# Exercise 7

# AJAX Application
(with REST service)

*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 REST service with AJAX Client.

## 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

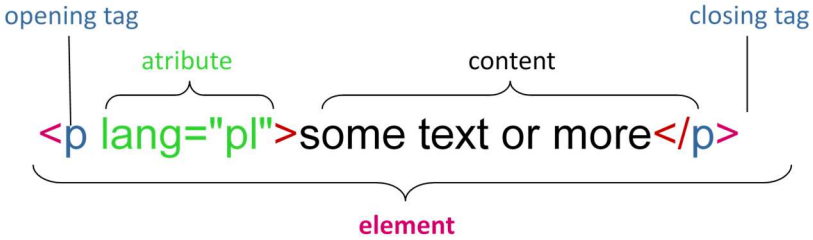
Wrocław University of Science and Technology
Wrocław 2022

# 1 Basic elements of creating a web application (for the browser)

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

## 1.1 Elements of HTML language 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 | **<ul>** | Defines an unordered list |
| **<head>** | Defines information about the document | **<ol>** | Defines an ordered list |
| **<body>** | Defines the document's body | **<li>** | Defines a list item |
| **<p>** | Defines a paragraph | **<table>** | Defines a table |
| **<h1> to <h6>** | Defines HTML headings | **<tr>** | Defines a row in a table |
| **<a>** | Defines a hyperlink | **<td>** | Defines a cell in a table |

| | | | |
|---|---|---|---|
| **<img>** | Defines an image | **<input>** | Defines an input element to enter data |
| **<button>** | Defines a clickable button | ** ** | Defines a section in a document |

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

```
<!DOCTYPE html>
<html>
  <head>
    <title>Web Page Title</title>
  </head>
  <body bgcolor="blue">
    <div id="section-1">
      <h2 id="id1">Section title</h2>
      <p id="id2">My text on the page</p>
    </div>
    <br>
    <p>Next line here</p>
    <!-- the comment -->
  </body>
</html>
```

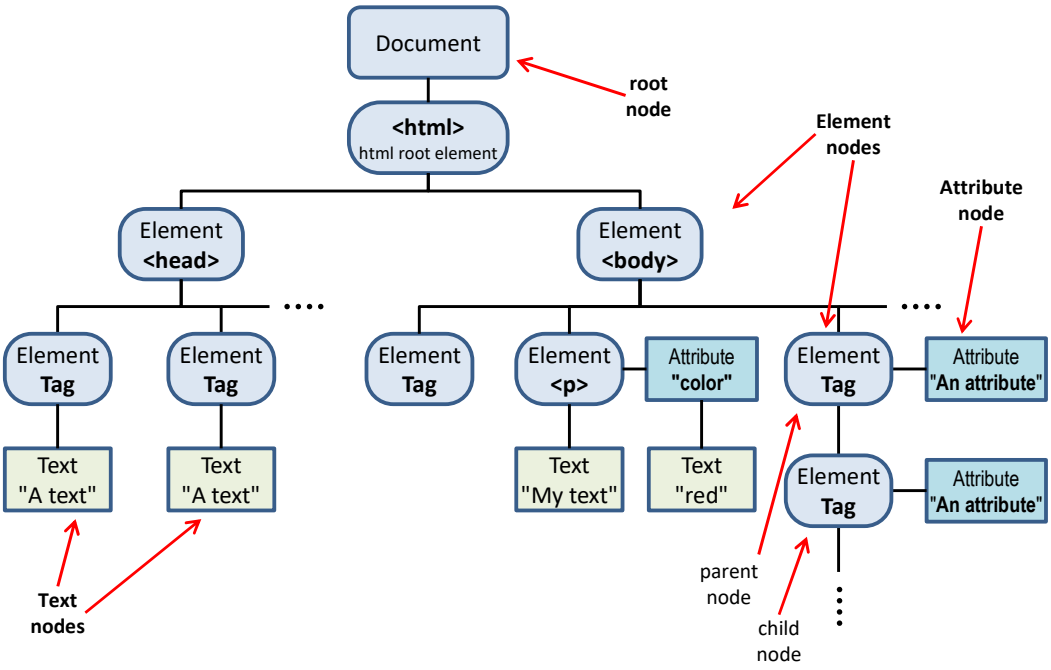
HTML5 declaration

nesting of elements within <html> root element

comment

## 1.2 DOM and JavaScript

Web browsers display the content on the screen (browser window) based on the DOM tree (including HTML DOM). The DOM (Document Object Model) 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 |
|---|---|
| ***element*.innerHTML = *new html content*** | Change the inner HTML of an element |
| ***element*.*attribute* = *new value*** | Change the attribute value of an HTML element |
| ***element*.*style*.*property* = *new style*** | Change the style of an HTML element |
| Method | Description |
| ***element*.setAttribute(*attribute*, value)** | 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.

