "Name" and "Value" input fields.
- A dashed box labeled "po wyświetleniu" (after displaying) has an arrow pointing to the "Second item" row in the table.

3 Exercise

The detailed and final requirements for are set by the teacher.

Implement the very basic web app (the web page) that performs very basic interactions with AJAX technique as getting information (GET request) and sending information (POST request), updating element (PUT), and delete element (DELETE).

jQuery is suggested to use (but also fetch method may be used).

You may use to interact with:

- service provided/run by the teacher.

or

- your own REST service

in such case:

- at the beginning simplify responses removing links in returned data to easier access given data
- remove/comment linkTo calls – return pure data without links,
- also for this purpose, for list of items (from previous exercise) replace `CollectionModel<EntityModel<Item>> getAllItems()` with `List<Item> getAllItems()`
- etc.

4 Exercise – Part II

The detailed and final requirements for Part II are set by the teacher.

A. Develop or prepare to develop a program according to the teacher's instructions.