
ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

# Introduzione a Javascript

## Temi trasversali (I parte)

**Fabio Vitali**
Corso di laurea in Informatica
Alma Mater – Università di Bologna

# Oggi parleremo di…

Javascript

- Sintassi base (parte I)
- Javascript client-side (parte II)
- Sintassi avanzata (parte III)
- Nuove feature di ES 2015 (parte IV)

***Temi trasversali***

- ***AJAX***
- ***Programmazione asincrona***


ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

# AJAX

# AJAX: Introduzione

AJAX (**A**synchronous **J**avaScript **A**nd **X**ML) è una tecnica per la creazione di applicazioni Web interattive.

Permette l'aggiornamento **asincrono** di **porzioni** di pagine HTML

Utilizzato per incrementare:

- l'**interattività**
- la **velocità**
- l'**usabilità**

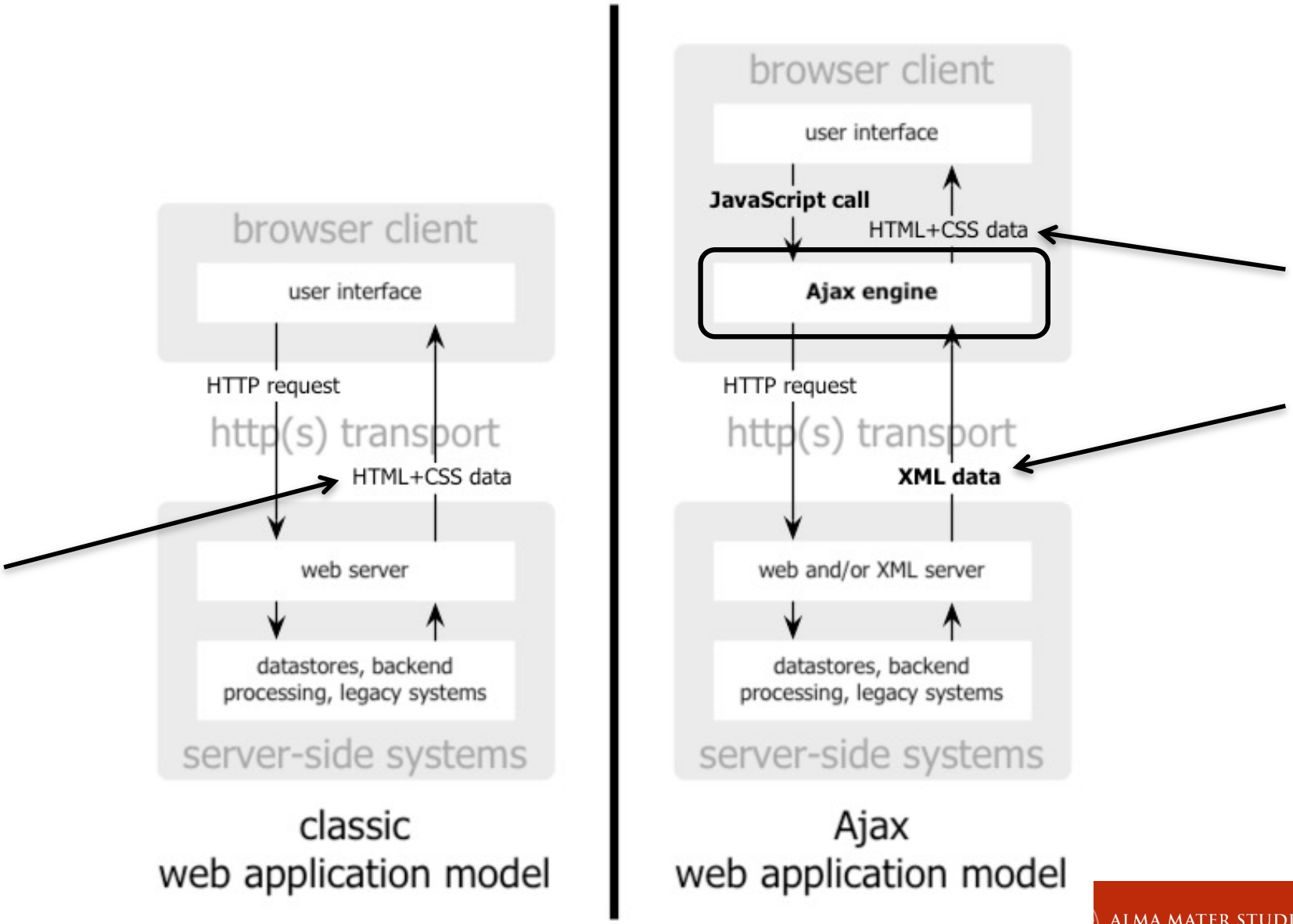
# AJAX: Discussione

**Non è un linguaggio di programmazione o una tecnologia specifica**

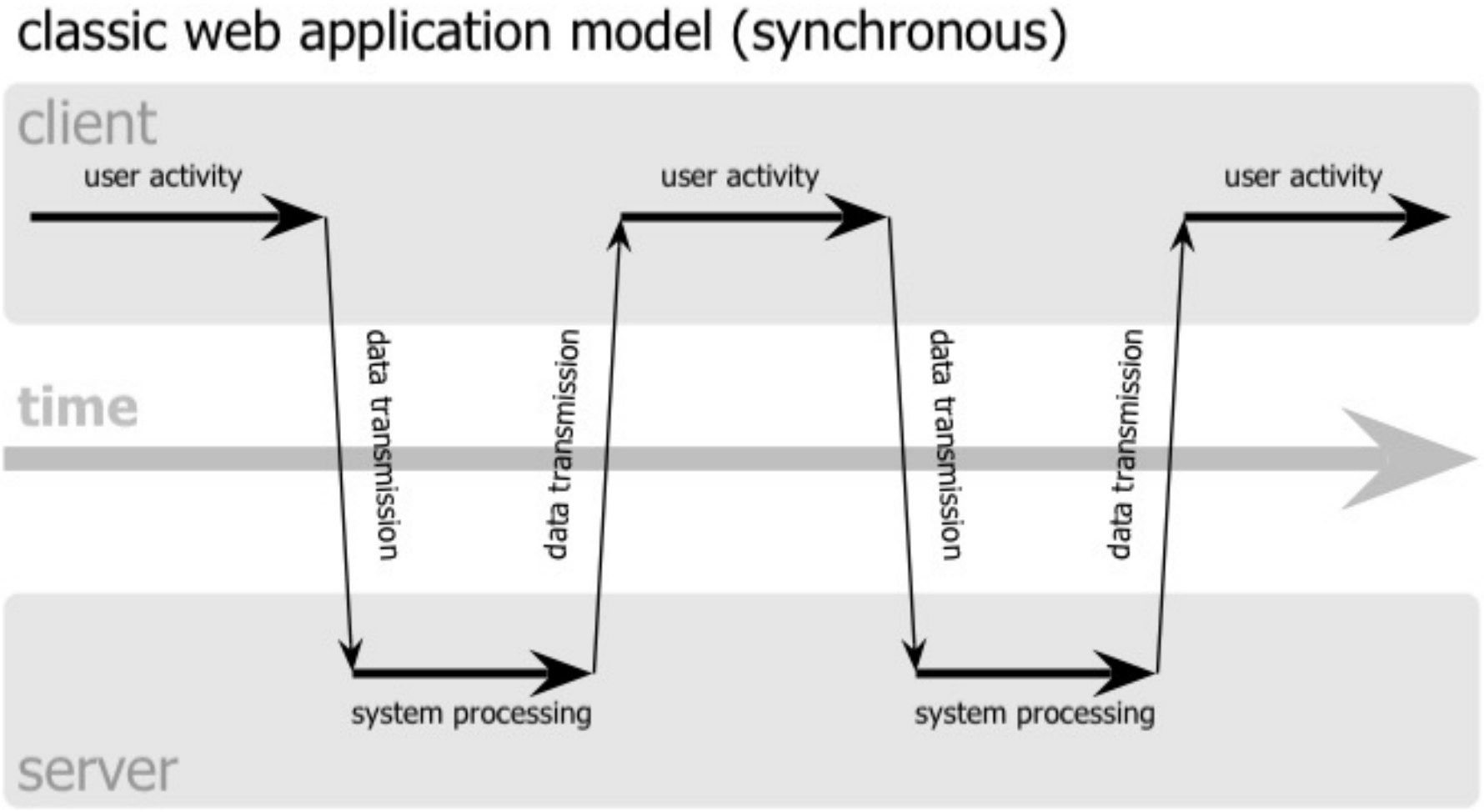
E' un termine che indica l'utilizzo di una combinazione di tecnologie comunemente utilizzate sul Web:

- HTML e CSS
- DOM (modificabile attraverso JavaScript per la manipolazione dinamica dei contenuti e dell'aspetto)
- La libreria XMLHttpRequest (XHR) per lo scambio di messaggi asincroni fra browser e web server
- XML o JSON come meta-linguaggi dei dati scambiati

