



LABFORTRAINING

AJAX

Docente: Nicola Ciaco
info@labfortraining.it

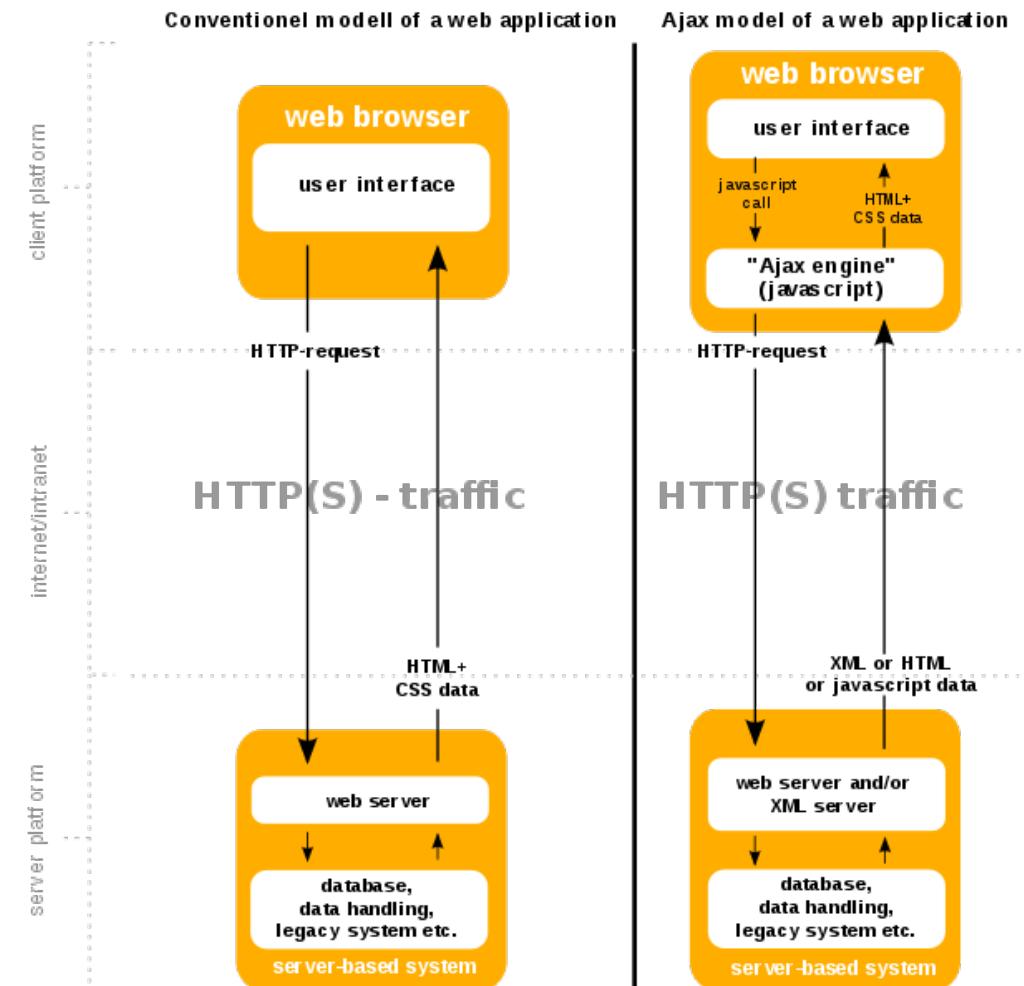
Che cos'è AJAX

- L'acronimo **AJAX** sta per **Asynchronous JavaScript And XML (JavaScript asincrono ed XML)**.
- **AJAX** ci permette di sviluppare siti/applicazioni web che sono in grado di aggiornare la pagina web su cui stiamo navigando, senza dover necessariamente ricaricare la pagina o cambiare url, attraverso delle chiamate asincrone.
- Si definisce **asincrona** una chiamata ad una risorsa esterna che non interferisce con l'esecuzione della risorsa chiamante; i risultati della risorsa esterna saranno utilizzabili solo quando disponibili senza "tempi morti" per l'utilizzatore (il caricamento della risorsa esterna avviene in *background*).