**Examples**:

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.getElementByClass("c2").innerHTML = "New Text 2";
  document.getElementByTagName("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() {
      ...
    }
  </script>
 </head>
 <body>
  <button id="id1" onClick = "myFunction()">...</button>
  ...
 </body>
</html>
```

### 1.3 JSON methods

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

```
JSON.parse(text)
```

```
JSON.stringify(object)
```

## 2 Creating an AJAX based web application

In the exercise will be implemented an AJAX based application i.e.:

- 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:
  - **URIs/URLs** are used to locate the resource,
  - the **application/json** or/and **application/xml** (or **text/xml**) data type (or similar open standard) should be used for transferred data,
  - 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.
- To implement CRUD operations the following scheme (a combination of URLs and HTTP methods) should be used:

  **GET** method to **R**etrieve resource (data),

  **POST** method to **C**reate resource (add data),

  **PUT** (or **PATCH**) method to **U**pdate resource (modify data),

  **DELETE** method to **D**elete resource (remove data).

  Usually PUT replaces whole resource and PATCH only a part.
- The scheme for operation on items in the collection (set, list, etc.) of items is:

| URL | Method | Operation |
|---|---|---|
| http://example.com/items | GET | Get list (collection) of items |
| http://example.com/items/{id} | GET | Get item data of item with id={id} |
| http://example.com/items | POST | Adding a new item in collection of items |
| http://example.com/items/{id} | PUT | Updating item data of item with id={id} (or adding an item – not a standard) |
| http://example.com/items | DELETE | Removal of the whole collection of items |
| http://example.com/items/{id} | DELETE | Removal of item with id={id} |

Where:

- **http://example.com/items** is base URL to the service (resource data) – the name items can be different of course,
- **{id}** is unique identifier (any unique string) for the resource.

The other (not CRUD) operations should be specified in the URL **after** the resource it apply.

The scheme takes one more level of path if the resource set contains subsets of items.

- The result status of operations should follow the following rules:

| HTTP method | CRUD | Entire Collection ( e.g. /items ) | Specific Item ( e.g. /items/{id} ) |
|---|---|---|---|
| **POST** | Create | 201 (Created), 'Location' header with link to /items/{id} containing new ID. | 404 (Not Found), 409 (Conflict) if resource already exists.. |
| **GET** | Read | 200 (OK), list of items. | 200 (OK), single item. 404 (Not Found), if ID not found or invalid. |
| **PUT** | Update / Replace | 405 (Method Not Allowed), unless you want to update / replace every resource in the entire collection. | 200 (OK) … or 204 (No Content). 404 (Not Found), if ID not found or invalid. |
| **PATCH** | Update / Modify | 405 (Method Not Allowed), unless you want to modify the collection itself. | 200 (OK) … or 204 (No Content). 404 (Not Found), if ID not found or invalid. |
| **DELETE** | Delete | 405 (Method Not Allowed), unless you want to delete the whole collection. | 200 (OK). 404 (Not Found), if ID not found or invalid. |

### 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, any ready to use library like **jQuery**, or a **framework** build on basic mechanisms.

#### 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. open() not called yet. |
| 1 | OPENED | open() has been called. |
| 2 | HEADERS_RECEIVED | send() has been called, and headers and status are available. |
| 3 | LOADING | Downloading; responseText 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 hanlers** 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 querry/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

*Not described yet – 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` – targed 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 returned 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 using a 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 book data:
    **Id** – the book identifier,
    **Title** – the book title,
    **Price** – the book price.
  - the base address of resources is **http://example.com/books**, (for local tests e.g.: **http://localhost:8080/books**).
  - It uses JSON format for transferred data.

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

```
<script type="text/javascript">
  var myUrl = "http://example.com/books"
  var Book = {
    "Id":0,
    "Title":"",
    "Price":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="bid" type="test" />
<input id="btitle" type="text" />
<input id="bprice" 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="getBook()">Read</button>
<button type="button" id="add" onclick="addBook()">Add</button>
```

- The table where the read data list is presented:

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

At the beginning the table is empty.

- The code to read and present the book data with Ajax request can be the following:

```
function getBook() {
  var id = $("#bid").val();

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

  .done(function(msg) {
    document.getElementById("btitle").value=msg.Title;
    document.getElementById("bprice").value=msg.Price;

  })

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

Here:

– the book **id** is read from input element of id=bid,
– next Ajax GET request is sent to **http://example.com/books/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 id=btitle and id=bprice).

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.Title;
```

- The example code to add a book data with Ajax request can be the following:

```
function addBook() {
  var book = Book;

  book.Id = $("#bid").val();
  book.Title = $("#title").val();
  book.Price = $("#price").val();
```

```
  var id = $("#bid").val();
  $.ajax( {
    type: "POST",
    url: myUrl + "/", // /"+id,
    data: JSON.stringify(book),
    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 book is stored');
          // or reloading data list, etc.
  })
  .fail(function(errMsg) {
    alert(JSON.stringify(errMsg));
  });
}
```

Here:

- the data are read from HTML <input> elements and saved in JSON *book* 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:
  - The Javascript function sending ajax request is almost the same as for reading single data item except for URL and the call of function displaying data on completion of the request:

```
function getAllBooks()
{
  [...]
    url: myUrl,
  [...]
  .done(function(msg) {
    showBooks(msg)
  });
}
```

  - The function presenting read data in <table> element (in browser window) will:
    - remove previous content of table (tbody),
    - iterate received data – all elements of received JSON data containing book list with use of **$.each** method,
    - add table rows with data taken from JSON objects into table <tbody> element.

```
function showBooks(msg)
{
  $("#booktable tbody").remove();  //remove previous data
  $.each(msg, function (id, book) {  // iterate by all JSON elements
                                     // (books)
    // First check if a <tbody> tag exists and add one if not
    if ($("#booktable tbody").length == 0) {
      $("#booktable").append("<tbody></tbody>");
    }
    // Append table row to <table> (in <tbody> part)
    $("#booktable tbody").append("<tr>" +
                                 "<td>" + book.Id + "</td>" +
                                 "<td>" + book.Title + "</td>" +
                                 "<td>" + book.Price + "</td>" +
                                 "</td>" +
                                 "</tr>");
  });
}
```

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

## 3 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.