AJAX

CS472 Web Programming

Maharishi University of Management

Department of Computer Science

Maharishi University of Management - Fairfield, Iowa © 2016

Main Point Preview

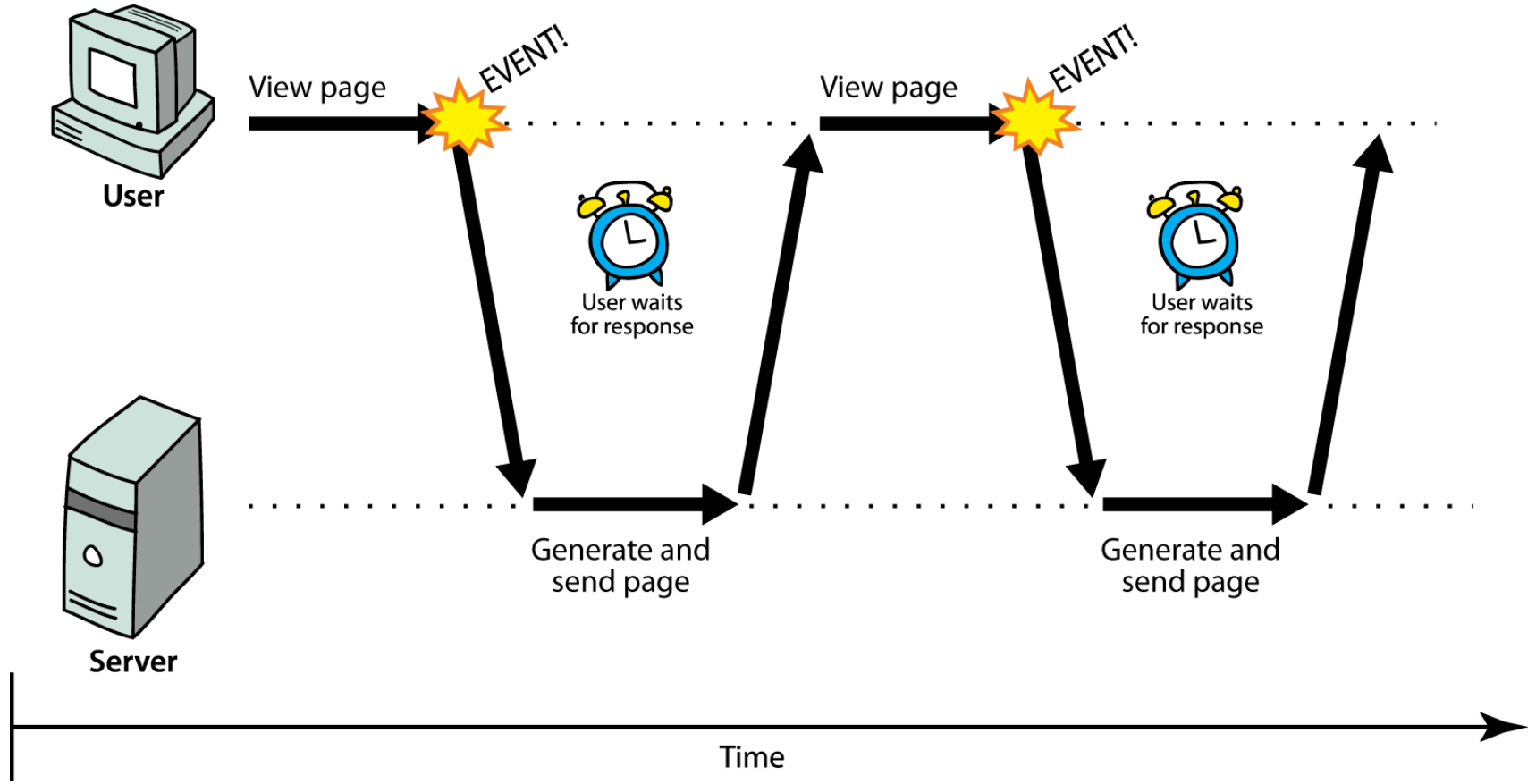
XMLHttpRequest object

The key component that a browser provides to enable Ajax is the XMLHttpRequest object, which is supported by all modern browsers. This object opens a connection with a server, sends a message, waits for the response, and then activates a given callback method.