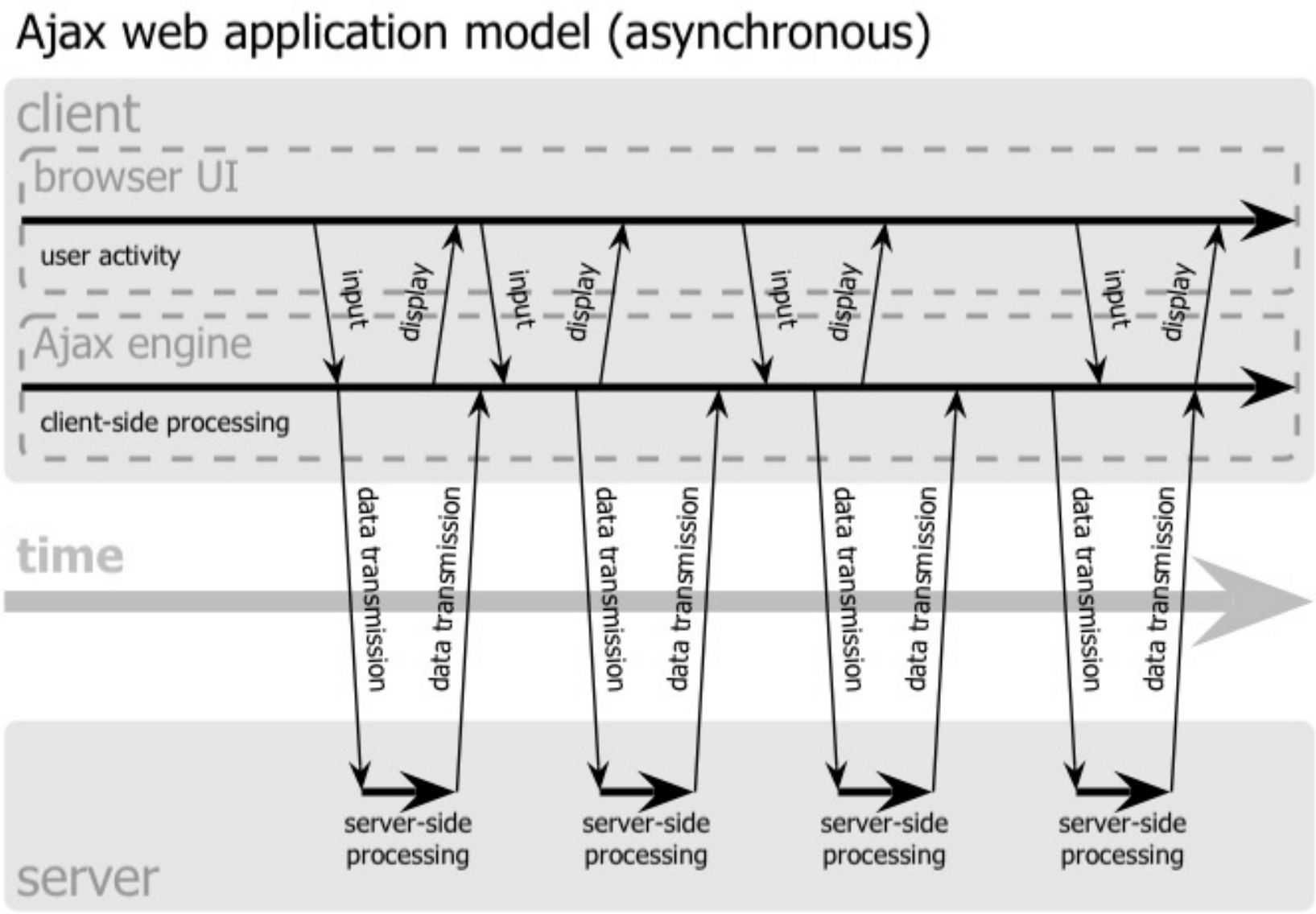
# AJAX: Architettura (1)



# AJAX: Architettura (2)



# AJAX: Architettura (3)



# AJAX: Un po’ di storia

Inizialmente sviluppatto da Microsoft (*XMLHttpRequest*) come oggetto ActiveX

In seguito implementato in tutti i principali browser ad iniziare da Mozillla 1.0 sebbene con alcune differenze

Il termine **Ajax** è comparso per la prima volta nel 2005 in un articolo di *Jesse James Garrett*

# Ajax: Pregi

- **Usabilità**
  - Interattività (Improve user experience)
  - Non costringe l'utente all'attesa di fronte ad una pagina bianca durante la richiesta e l'elaborazione delle pagine (non più click-and-wait)
- **Velocità**
  - Minore quantità di dati scambiati (non è necessario richiedere intere pagine)
  - Una parte della computazione è spostata sul client
- **Portabilità**
  - Supportato dai maggiori browser
  - Se correttamente utilizzato è platform-independent
  - Non richiede plug-in

# Ajax: Difetti

- **Usabilità**
  - Non c'è navigazione: il pulsante “back” non funziona
  - Non c'è navigazione: l'inserimento di segnalibri non funziona
  - Poiché i contenuti sono dinamici non sono correttamente indicizzati dai motori di ricerca
- **Accessibilità**
  - Non supportato da browser non-visuali
  - Richiede meccanismi di accesso alternativi
- **Configurazione**:
  - È necessario aver abilitato Javascript
  - in Internet Explorer è necessario anche aver abilitato gli oggetti ActiveX
- **Compatibilità**:
  - È necessario un test sistematico sui diversi browser per evitare problemi dovuti alle differenze fra i vari browser
  - Richiede funzionalità alternative per i browser che non supportano Javascript

# Creare un’applicazione AJAX

Un’applicazione AJAX è divisa in alcuni momenti chiave:

1. Creazione e configurazione delle richieste per il server
   - Usando XMLHTTPRequest
   - *Usando funzioni di libreria che nascondono XMLHTTPRequest*
   - *Usando fetch()*
2. Attivazione della richiesta HTTP…
3. ***… passa del tempo…***
4. … ricezione della risposta HTTP e analisi dei dati (o errore)
5. Modifiche al DOM della pagina

# XHR: la richiesta

```
if (window.XMLHttpRequest) {
    http_request = new XMLHttpRequest();
    ...
```

La funzione open prepara la connessione HTTP (non viene ancora attivata). Due tipi di connessioni: sincrona e bloccante, oppure asincrona.

```
http_request.open('GET','http://www.example.org/file.json',false);
```

Nel caso di funzione asincrona, prima di attivare la richiesta è necessario specificare la funzione che si occuperà di gestire la risposta e aprire la connessione con il server

```
http_request.onreadystatechange = myHandler;

http_request.open('GET','http://www.example.org/file.json',true);
```

I parametri della 'open' specificano:

- il metodo HTTP della richiesta,
- l'URL a cui inviare la richiesta,
- un booleano che indica se la richiesta è asincrona;

# XHR: invio della richiesta

La richiesta viene inviata per mezzo di una ‘send’:

```
http_request.send(null);
```

Il parametro della ‘send’ contiene il body della risorsa da inviare al server:

- per una POST ha la forma di una query-string

```
name=value&anothername=othervalue&so=on
```

- per un GET ha valore “null” (in questo caso i parametri sono passati tramite l’URL indicato della precedente “open”)
- può anche essere un qualsiasi altro tipo di dati; in questo caso è necessario specificare il tipo MIME dei dati inviati:

```
http_request.setRequestHeader('Content-Type', 'mime/type');
```

# XHR: la risposta

La funzione asincrona incaricata di gestire la risposta deve controllare lo stato della richiesta:

```
function myHandler() {
    if (http_request.readyState == 4) {
            // risposta ricevuta
    } else {
            // risposta non ricevuta ancora
    }
    ...
}
```

I valori per 'readyState' possono essere:

0 = uninitialize

1 = loading

2 = loaded

3 = interactive

4 = complete

# XHR: la risposta

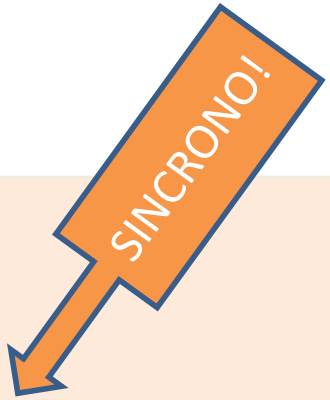
E’ poi necessario controllare lo status code della risposta HTTP:

```
if (http_request.status == 200) {
    // perfetto!
}
else {
    // c'è stato un problema con la richiesta,
    // per esempio un 404 (File Not Found)
    // oppure 500 (Internal Server Error)
}
```

Infine è possibile leggere la risposta inviata dal server utilizzando:

- http_request.responseText che restituisce la risposta come testo semplice
- http_request.responseXML che restituisce la risposta come XMLDocument

# XHR codice completo (modalità sincrona)




```
function getData(){
  try {
    // Attiva la connessione Ajax
    myXHR = new XMLHttpRequest();
    myXHR.open("GET", "http://www.server.it/server ", false);
    myXHR.send(null);
```

Browser bloccato e non accetta interazione con l'utente

```
    //legge la risposta
    let d = JSON.parse(myXHR.responseText);

    // modifica il documento corrente
    let data = prepareHTML(d);
    document.querySelector('#result').innerHTML = data ;
  } catch(error) {
      alert("Caricamento impossibile");
  }
}
```

# XHR codice completo (modalità asincrona)


```
function getData(){
    // Attiva la connessione Ajax
    myXHR = new XMLHttpRequest();
    myXHR.onreadystatechange = myHandler;
    myXHR.open("GET", "http://www.server.it/server ", true);
    myXHR.send(null);
}
```

Browser immediatamente libero e accetta interazioni con l'utente

```
function myHandler() {
  if (myXHR.readyState == 4) {
    if (myXHR.status == 200) {
       //legge la risposta
       let d = JSON.parse(myXHR.responseText);
       // modifica il documento corrente
       let data = prepareHTML(d);
       document.querySelector('#result').innerHTML = data ;
    }
  }
}
```

# Semplificando...

La complessità di XMLHttpRequest e le differenze di implementazione tra browser e browser hanno portato a suggerire molte alternative:

- jQuery è stato introdotto ed è diventato famoso anche perché forniva un meccanismo per fare connessioni Ajax molto più semplice anche se ancora basato su callback.
- Sia React sia Angular sia Vue introducono librerie interne e/o esterne per fare connessioni Ajax nel loro framework
- Con il passaggio da W3C a WhatWG, e con l'introduzione delle promesse, è stata proposta e standardizzata una specifica API nativa, chiamata Fetch, per realizzare connessioni Ajax con una libreria nuova NON basata su XMLHttpRequest.