AJAX

- E' una modalità di interazione di tipo **Request-Response** tra **client e server** che sfrutta il canale **asincrono** del protocollo **HTTP**
- Il risultato è un processo che permette, nell'interazione, di **non ricaricare la pagina**, ma di **posizionare la response dove noi vogliamo all'interno del document**
- A destra la differenza tra Request / Response sincrono ed asincrono



HTTP

- HTTP (Hypertext Transfer Protocol, letteralmente protocollo di trasferimento di un ipertesto), è un protocollo, una sorta di **linguaggio di testo** che consente la comunicazione tra client e server attraverso internet.
- Ogni volta che visitiamo un sito internet, **HTTP** ci consente di visualizzare le risorse di quel sito.
- Una **richiesta HTTP** è composta dalle seguenti parti:
 - [method] [URI] [version]
 - [headers]
 - [body]

Fonte: <https://www.flaviobiscaldi.it/blog/protocollo-http-cosa-e-come-funziona>

Tipologia delle pagine web – Pagine statiche

Le pagine web possono essere **statiche** o **dinamiche**

Pagine statiche HTML

Html: non è un linguaggio nel senso tradizionale ma un impaginatore che consente di posizionare degli oggetti (testo, grafica ed elementi “inglobati”) all’interno di un file che viene interpretato dal browser (formati: .htm, .html)

Non c’è interazione con l’utente se non tramite link ipertestuali (elementi sensibili di una pagina web che consentono il salto incondizionato verso un indirizzo diverso da cui l’utente si trova oppure verso un altro punto nella stessa pagina web (ancoraggio)

Pagine dinamiche lato server

Prima di Javascript l'interazione/elaborazione dinamica era solo “lato server”

Es: nei Moduli (Forms) i dati venivano inviati al server e lì elaborati tramite programmi scritti in “linguaggi di programmazione lato server” (Perl, C, ..)

Tali programmi generavano dinamicamente pagine HTML di ritorno visualizzate direttamente dal browser dell’utente

Cosa può fare un programma “lato server”?

- ✓ eseguire ciclicamente un certo numero di operazioni strutturate in blocchi di sottoprogrammi
- ✓ validazione ed elaborazioni di dati
- ✓ accesso a basi di dati o a file di dati formato testo
- ✓ Chiamate HTTP e gestione del processo Request/Response

Programmazione web lato server

Linguaggi di programmazione

- ✓ PHP
- ✓ C# / .NET
- ✓ JAVA
- ✓ PYTHON
- ✓ RUBY
- ✓ etc...

Pagine dinamiche lato client (Javascript)

Con Javascript l'interazione diventa anche “lato client”

Alcune operazioni prima demandate esclusivamente al server vengono eseguite in locale dal javascript sul client.
Cosicché javascript può:

- ✓ eseguire ciclicamente un certo numero di operazioni strutturate in blocchi di sottoprogrammi
- ✓ validazione ed elaborazioni di dati di Form e controllo dell'integrità dei dati stessi
- ✓ produzione di effetti grafici (Poi evoluzione verso le librerie JQuery)

In questo modo una certa mole di operazioni viene decentrata dal server al client contribuendo a “decongestionare” le attività del server

Come gestire Request/Response Asincrone

- Durante questo corso vedremo 3 modalità per gestire le request e le response in modo asincrono
 - 2 di queste in Javascript nativo e 1 con la libreria jQuery:
 - Con l'oggetto **XMLHttpRequest**
 - Con il metodo **fetch()**
 - Con il metodo **\$.ajax()**

XMLHttpRequest

- Ajax ruota intorno all'oggetto **XMLHttpRequest** che rappresenta l'intermediario tra il codice JavaScript eseguito sul browser e il codice eseguito sul server.
- L'oggetto **XMLHttpRequest** può essere utilizzato per scambiare dati con un server Web dietro le quinte. Ciò significa che è possibile aggiornare parti di una pagina Web, senza ricaricare l'intera pagina.
- Per gestire il processo request/response asincrono (Ajax) possiamo utilizzare l'oggetto **XMLHttpRequest**, che è in grado di incapsulare le richieste e le risposte che utilizzano il **canale asincrono** del protocollo **HTTP**.