Science of Consciousness: The TM Technique is supported by any human nervous system. It allows us to connect with the source of thought, experience restful alertness, and then return to activity with that influence of calm alertness.

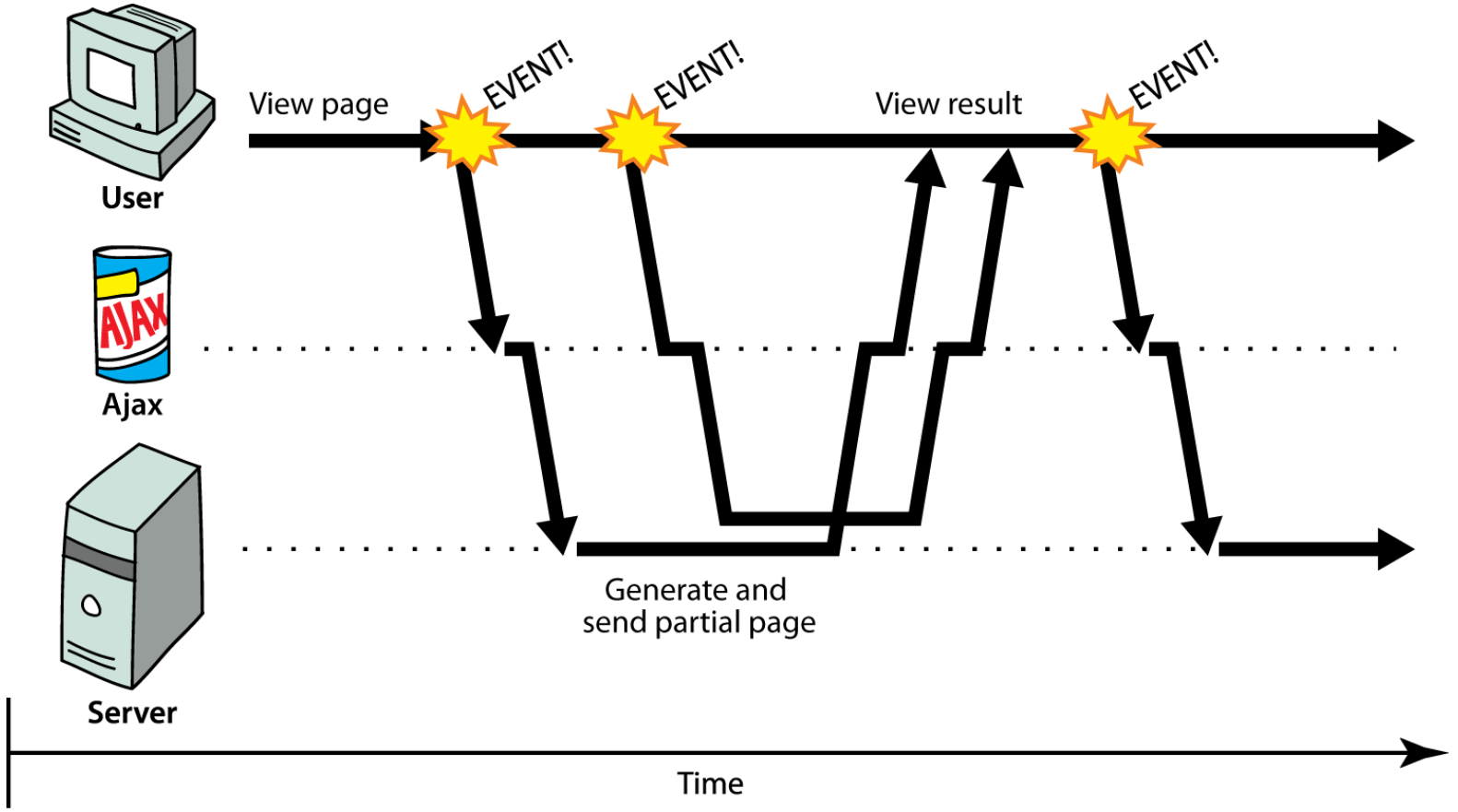
Synchronous web communication

Synchronous: user must wait while new pages load (click, wait, refresh)