# jQuery: Ajax

Una singola funzione si occupa di tutta la comunicazione asincrona usando XMLHttpRequest (XHR):

```
$.ajax({
    method: "GET",
    url: "http://www.server.it/server",
    success: function(d) {
            let data = prepareHTML(d);
            $('#result').html(data);
            alert("Caricamento effettuato");
    },
    error: function(data) {
            alert("Caricamento impossibile");
    }
})
```

membri url e success sono obbligatori. Alcune funzioni equivalenti ($.get(), $.post() e $.put() ) sono disponibili per maggiore rapidità.

Le funzioni success() e error() vengono chiamate appena la connessione HTTP si è conclusa e dunque readystate = 4 (***loaded***).

# jQuery: anche con promesse

Una alternativa al call-back è usare le promesse.

Il metodo $.ajax() è già una promessa, per cui si possono creare catene di funzioni then() con due parametri, la funzione da chiamare in caso di successo e quella in caso di insuccesso:

```
let good = (d) => {
    let data = prepareHTML(d);
    $('#result').html(data);
     alert("Caricamento effettuato");
};
let bad = () => { alert("Caricamento impossibile"); };

$.get("http://www.server.it/server").then(good, bad);
```

# Axios

- Una libreria open source del 2015 basata su promesse, usata frequentemente su React e Vue (e anche su Angular se vi sta antipatica la libreria standard).

```
const axios = require('axios');

axios.get("http://www.server.it/server")
    .then((d) => {
        let data = prepareHTML(d);
        document.querySelector('#result').innerHTML = data;
        alert('Caricamento effettuato');
    }).catch((error) => {
        alert("Caricamento impossibile");
    });
```

# Axios

- Oppure usando async/await:

```
const axios = require('axios');

async function getData() {
  try {
     let response = await axios.get("http://www.server.it/server");
     let data = prepareHTML(response.data);
     document.querySelector('#result').innerHTML = data;
     alert('Caricamento effettuato');
  } catch((error) => {
     alert("Caricamento impossibile");
  }
}

getData();
```

# Fetch

- Una libreria standard dei browser ispirata a jQuery.
- Un altro modo per fare richieste Ajax basato su promesse.
- Ogni singolo passaggio è una promessa, incluso la ricezione e la decodifica della risposta:

```
fetch("http://www.server.it/server")
.then(response => response.json())
.then(d => {
   let data = prepareHTML(d);
   document.querySelector('#result').innerHTML = data;
   alert('Caricamento effettuato');
}).catch((error) => {
   alert("Caricamento impossibile"); })
});
```

# Fetch

Un parametro opzionale permette di configurare la richiesta:

```
fetch("http://www.server.it/server", {
    method: 'POST',
    headers: {
       'Content-Type': 'application/json'
    },
    body: JSON.stringify(formData),
}).then(
   response => response.json())
.then(d => {
   let data = prepareHTML(d);
   document.querySelector('#result').innerHTML = data;
   alert('Caricamento effettuato');
}).catch((error) => {
   alert("Caricamento impossibile"); })
});
```


ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

# Programmazione asincrona

# Programmazione asincrona

La caratteristica più peculiare e tipica di Javascript, evidente immediatamente, è la asincronicità come filosofia di design.

1. Una richiesta Ajax viene eseguita asincronicamente rispetto alla navigazione della pagina HTML
2. La gestione dei dati ricevuti via Ajax viene eseguita asincronicamente rispetto alla emissione della richiesta
3. La gestione degli eventi dell'utente viene eseguita asincronicamente rispetto alla specifica della funzione callback
4. setTimeout() posticipa di n millisecondi l'esecuzione di una funzione
5. ...

# Programmazione asincrona (1)

Ad esempio:

```
let msg = document.getElementById('msg') ;
msg.innerHTML = '<p>via!</p>' ;
window.setTimeout(() => {
    msg.innerHTML += '<p>0 secondi sono passati</p>' ;
}, 0) ;
```

Il paragrafo `<p>via</p>` compare giustamente prima di quello interno alla funzione. Ma il paragrafo `<p>via</p>` comparirebbe prima lo stesso anche se fosse:

```
let msg = document.getElementById('msg') ;
window.setTimeout(() => {
    msg.innerHTML += '<p>0 secondi sono passati</p>' ;
}, 0);
msg.innerHTML = '<p>via!</p>' ;
```

L'interprete Javascript prima esegue completamente lo script in corso, e DOPO esegue quelli in callback.