XMLHttpRequest - Step del processo (1)

- 1) Creare una nuova istanza dell'oggetto XMLHttpRequest



```
1 // istanza dell'oggetto XMLHttpRequest  
2 const xhr = new XMLHttpRequest();
```

XMLHttpRequest - Step del processo (2)

2) Assegniamo alla proprietà **onreadystatechange** che è associata all'evento **readystatechange** dell'istanza di XMLHttpRequest una funzione che avrà il compito di catturare i vari stati di avanzamento della request, essa ha anche il compito di manipolare la risposta del server e mostrare i dati nella pagina web.

```
1 // istanza dell'oggetto XMLHttpRequest
2 const xhr = new XMLHttpRequest();
3
4 xhr.onreadystatechange = function() {
5   if (xhr.readyState == 4 && xhr.status == 200) {
6     document.getElementById("myDiv").innerHTML = xhr.responseText;
7   }
8 }
```

XMLHttpRequest - Step del processo (2)

- Immagazzinata nella proprietà xhr.responseText avremo la risposta che viene restituita dal lato server. In questo caso la risposta è di tipo testuale (può essere json, text, HTML, XML)

```
1 // istanza dell'oggetto XMLHttpRequest
2 const xhr = new XMLHttpRequest();
3
4 xhr.onreadystatechange = function() {
5   if (xhr.readyState == 4 && xhr.status == 200) {
6     document.getElementById("myDiv").innerHTML = xhr.responseText;
7   }
8 }
```

XMLHttpRequest - Step del processo (2)

- La proprietà xhr.readyState può assumere 5 valori numerici differenti in base allo stato in cui si trova la request effettuata
- In basso a destra possiamo vedere i valori con le rispettive descrizioni



```
1 // istanza dell'oggetto XMLHttpRequest
2 const xhr = new XMLHttpRequest();
3
4 xhr.onreadystatechange = function() {
5   if (xhr.readyState == 4 && xhr.status == 200) {
6     document.getElementById("myDiv").innerHTML = xhr.responseText;
7   }
8 }
```

Valore	Descrizione
0	Richiesta non inizializzata
1	Connessione al server stabilita
2	Ricezione degli header HTTP
3	Ricevimento della risposta
4	Operazione completata

Fonte: <https://www.html.it/pag/50473/ajax-e-javascript/>

XMLHttpRequest - Step del processo (2)

- La proprietà xhr.status può assumere differenti valori numerici,
- Gli stati che iniziano con 2 come nel nostro caso, rappresentano i casi di richieste completate con successo
- Gli stati che iniziano con 4 indicano generalmente un errore lato client, uno degli errori più comuni è il 404 Not Found
- Gli stati che iniziano con 5 indicano un errore nel server, il più generico è 500 internal server error

```
1 // istanza dell'oggetto XMLHttpRequest
2 const xhr = new XMLHttpRequest();
3
4 xhr.onreadystatechange = function() {
5   if (xhr.readyState == 4 && xhr.status == 200) {
6     document.getElementById("myDiv").innerHTML = xhr.responseText;
7   }
8 }
```

Fonte: <https://developer.mozilla.org/it/docs/Web/HTTP>Status>



XMLHttpRequest - Step del processo (2) (status)

Codice	Descrizione
200 OK	La richiesta è andata a buon fine
301 Moved Permanently	La risorsa richiesta è stata spostata definitivamente ad un nuovo URI
404 Not Found	La risorsa richiesta non è stata trovata
500 Internal Server Error	Errore generico causato di solito da una configurazione errata del server

XMLHttpRequest - Step del processo (3)

3) Dopo aver preparato la funzione che gestirà l'evento di risposta, apriamo una connessione HTTP con il server tramite il metodo **open()**. I parametri che passiamo a questo metodo rappresentano:

- Il method HTTP (GET nel nostro caso)
- l'*URL* della pagina o dello script server side richiesto
- un valore booleano opzionale che indica se la richiesta deve essere effettuata in maniera *asincrona* (true) o sincrona (false).

```
1 // istanza dell'oggetto XMLHttpRequest
2 const xhr = new XMLHttpRequest();
3
4 xhr.onreadystatechange = function() {
5   if (xhr.readyState == 4 && xhr.status == 200) {
6     document.getElementById("myDiv").innerHTML = xhr.responseText;
7   }
8 }
9
10 xhr.open('GET', '<server_url>', true);
11
```



XMLHttpRequest - Step del processo (4)

4) Infine inviamo la richiesta tramite il metodo `send()`.

```
1 // istanza dell'oggetto XMLHttpRequest
2 const xhr = new XMLHttpRequest();
3
4 xhr.onreadystatechange = function() {
5   if (xhr.readyState == 4 && xhr.status == 200) {
6     document.getElementById("myDiv").innerHTML = xhr.responseText;
7   }
8 }
9
10 xhr.open('GET', '<server_url>', true);
11 xhr.send();
```

I Verbi/Metodi delle richieste HTTP

- HTTP definisce un insieme di metodi di richiesta per indicare l'azione desiderata da eseguire per una determinata risorsa. Sebbene possano anche essere nomi, questi metodi di richiesta vengono talvolta definiti verbi HTTP.
- I più utilizzati sono:
 - **GET** utilizzato per recuperare una risorsa dal server
 - **POST** utilizzato per inviare una risorsa sul server
 - **PUT** utilizzato anch'esso per inviare una risorsa sul server con la differenza che generalmente viene utilizzato fare un update di un dato già presente
 - **DELETE** utilizzato per cancellare una risorsa dal server

Per approfondire: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>

Formati per l'interscambio di dati

- Nel web sono presenti due principali formati per l'interscambio di dati tra client e server
- XML (eXtensible Markup Language) come si evince dal nome è un linguaggio di markup simile all'HTML
- JSON (Javascript Object Notation) Si tratta di un formato testuale per la strutturazione di dati

XML (eXtensible Markup Language)

- XML è un linguaggio di markup creato dal World Wide Web Consortium ([W3C](#)) per definire una sintassi per la codifica dei documenti che sia gli umani che le macchine potrebbero leggere. Lo fa attraverso l'uso di tag che definiscono la struttura del documento



```
1 <employees>
2   <employee>
3     <firstName>John</firstName>
4     <lastName>Doe</lastName>
5   </employee>
6   <employee>
7     <firstName>Anna</firstName>
8     <lastName>Smith</lastName>
9   </employee>
10  <employee>
11    <firstName>Peter</firstName>
12    <lastName>Jones</lastName>
13  </employee>
14 </employees>
```

JSON (Javascript Object Notation)

- è un formato relativamente recente per lo scambio di dati in applicazioni web client-server:
 - è stato progettato per essere minimale, testuale e integrato in JavaScript.

```
1 {  
2     "employees": [  
3         { "firstName":"John", "lastName":"Doe" },  
4         { "firstName":"Anna", "lastName":"Smith" },  
5         { "firstName":"Peter", "lastName":"Jones" }  
6     ]  
7 }
```

Eseguire una POST con XMLHttpRequest

- Di seguito vediamo la struttura di una **POST** che risulta molto simile alla struttura di una GET con la differenza che in più andiamo ad aggiungere alcune informazioni alla request, inserendole nell'header, con il metodo **setRequestHeader**

```
1 const xhr = new XMLHttpRequest();
2
3 let json = JSON.stringify({
4   name: "John",
5   surname: "Smith"
6 });
7
8 xhr.open("POST", '/submit')
9 xhr.setRequestHeader('Content-type', 'application/json; charset=utf-8');
10
11 xhr.send(json);
```

jQuery

- **JQuery** è un libreria **javascript** che permette ai **Designer/Developer** di sviluppare pagine web con funzionalità interattive in maniera semplice.
- Non è solo una libreria che semplifica la manipolazione del DOM, ma ci mette a disposizione dei metodi per interagire con un server attraverso delle richieste HTTP asincrone
- Per utilizzare **jQuery** basta includere la CDN(content distribution network) nell'header della nostra pagina html:

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
```

Request/Response HTTP asincrone con jQuery

- **JQuery** come abbiamo detto nella slide precedente ci fornisce metodi per effettuare richieste http asincrone verso un webserver
- I metodi che ci fornisce jQuery sono:
 - `$.get()`
 - `$.post()`
 - `$.ajax()`



\$.get()

- \$.get ci permette di effettuare richieste HTTP GET

```
1 $.get( '<server url>' )
2     .done(
3         function(data) {
4             console.log(data)
5         }
6     ).fail(
7         function(xhr) {
8             console.log(xhr.statusText)
9         }
10    )
```



\$.get()

- \$.get accetta come argomento l'indirizzo della risorsa da chiamare
- ci espone i metodi **done** e **fail**
- **done** accetta come argomento una funzione di callback per gestire le response per le request effettuate e completate con successo
- **fail** come done accetta come argomento una funzione di callback per gestire gli errori che può generare la request

```
● ● ●  
1 $.get('<server url>')  
2   .done(  
3     function(data) {  
4       console.log(data)  
5     }  
6   ).fail(  
7     function(xhr) {  
8       console.log(xhr.statusText)  
9     }  
10 )
```



\$.post()

- \$.post ci permette di effettuare richieste HTTP POST

```
1 $.ajaxSetup({  
2     headers: {  
3         "Content-Type": "application/json"  
4     }  
5 })  
6  
7 $.post('<server url>', {nome: 'Mario', cognome: 'Rossi'})  
8     .done(function(data) {  
9         console.log(data)  
10    }).fail(  
11        function(xhr) {  
12            console.log(xhr.statusText)  
13        })
```



- \$.post accetta come primo argomento l'indirizzo della risorsa da chiamare e come secondo argomento il body della request
- Per poter aggiungere altre informazioni come gli headers dobbiamo utilizzare il metodo \$.ajaxSetup
- ci espone i metodi **done** e **fail**
- **done** accetta come argomento una funzione di callback per gestire le response per le request effettuate e completate con successo
- **fail** come done accetta come argomento una funzione di callback per gestire gli errori che può generare la request

\$.post()

```
● ● ●  
1 $.ajaxSetup({  
2   headers: {  
3     "Content-Type": "application/json"  
4   }  
5 })  
6  
7 $.post('<server url>', {nome: 'Mario', cognome: 'Rossi'})  
8   .done(function(data) {  
9     console.log(data)  
10  }).fail(  
11    function(xhr) {  
12      console.log(xhr.statusText)  
13  })
```



\$.ajax()

- \$.ajax ci permette di effettuare richieste HTTP GET, POST, PUT, DELETE

```
1 $.ajax({
2     url: 'http://localhost:3000/studenti',
3     method: 'GET',
4     dataType: 'json',
5 }).done(
6     function(data) {
7         console.log(data)
8     }
9 ).fail(
10    function(xhr) {
11        console.log(xhr)
12    }
13 ).always(
14    function() {
15        console.log('operazione completata')
16    }
17 )
```



\$.ajax()

- \$.ajax ci permette di effettuare richieste HTTP GET, POST, PUT, DELETE