Asynchronous web communication

Asynchronous: user can keep interacting with page while data loads



Web applications and Ajax

- **Web Application**: a dynamic web site that mimics the feel of a desktop app
  - Presents a continuous user experience rather than disjoint pages
  - Examples: Gmail, Google Maps, Google Docs and Spreadsheets
- **Ajax**: Asynchronous JavaScript and XML
  - Not a programming language; a particular way of using JavaScript
  - Downloads data from a server in the background
  - Allows dynamically updating a page without making the user wait
  - Avoids the "click-wait-refresh" pattern
  - Example: Google Suggest

XMLHttpRequest *(and why we won't use it)*

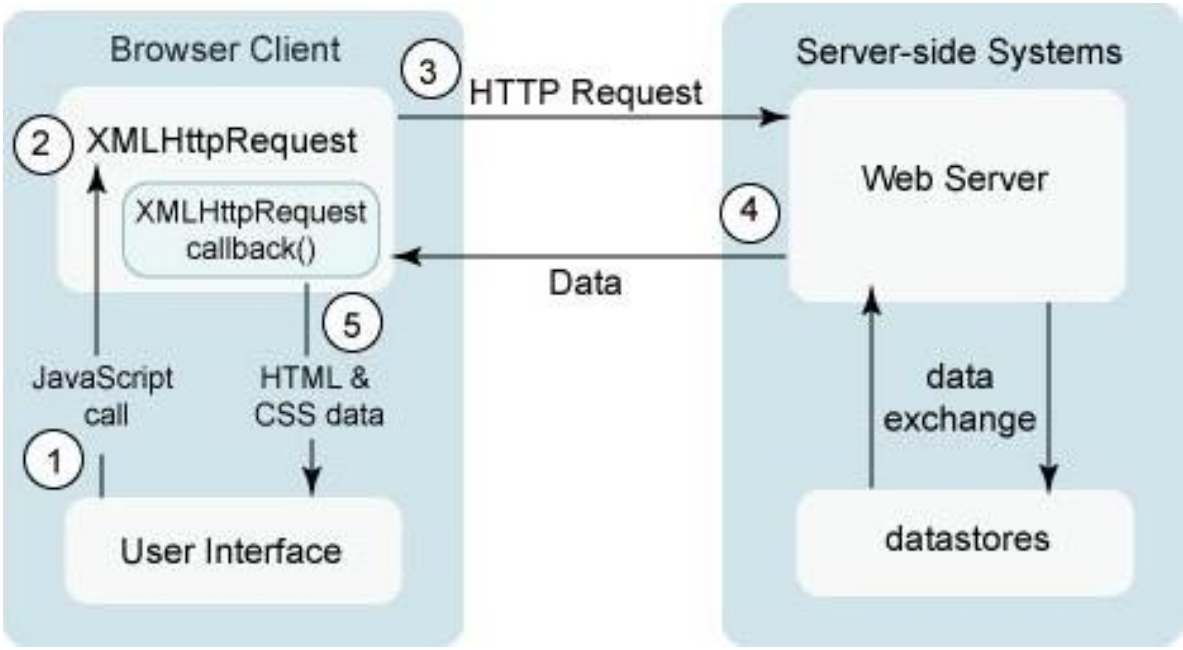
- JavaScript includes an **`XMLHttpRequest`** object that can fetch files from a web server
- Supported in IE5+, Safari, Firefox, Opera, Chrome, etc. (with minor compatibilities)
- It can do this asynchronously (in the background, transparent to user)
- The contents of the fetched file can be put into current web page using the DOM
- Sounds great!...

XMLHttpRequest *(and why we won't use it)*

- It is clunky to use, and has various browser incompatibilities
- jQuery provides a better wrapper for Ajax, so we will use that instead

A typical Ajax request

1. User clicks, invoking an event handler
2. Handler's code creates an XMLHttpRequest object
3. XMLHttpRequest object requests response from server
4. Server retrieves appropriate data, sends it back
5. XMLHttpRequest fires an event when data arrives (callback, you can attach a handler function to this event)
6. Your callback event handler processes the data and displays it



Main Point

XMLHttpRequest object

The key component that a browser provides to enable Ajax is the XMLHttpRequest object, which is supported by all modern browsers. This object opens a connection with a server, sends a message, waits for the response, and then activates a given callback method.

Science of Consciousness: The TM Technique is supported by any human nervous system. It allows us to connect with the source of thought, experience restful alertness, and then return to activity with that influence of calm alertness.