# Programmazione asincrona (2)

Questa è una domanda di un compito di TW di tempo fa:

Qual è il contenuto dell'elemento con id test dopo l'esecuzione dello script dell'esempio seguente? Si assuma che esista una funzione *AjaxCall()* con parametri appropriati e che la connessione Ajax abbia restituito un codice "200";

```
function test() {
        el.innerHTML += "Ho messo ";
        ajax = new AjaxCall('GET', "http://www.mysite.com", true);
        ajax.success = (data) => { el.innerHTML += "l'arrosto "};
        ajax.error = (data) => { el.innerHTML += "la verdura "};
        ajax.send();
        el.innerHTML += "nel forno " ;
}

let el = document.getElementById('test')
el.innerHTML = '';
test();
```

# Programmazione asincrona (2)

Questa è una domanda di un compito di TW di tempo fa:

Qual è il contenuto dell'elemento con id test dopo l'esecuzione dello script dell'esempio seguente? Si assuma che esista una funzione *AjaxCall()* con parametri appropriati e che la connessione Ajax abbia restituito un codice "200";

```
function test() {
        el.innerHTML += "Ho messo ";
        ajax = new AjaxCall('GET', "http
        ajax.success = (data) => { el.in
        ajax.error = (data) => { el.inne
        ajax.send();
        el.innerHTML += "nel forno " ;
}

let el = document.getElementById('test'
el.innerHTML = '';
test();
```

Hanno risposto TUTTI:
"Ho messo l'arrosto nel forno"

Invece la risposta esatta è
"Ho messo nel forno l'arrosto"

perché la funzione del risultato della chiamata Ajax viene eseguita asincronicamente rispetto allo script di invocazione

# Programmazione asincrona (3)

Abbiamo esigenze di asincronicità ogni volta che abbiamo l'esigenza di chiamare un servizio sulla cui disponibilità o sui cui tempi di esecuzione non abbiamo controllo.

```
function searchSomeProducts(query,db) {
  let database = dbConnect("http://wwww.site.com",db,acct,pwd);
  let result = database.search(query, false);
  let output = prepareTable(result);
  document.getElementById("display").innerHTML = output;
}
```

Non ho controllo sui tempi di esecuzione del comando ***database.search***, che potrebbe metterci molto tempo.

Nel frattempo il processo è bloccato in attesa del ritorno della funzione, e l'utente percepisce un'esecuzione a scatti, non fluida, non responsive.

# Programmazione asincrona (4)

La situazione potrebbe peggiorare se l'esecuzione avesse necessità, a catena, di tante altre richieste esterne non controllabili.

- *Supponiamo di avere tre database grandi, uno di ordini, uno di prodotti ed uno di opinioni dei clienti.*
- *Vogliamo mandare mailing list personalizzate, per offrire a tutti quelli che hanno comprato i nostri prodotti delle alternative migliori.*
- *Quindi cerchiamo tra gli ordini i prodotti ordinati, tra i prodotti quelli dello stesso tipo, e tra le opinioni i prodotti con punteggio maggiore.*

```
function searchBetterProducts(query) {
  let async = ???;
  let orderDB = dbConnect("http://wwww.site.com","orders",acct,pwd)
  let productDB = dbConnect("http://wwww.site.com","products",acct,pwd)
  let opinionDB = dbConnect("http://wwww.site.com","opinions",acct,pwd)

  let orders = orderDB.search(query, async);
  let products = productDB.search(orders, async) ;
  let opinions = opinionDB.search(products, async) ;

  let output = prepareTable(orders, products, opinions);
  document.getElementById("display").innerHTML = output;
}
```

# Programmazione asincrona (5)

## Soluzione -1: codice asincrono e speriamo

Beh, dai, facciamolo asincrono e basta... cosa vuoi che succeda?

***Non funziona!*** Il secondo e il terzo search vengono attivati quando sto ancora aspettando il risultato del primo, quindi orders è ancora vuoto e products pure.

```
function searchBetterProducts(query) {
  let async = true;
  let orderDB = dbConnect("http://wwww.site.com","orders",acct,pwd)
  let productDB = dbConnect("http://wwww.site.com","products",acct,pwd)
  let opinionDB = dbConnect("http://wwww.site.com","opinions",acct,pwd)

  let orders = orderDB.search(query, async);
  let products = productDB.search(orders, async) ;
  let opinions = opinionDB.search(products, async) ;

  let output = prepareTable(orders, products, opinions);
  document.getElementById("display").innerHTML = output;
}
```

# Programmazione asincrona (6)

## Soluzione 0: codice bloccante e amen

Aspettiamo, intanto ci fumiamo una sigaretta e chiacchieriamo con gli amici

```
function searchBetterProducts(query) {
  let async = false;
  let orderDB = dbConnect("http://wwww.site.com","orders",acct,pwd)
  let productDB = dbConnect("http://wwww.site.com","products",acct,pwd)
  let opinionDB = dbConnect("http://wwww.site.com","opinions",acct,pwd)

  let orders = orderDB.search(query, async);
  let products = productDB.search(orders, async) ;
  let opinions = opinionDB.search(products, async) ;

  let output = prepareTable(orders, products, opinions);
  document.getElementById("display").innerHTML = output;
}
```

# Programmazione asincrona (7)

## Soluzione 1: logica server-side

Potrei usare un linguaggio multi-threaded server-side, e fare un'unica richiesta al server, che si occupi di tutti i dettagli.

```
function searchBetterProducts(query) {
  let async = false;
  let DB = dbConnect("http://wwww.site.com","*",acct,pwd)

  let all = DB.chainedSearch(query,["order","products","opinions"],async);

  let output = prepareTable(all.orders, all.products, all.opinions);
  document.getElementById("display").innerHTML = output;
}
```

La richiesta è comunque sincrona, ho comunque un'attesa, e non ho nessun particolare vantaggio da Ajax. Inoltre distribuisco la logica dell'applicazione in due luoghi, con evidente complessità della gestione:

- Chi si occupa della gestione delle eccezioni e degli errori?
- Chi si occupa del filtro delle opinioni negative o false?