```
1 $.ajax({
2     url: '<server url>',
3     method: 'POST',
4     dataType: 'json',
5     data: {nome: 'Mario', cognome: 'Rossi'},
6     headers: {
7         "Content-Type": "application/json"
8     }
9 }).done(
10     function(data) {
11         console.log(data)
12     }
13 ).fail(
14     function(xhr) {
15         console.log(xhr)
16     }
17 ).always(
18     function() {
19         console.log('operazione completata')
20     }
21 )
```

\$.ajax()

- \$.ajax accetta come argomento un oggetto contenente i dati per poter effettuare la request, come url, method, dataType, headers, data
- ci espone i metodi **done**, **fail** e **always**
- **done** accetta come argomento una funzione di callback per gestire le response per le request effettuate e completate con successo
- **fail** come done accetta come argomento una funzione di callback per gestire gli errori che può generare la request
- **always** riceve come argomento una funziona di callback che viene eseguita alla fine dell'esecuzione della request, qualunque sia il risultato ottenuto

FETCH API

- L'utilizzo di **XMLHttpRequest** per la gestione di chiamate HTTP da JavaScript risulta abbastanza prolioso e scomodo
- Nel tempo sono nate diverse alternative come nel caso di jQuery con i metodi `$.get`, `$.post`, `$.ajax`
- il gruppo di lavoro WHATWG ha definito recentemente una alternativa a **XMLHttpRequest**: l'API **fetch()**
- `fetch()` ci permette di effettuare richieste HTTP GET, POST, PUT, DELETE, ecc...

FETCH API

- **fetch** rispetto ad **XMLHttpRequest** ha una sintassi più semplice e gestisce le chiamate asincrone con le promise, ed è pensata per essere estesa ed utilizzabile in diversi contesti, non solo all'interno del browser
- Dal momento che **fetch** è una API recente e non supportata da tutti i browser e ambienti, possiamo ricorrere ai **polyfill** come ad esempio **isomorphic-fetch**

FETCH API

- Il metodo fetch accetta come primo argomento l'indirizzo della risorsa
- Di default se non viene specificato in un secondo argomento, il method è GET
- In caso di successo della chiamata la **promise** viene risolta e quindi gestita nel **.then()**, la funzione di callback gestirà la risposta del server sotto forma di Response

```
1 fetch("<server-url>")
2   .then(response => {
3     console.log(response);
4   })
5   .catch(error => console.log("Si è verificato un errore!"))
```

FETCH API

- Vediamo nella **Response** le proprietà più utilizzate dell'oggetto che ci ritorna il server

Proprietà	Descrizione
status	È un valore intero che indica il codice di stato HTTP inviato dal server, per esempio 200 in caso di risposta con successo
statusText	È una stringa associata al codice di stato, che ne descrive testualmente il significato. Ad esempio, se il codice di risposta è 200, la stringa sarà "OK"
ok	È un valore booleano che indica se la risposta del server è stata positiva, cioè se il codice di stato restituito è compreso tra 200 e 299, estremi inclusi

FETCH API

- La **Response** espone inoltre dei metodi per interpretare una risposta HTTP

Metodo	Descrizione
text()	Restituisce il contenuto sotto forma di testo
json()	Effettua il parsing del contenuto e lo restituisce sotto forma di oggetto
blob()	Restituisce il contenuto sotto forma di dati non strutturati (<i>blob</i>)
arrayBuffer()	Restituisce il contenuto strutturato in un <i>arrayBuffer</i>



FETCH API

- Di seguito vediamo la gestione completa di una GET con fetch

```
1 fetch("<server url>")  
2   .then(response => {  
3     if (response.ok) {  
4       return response.json();  
5     }  
6   })  
7   .then(obj => console.log(obj))  
8   .catch(error => console.log("Si è verificato un errore!"))
```

FETCH API

- Per poter effettuare una POST oppure qualsiasi altra azione dobbiamo necessariamente passare un altro argomento alla fetch, oppure creare un oggetto request che contiene tutti i dati della richiesta

```
● ● ●  
1 const request = new Request("<server url>", {  
2   method: "POST",  
3   headers: new Headers({  
4     "Content-Type": "application/json"  
5   }),  
6   body: JSON.stringify({  
7     titolo: "Un articolo",  
8     autore: "Mario Rossi"  
9   })  
10});  
11 fetch(request).then(...).catch(...)
```



FETCH API

- in alternativa si può passare come primo argomento l'url e come secondo argomento un oggetto contenente i parametri della request



```
1 fetch("<server url>", {  
2   method: "POST",  
3   headers: new Headers({  
4     "Content-Type": "application/json"  
5   }),  
6   body: JSON.stringify({  
7     titolo: "Un articolo",  
8     autore: "Mario Rossi"  
9   })  
10 }).then(...).catch(...)
```