Main Point Preview

Ajax requests

Ajax requests require a url for the target on the server, a designation of whether to send a Get or Post request, one or more callback functions to be called with the result, and optionally a set of request parameters. jQuery provides convenient wrapper methods: $.ajax(), $.get(), and $.post.

Science of Consciousness: Ajax requests are a highly efficient means to obtain information from the server that is the source of the application. The TM Technique is a highly efficient means to experience pure consciousness from the source of thought.

jQuery's ajax method

- Call the $.ajax() method
- Argument accepts an object literal full of options that dictate the behavior of the AJAX request:
  - The url to fetch, as a String,
  - The type of the request, GET or POST.. etc
- Hides icky details of the raw XMLHttpRequest; works well in all browsers

```
$.ajax({
  "url": "http://foo.com",
  "option" : "value",
  "option" : "value",
  ...
  "option" : "value"
});
```

$.ajax() options

option	description
url	The URL to make a request from
type	whether to use POST or GET
data	an object literal filled with query parameters and their values
dataType	The type of data you are expecting to receive, one of: "text", "html", "json", "xml"
timeout	an amount of time in seconds to wait for the server before giving up
success	*event:* called when the request finishes successfully
error	*event:* called when the request fails
complete	*event:* called when the request finishes successfully or erroneously

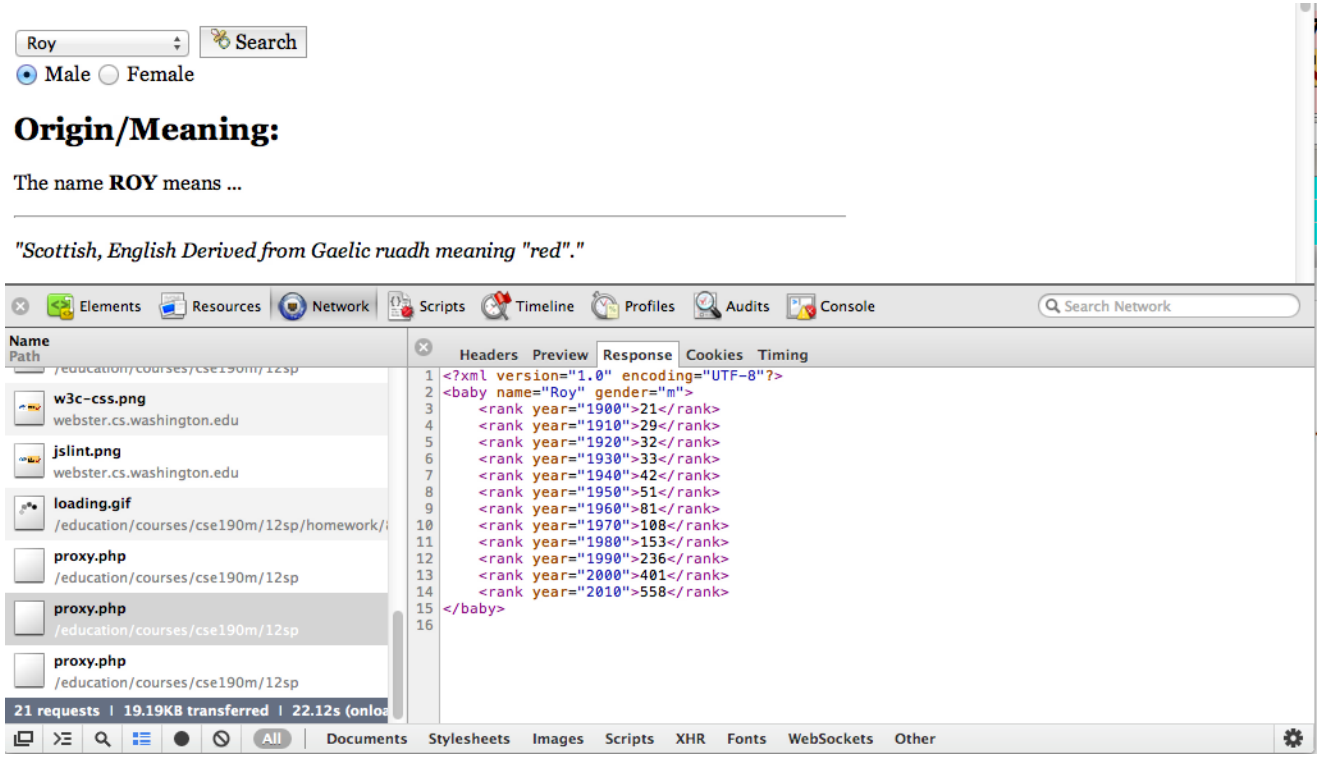
jQuery AJAX example

```
$.ajax({
        "url": "foo/bar/mydata.txt",
        "type": "GET",
        "success": myAjaxSuccessFunction,
        "error": ajaxFailure
});

function myAjaxSuccessFunction(data) {
        // do something with the data
}
function ajaxFailure(xhr, status, exception) {
         console.log(xhr, status, exception);
}
```