# Programmazione asincrona (8)

## Soluzione 2: codice asincrono e callback

Nei casi semplici posso passare una funzione callback come argomento di chiamata a funzione, che viene eseguita alla conclusione del servizio.

```
function searchSomeProducts(query) {
  let async = true;
  let orderDB = dbConnect("http://wwww.site.com","orders",acct,pwd)

  orderDB.search(query, async, (data) => {
      let output = prepareTable(orders);
      document.getElementById("display").innerHTML = output;
  });
}
```

Però attenzione! Le callback:

- Non possono restituire valori alla funzione chiamante, ma solo eseguire azioni coi dati ottenuti.
- Sono funzioni indipendenti, e vengono eseguite alla fine dell'esecuzione della funzione che le chiama.
- Non hanno accesso alle variabili locali della funzione chiamante, ma solo a variabili globali e closure.

# Programmazione asincrona (9)

## Soluzione 2: codice asincrono e callback

- L'approccio delle callback è comunissimo, ma Javascript è *single thread*
- Anche quando il servizio è molto veloce, le funzioni callback vengono comunque eseguite alla fine del flusso di esecuzione del thread chiamante, e hanno un ambiente indipendente.
- Quindi i flussi asincroni vengono eseguiti sempre e solo alla fine dell'esecuzione del flusso principale.

# Programmazione asincrona (10)

## Soluzione 2: codice asincrono e callback

Con più query in sequenza la cosa si complica. Entriamo nel ***callback hell***

```
function searchBetterProducts(query) {
  let async = true;
  let orderDB = dbConnect("http://wwww.site.com","orders",acct,pwd);

  orderDB.search(query, async, (orders) => {
    let productDB = dbConnect("http://wwww.site.com","products",acct,pwd)

    productDB.search(orders, async, (products) => {
      let opinionDB = dbConnect("http://wwww.site.com","opinions",acct,pwd)

      opinionDB.search(products, async, (opinions) => {
        let output = prepareTable(orders,products,opinions);
        document.getElementById("display").innerHTML = output;
      }) ;
    });
  }) ;
}
```

# Programmazione asincrona (11)

## Soluzione 3: le promesse

Una promessa è un oggetto che, si promette, tra un po' conterrà un valore. La promessa è creata dalla funzione chiamante e mantenuta dalla funzione chiamata. Qui nella versione semplice con un solo database:

```
function searchSomeProducts(query) {
  promiseSearch('orders',query).then( (orders) => {
    let output = prepareTable(orders);
    document.getElementById("display").appendChild(output);
  })
}

function promiseSearch(db, query) {
  let async = true;
  return new Promise(function(resolve) {
    let DB = dbConnect("http://wwww.site.com",db,acct,pwd)
    let DB = DB.search(query, async, function(data) {
      resolve(data);
    }
  });
}
```

# Programmazione asincrona (12)

## Soluzione 3: le promesse

Grazie alle promesse semplifichiamo il callback hell nelle chiamate a catena:

```
function searchBetterProducts(query) {
  let orders, products, opinions ;
  promiseSearch('orders',query)
   .then( data => {
      orders = data;
      return promiseSearch('products',orders);
   }).then( data => {
      products = data;
      return promiseSearch('opinion',products)
   }).then( data => {
      opinions = data
      let output = prepareTable(orders,products,opinions);
      document.getElementById("display").innerHTML = output;
   })
}
function promiseSearch(dbName, query) {
  let async = true;
  return new Promise(function(resolve) {
    let DB = dbConnect("http://wwww.site.com",dbName, acct, pwd)
    let DB = DB.search(query, async, function(data) {
      resolve(data);
    }
```

# Programmazione asincrona (13)

## Soluzione 4: generator/yield

- Alcune librerie (iniziando con Q) avevano introdotto il concetto di generatore  e di yield. Questa è stata poi standardizzata in ES 2016.
- Il generatore è una metafunzione (una funzione che restituisce una funzione che può essere chiamata ripetutamente ed interrotta fino a che ne hai nuovamente bisogno).
- Ha una sintassi particolare con * dopo function.
- Il comando yield mette in attesa la assegnazione di valore fino a che non si chiude l'esecuzione della funzione chiamata.
- la funzione next() fa proseguire l'esecuzione della funzione fino al prossimo yield