Debugging AJAX code

- Chrome DevTool's **Network** tab shows each request, parameters, response, errors
- expand a request by clicking on it and look at **Response** tab to see Ajax result
- check the **Console** tab for any errors that are thrown by requests



Examples – Check in DevTools!

Adding AJAX code to load in a homework output file into the textarea (output page)

Another example: The quotes page

Better jQuery AJAX

Rather than specify all of the options in an object literal...

```
$.ajax({ "url": "http://foo.com",
         "type": "GET",
         "success": functionName,
         "error": ajaxFailure });
```

one can pass the URL as the first parameter and the rest as an object literal. Why? It makes it even easier to see what this AJAX request is doing.

```
$.ajax("http://foo.com", { "type": "GET",
                           "success": functionName,
                           "error": ajaxFailure });
```

Even Better jQuery AJAX

Using these event handler function calls done() and fail() instead

```
$.ajax("http://foo.com", { "type": "GET" })
       .done(functionName)
       .fail(ajaxFailure);
```

Passing query parameters to a request

```
$.ajax("lookup_account.php", { "type": "GET",
                               "data": { "name": "Asaad Saad",
                                         "height": 180,
                                         "password": "abcdef" },
                             }) .done(function)
                                .fail(function);
```

- Don't concatenate the parameters onto the URL yourself with "?" + ...
  - won't properly URL-encode the parameters
  - won't work for POST requests
- Query parameters are passed as an object literal with the data property

  (the above is equivalent to: "name=Asaad+Saad&height=180&password=abcdef")

Creating a POST request

`type` should be changed to "`POST`" (`GET` is default)

```
$.ajax("url", {
        "type": "POST",
        "data": {
                "name": value,
                "name": value,
                    ...,
                "name": value
        },
}) .done(function)
   .fail(function);
```

$.get() and $.post() shortcuts

Often you don't need the flexibility of $.ajax() function

- The options are hard to remember
- You don't usually need all of the options

These shortcut functions are preferred when additional options are not needed.

```
$.get(url, {data})
        .done(function)
        .fail(function);

$.post(url, {data})
        .done(function)
        .fail(function);
```

More about `$.get()` and `$.post()`

Why bother making the distinction if it all boils down to a call to $.ajax() under the hood

- It is less error prone
- It is easier to read

function	description
`$.ajax()`	A general function for making AJAX requests, other AJAX functions rely on this
`$.get()`	makes a GET request via AJAX
`$.post()`	makes a POST request via AJAX

AJAX user feedback

- Ajax calls are silent, no visual action to users.
- Users don't like unresponsiveness
- Often you show some sort of loader image or text while the request is made

User feedback with `always()`

The general technique is to `show()` some feedback when the AJAX request starts and `hide()` it again once it finishes.

- The `always()` function is an event that the AJAX request fires every time the request finishes, whether it was successful or not
- This might be some typical user feedback code

Async

```
$.get("url")
        .done(function)
        .fail(function)
        .always(function() { //fires when request finishes
                        $("#loader").hide(); });


$("#loader").show();
```

Global AJAX events

- User feedback and similar scenarios are so common that jQuery made global events for them
- Any element can register for these events just like a `click()` event method

event method	description
`.ajaxStart()`	fired when new AJAX activity begins
`.ajaxStop()`	fired when AJAX activity has halted
`.ajaxSend()`	fired each time an AJAX request is made
`.ajaxSuccess()`	fired each time an AJAX request finishes successfully
`.ajaxError()`	fired each time an AJAX request finishes with errors
`.ajaxComplete()`	fired each time an AJAX request finishes

User feedback example

```
$(function() {
        //loader will show whenever any ajax call is made on this page
        $("#loader").hide();
        $(document).ajaxStart(function() { $("#loader").show(); })
                   .ajaxStop(function() { $("#loader").hide(); });

        $("#mybutton").click(function() {
                              $.get("http://foo.com")
                                    .done(function)
                                    .fail(function);
        });
});
```

Main Point

Ajax requests

Ajax requests require a url for the target on the server, a designation of whether to send a Get or Post request, one or more callback functions to be called with the result, and optionally a set of request parameters. jQuery provides convenient wrapper methods: $.ajax(), $.get(), and $.post.

Science of Consciousness: Ajax requests are a highly efficient means to obtain information from the server that is the source of the application. The TM Technique is a highly efficient means to experience pure consciousness from the source of thought.

`XMLHttpRequest` security restrictions

- Ajax must be run on a web page stored on a web server
  - cannot be run from a web page stored on your hard drive
  - can open an html page from hard drive and run normal JavaScript
  - will not work if there is an Ajax call on that same page because browsers only allow Ajax calls to go to a 'domain' from which the page is served
  - Same Origin Policy

- Ajax can only fetch files from the same server that the page is on
  - http://www.foo.com/a/b/c.html can only fetch from www.foo.com
  - This is the ''

Main Points
Same origin policy

The same origin policy is a security constraint on browsers that restricts scripts to only contact a site with the same domain name, application protocol, and port. This means that browsers only allow Ajax calls to the same web server from which the page originated.

Science of Consciousness: If we are calm and alert then we are more secure from disruptions by external distractions or deceptions.

Main Point Preview
JSON

JSON has become more widely used for Ajax data representations than XML because JSON is easier to write and read and is almost identical to JavaScript object literal syntax.

Science of Consciousness: We always prefer to do less and accomplish more. Actions arising from deep levels of consciousness are more efficient and effective.

JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON): Data format that represents data as a set of JavaScript objects

- Natively supported by all modern browsers
- Replaced XML for data representation due to its simplicity and ease of use

http://www.json.org/

XML vs JSON

```xml
<?xml version="1.0" encoding="UTF-8"?>
<note private="true">
  <from>Alice Smith (alice@example.com)</from>
  <to>Robert Jones (roberto@example.com)</to>
  <to>Charles Dodd (cdodd@example.com)</to>
  <subject>Tomorrow's "Birthday Bash" event!</subject>
  <message language="english">
    Hey guys, don't forget to call me this weekend!
  </message>
</note>
```

```json
{
  "private": "true",
  "from": "Alice Smith (alice@example.com)",
  "to": [
    "Robert Jones (roberto@example.com)",
    "Charles Dodd (cdodd@example.com)"
  ],
  "subject": "Tomorrow's \"Birthday Bash\" event!",
  "message": {
    "language": "english",
    "text": "Hey guys, don't forget to call me this weekend!"
  }
}
```

JavaScript Object Notation (JSON)

JSON is a syntax for storing and exchanging data and an efficient alternative to XML

```
{"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]}
```

A name/value pair consists of a field name **(in double quotes)**, followed by a colon, followed by a value.

JSON values can be:

- A number (integer or floating point)
- A string (in double quotes)
- A Boolean (true or false)
- An array (in square brackets)
- An object (in curly braces)
- null

Browser JSON methods

- You can use Ajax to fetch data that is in JSON format
- Then call JSON.parse on it to convert it into an object
- Then interact with that object as you would with any other JavaScript object

method	description
`JSON.parse(`*`string`*`)`	converts the given string of JSON data into an equivalent JavaScript object and returns it
`JSON.stringify(`*`object`*`)`	converts the given object into a string of JSON data (the opposite ofJSON.parse)

JSON expressions exercise

Given the JSON data at right, what expressions would produce:

- The window's title?
- The image's third coordinate?
- The number of messages?
- The y-offset of the last message?

```
const jsonString = '{
  "window": {
    "title": "Sample Widget",
    "width": 500,
    "height": 500
  },
  "image": {
    "src": "images/logo.png",
    "coords": [250, 150, 350, 400],
    "alignment": "center"
  },
  "messages": [
    {"text": "Save", "offset": [10, 30]}
    {"text": "Help", "offset": [ 0, 50]},
    {"text": "Quit", "offset": [30, 10]},
  ],
  "debug": "true"
}';
const data = JSON.parse(jsonString);
```