# Programmazione asincrona (14)

## Soluzione 4: generator/yield

```
function *searchBetterProducts(query) {
  let async = false;
  let orderDB = dbConnect("http://wwww.site.com","orders",acct,pwd)
  let orders = yield orderDB.search(query, async, function(d) {
    generator.next(d)
  });
  let productDB = dbConnect("http://wwww.site.com","products",acct,pwd)
  let products = yield productDB.search(orders,async, function(d) {
    generator.next(d)
  });
  let opinionDB = dbConnect("http://wwww.site.com","opinions",acct,pwd)
  let opinions = yield opinionDB.search(products, async, function(d) {
    generator.next(d)
  });
  let output = prepareTable(orders, products, opinions);
  document.getElementById("display").innerHTML = output;
}

let generator = searchBetterProducts(query);
generator.next() ;
```

# Programmazione asincrona (15)

## Soluzione 4: generator/yield

Forse l'uso migliore di generator/yield è con gli iteratori, come gli array o le map o oggetti speciali su cui è definita la funzione next()

```
function *searchBetterProducts(query, databases) {
  let async = false;
  let q = [query] ;
  for (i in databases) {
    let db = dbConnect("http://wwww.site.com",databases[i],acct,pwd);
    let q[] = yield db.search(q.last(), async, (d) => {
        generator.next(d)
    });
  }
  let output = prepareTable(...q);
  document.getElementById("display").innerHTML = output;
}
let databases = ['orders','products','opinions'] ;
let generator = searchBetterProducts(query,databases);
generator.next() ;
```

In particolare, gli array sono iteratori di dimensione nota in partenza, ma ci sono iteratori di dimensione dinamica (ad esempio il contenuto di una directory server-side) per gli iteratori debbono essere flessibili e dinamici.

# Programmazione asincrona (16)

## Soluzione 5: async/await

Forse il modo più semplice per fare programmazione asincrona.

```
async function searchBetterProducts(query) {
  let async = true;
  let orderDB = dbConnect("http://wwww.site.com","orders",acct,pwd)
  let orders  = await orderDB.search(query, async);

  let productDB = dbConnect("http://wwww.site.com","products",acct,pwd)
  let products  = await productDB.search(orders, async) ;

  let opinionDB = dbConnect("http://wwww.site.com","opinion",acct,pwd)
  let opinions    = await opinionDB.search(products, async) ;

  let output = prepareTable(orders, products, opinions);
  document.getElementById("display").innerHTML = output;
}
```

La funzione async `searchProducts()` non viene eseguita più volte, ma sospesa fino alla conclusione della chiamata await, poi prosegue normalmente, con lo stesso stack, le stesse variabili locali e lo stesso contesto di esecuzione.

# Programmazione asincrona (17)

## Soluzione 6: Promise.all

Se ho molte chiamate asincrone indipendenti (cioè in cui non debbo aspettare il risultato di una per richiedere la seconda) posso usare Promise.all().
Nell'esempio seguente assumo che la seconda e la terza chiamata non dipendano dal risultato delle chiamate precedenti:

```
let p1 = new Promise(orderDB.search(query));
let p2 = new Promise(productDB.search(query));
let p3 = new Promise(opinionDB.search(query));

Promise.all([p1, p2, p3])
  .then( (values) => {
     let output = prepareTable(values[0],values[1],values[2]);
     document.getElementById("display").innerHTML = output;
  }
);
```

# Programmazione asincrona (17)

## Soluzione 6: Promise.all

Se ho molte chiamate asincrone indipendenti (cioè in cui non debbo aspettare il risultato di una per richiedere la seconda) posso usare Promise.all().
Nell'esempio seguente assumo che la seconda e la terza chiamata non dipendano dal risultato delle chiamate precedenti:

```
let p1 = new Promise(orderDB.search(query));
let p2 = new Promise(productDB.search(query));
let p3 = new Promise(opinionDB.search(query));

Promise.all([p1, p2, p3])
  .then( (values) => {
     let output = prepareTable(...values);
     document.getElementById("display").innerHTML = output;
  }
);
```

... usando l'operatore spread ...

# Programmazione asincrona (18)

## Soluzione 7: for await ... of ...

- Novità del 2023. Non ancora molto diffusa
- Mescola generatori, iteratori e await/async.

```
async function* searchBetterProducts(q,dbName) {
  let async = false;
  let db dbConnect("http://wwww.site.com",dbName,acct,pwd);
  yield db.search(q, async);
}

(async function () {
[ let databases = ['orders','products','opinions'];
  let q = [query] ;
  for await (db of databases) {
    q[] = searchBetterProducts(q.last(), db)
  }
  let output = prepareTable(...q);
  document.getElementById("display").innerHTML = output;
})();
```

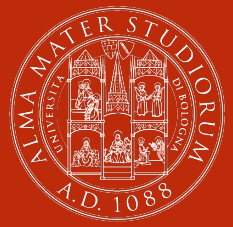
- Magari non lo vediamo, ma sappiate che c'è.

# Un piccolo esercizio

Andiamo a creare un sistema di connessioni parallele

Troviamo tutto a:

**https://www.fabiovitali.it/TW/2024/async/**



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA


**Fabio Vitali**

Dipartimento di Informatica – Scienze e Ingegneria
Alma mater – Università di Bologna

Fabio.vitali@unibo.it


www.unibo.it