JSON expressions exercise

Given the JSON data at right, what expressions would produce:

- The window's title?

```
var title = data.window.title;
```

- The image's third coordinate?

```
var coord = data.image.coords[2];
```

- The number of messages?

```
var len = data.messages.length;
```

- The y-offset of the last message?

```
var y = data.messages[len - 1].offset[1];
```

```
const jsonString = '{
  "window": {
    "title": "Sample Widget",
    "width": 500,
    "height": 500
  },
  "image": {
    "src": "images/logo.png",
    "coords": [250, 150, 350, 400],
    "alignment": "center"
  },
  "messages": [
    {"text": "Save", "offset": [10, 30]}
    {"text": "Help", "offset": [ 0, 50]},
    {"text": "Quit", "offset": [30, 10]},
  ],
  "debug": "true"
}';
const data = JSON.parse(jsonString);
```

JSON and AJAX

Your event handler is passed a JSON object as a parameter

```
$.get("url")
        .done(functionName); // we are expecting a JSON string/object

function functionName(jsonData) {
        // do stuff with the jsonData Object
        // if string: JSON.parse(jsonData); (not necessary if dataType: json)
}
```

Exercise: Parsing JSON

Suppose we have a service http://jsonplaceholder.typicode.com about blogs.

```
$.ajax('https://jsonplaceholder.typicode.com/todos/1')
  .done(response => {
    console.log(response);
    console.log(JSON.stringify(response));
    console.log("userid is: " + response.userId);
  });
```

Exercise: Parsing JSON

Suppose we have a service http://jsonplaceholder.typicode.com about blogs.

- Write a page that processes this JSON blog data.
- Display all users /users
- Display all posts from selected user /posts?userId=1
- Display all comments from selected post /comments?postId=1

- Create a SPA with input form to take userId from the browser
- Display user name and email and address and all posts belonging to an entered userId
- For every post, you need to show a button (show comments) once clicked you need to show all comments for the specific post.
  - Hint: hide postId in data- attribute in each post.
  - Consider using event delegation for the list of posts
- For efficiency remember to attach entire lists to DOM rather than each list item

Main Point
JSON

JSON has become more widely used for Ajax data representations than XML because JSON is easier to write and read and is almost identical to JavaScript object literal syntax.

Science of Consciousness: We always prefer to do less and accomplish more. Actions arising from deep levels of consciousness are more efficient and effective.

AJAX using Fetch API

- The Fetch API provides global `fetch()` method that provides an easy, logical way to fetch resources asynchronously across the network.
- Fetch provides a better alternative to XMLHttpRequest.

Basic fetch request

```
fetch('https://jsonplaceholder.typicode.com/todos/1’)
  .then(function(response) {
       return response.json();
  })
  .then(function(myJson) {
    console.log(stringified: ' + JSON.stringify(myJson));
  });
```

- Most basic fetch call takes single argument of the URI you want to fetch
- Returns a ‘Promise’ which contains a Fetch Response object
- If expecting JSON response first step is to extract the JSON from the body attribute of the Promise
  - Response.json() method does that
  - Where is Response.json() in the above code?
- Can then chain the resulting JSON (wrapped in another Promise) using Promise.then and process the JSON
  - In this case turns the JSON object into a string and prints to console
- So, what’s a ‘Promise’ …

Promise

- A Promise is an object representing the *eventual* completion or failure of an asynchronous operation.
- Most of the time in our application, we consume promises returned from calling some other APIs like `fetch()` or `json()`
  - But you can easily create and return a Promise from your own APIs.
- Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function.

Demo: Promise Constructor

```
let promise1 = new Promise(function (resolve, reject) {
    let r = Math.ceil(Math.random() * 10);
    if (r % 2 === 0) {
        setTimeout(function () {
            resolve('even');
        }, 300);
    } else {
        setTimeout(function () {
            reject('odd');
        }, 300);
    }
});

/* define what to do when the promise is resolved with the then() call,
 and what to do when the promise is rejected with the catch() call */
promise1
    .then(function (value) {
        console.log("Success: "+value);
    })
    .catch(function (err) {
        console.log("Error: "+ err);
    });

console.log(promise1); // this line will execute before promise resolves or rejects
```

Promise Constructor

- Syntax

```
new Promise(executor);
```

  - Executor is a function that is passed with the arguments `resolve` and `reject`.
  - The `executor` function is executed immediately by the `Promise` implementation, passing `resolve` and `reject` functions.
  - The resolve and reject functions, when called, `resolve` or `reject` the promise, respectively.
  - The executer normally initiates some asynchronous work, and then, once that completes, either calls the `resolve` function to resolve the promise or else rejects it if an error occurred.

Description

- A `Promise` is a proxy for a value not necessarily known when the promise is created.
  - It allows you to associate handlers with an asynchronous action's eventual success value or failure reason.
  - This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to *supply* the value at some point in future.
- When using fetch the promise is consumed by the chained .then methods.
  - Promise.then waits (asynchronously, just like XMLHttpRequest) for the (Promise) response to be returned and then supplies the callback function to be either the resolve or reject function for the Promise depending on whether the Promise was resolved or rejected.

Promise states

- A promise is in one of these states:
  - *pending:* initial state, neither fulfilled nor rejected.
  - *fulfilled*: meaning that the operation completed successfully.
  - *rejected*: meaning that the operation failed.

Fetch with POST data

```
let url = 'https://example.com/profile';
let data = {username: 'example'};

fetch(url, {
    method: 'POST', // or 'PUT'
    body: JSON.stringify(data),
    headers:{
        'Content-Type': 'application/json'
    }
}).then(res => res.json())
    .then(response => console.log('Success:', JSON.stringify(response)))
    .catch(error => console.error('Error:', error));
```

$.ajax versus fetch

```
$.ajax('https://jsonplaceholder.typicode.com/todos/1')
  .done(response => {
    console.log(response);
    console.log(JSON.stringify(response));
    console.log("userid is: " + response.userId);
  });


fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(res => res.json())
  .then(response => {
    console.log(response);
    console.log(JSON.stringify(response));
    console.log("userid is: " + response.userId);
  });
```

AJAX summary

- So, what makes AJAX, well… AJAX?
  - Asynchronous request.
  - Partial page rendering.

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Frictionless Flow of Information

1. Client-side programming with JavaScript is useful for making web applications highly responsive.
2. Ajax allows JavaScript to access the server in a very efficient manner using asynchronous messaging and partial page refreshing.

---

3. **Transcendental consciousness** is the experience of the home of all the laws of nature where all information is available at every point.
4. **Impulses within the transcendental field**: Communication at this level is instantaneous and effortless.
5. **Wholeness moving within itself**: In unity consciousness daily life is experienced in terms of this frictionless and effortless flow of information.

Promise

Imagine a function, createAudioFileAsync(), which asynchronously generates a sound file given a configuration record and two callback functions, one called if the audio file is successfully created, and the other called if an error occurs.

```
function successCallback(result) {
  console.log("Audio file ready at URL: " + result);}
function failureCallback(error) {
  console.log("Error generating audio file: " + error);}
createAudioFileAsync(audioSettings, successCallback, failureCallback);  //imagine using Ajax …
```

rewritten to return a promise, using it could be as simple as this:

```
createAudioFileAsync(audioSettings).then(successCallback, failureCallback);
```

That's shorthand for:

```
const promise = createAudioFileAsync(audioSettings);
promise.then(successCallback, failureCallback);
```

Promise methods

Promise.reject()Returns a new Promise object that is rejected with the given reason.
Promise.resolve()Returns a new Promise object that is resolved with the given value. If the value is a thenable (i.e. has a then method), the returned promise will "follow" that thenable, adopting its eventual state; otherwise the returned promise will be fulfilled with the value.

Promise.prototype.catch()Appends a rejection handler callback to the promise, and returns a new promise resolving to the return value of the callback if it is called, or to its original fulfillment value if the promise is instead fulfilled.
Promise.prototype.then()Appends fulfillment and rejection handlers to the promise, and returns a new promise resolving to the return value of the called handler, or to its original settled value if the promise was not handled (i.e. if the relevant handler onFulfilled or onRejected is not a function).
Promise.prototype.finally()Appends a handler to the promise, and returns a new promise which is resolved when the original promise is resolved. The handler is called when the promise is settled, whether fulfilled or rejected.