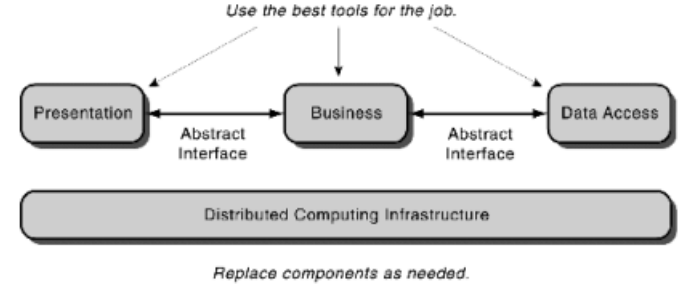
Web-app client dinamici - Ajax

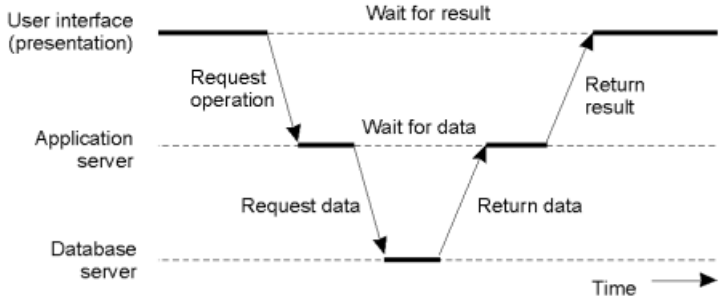
8.1 Applicazioni multi-tier

Abbiamo già introdotto molte **architetture multi-layer**, come ad esempio il *Middleware*, la *comunicazione RPC/RMI* oppure la *comunicazione basata su stream di byte*.

> **Important**
>
> Un'applicazione è strutturata in un **architettura multi-tier** quando è costituita da componenti che collaborano tra di loro per eseguire un task.
> Una tipica composizione è in tre livelli (**MCV**)
>
> 1. Presentation
> 2. Logic
> 3. Data
>
> 
>

8.1.1 Partizionamento delle task tra i layer

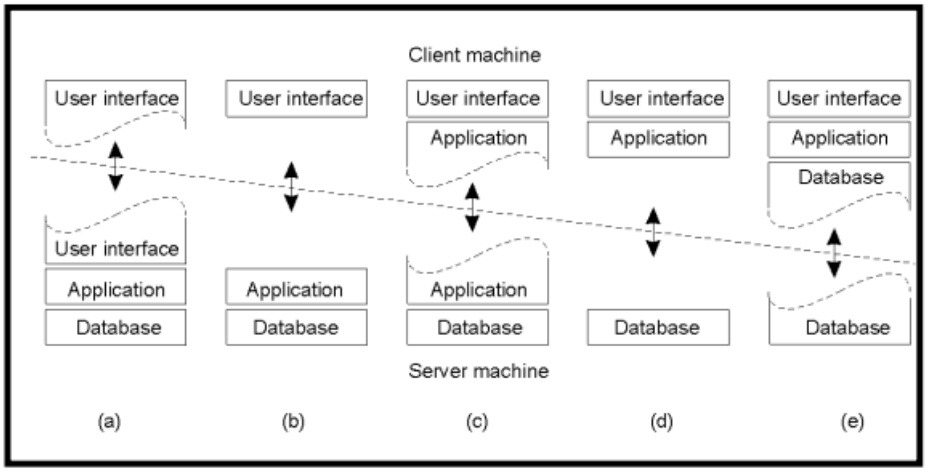
In un'architettura three-tier, il business tier funge da "server" per la presentation tier e da client per il data tier.



Questo genera un problema su come organizzare il sistema, per esempio:

- Come distribuire i componenti sulle macchine client/server?
- Cosa può essere implementato lato client e lato server?

Possono esserci diverse alternative su come suddividere i compiti tra client-server. Alcuni di questi possono essere:



8.2 Applicazioni RIA (Rich Internet Application)

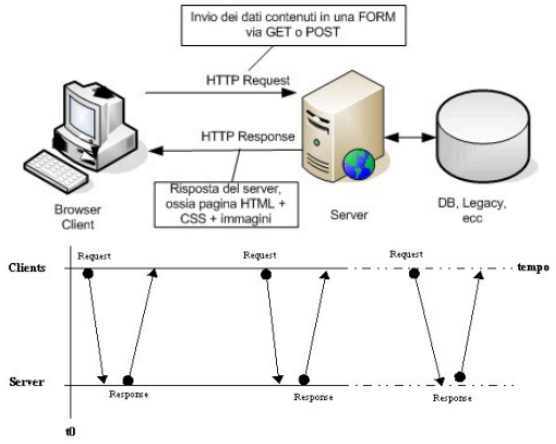
8.2.1 Storia del Web

Basis	Web 1.0	Web 2.0	Web 3.0
Functionality	Readable web	Writable web	Executable web
Type of data	Flat data	Interactive data	Immersive data
Basic idea	Connect information	Connect people	Connect knowledge
Interaction between web and user	Static	Interactive	Personalised
Example	Personal websites	Blogs, Social Networking	Sematic websites

8.2.2 Ajax (Asynchronous JavaScript and XML)

Caratteristiche del Web tradizionale

Le pagine web tradizionali si ricaricano completamente ogni volta che l'utente richiede nuovi dati o compie un'azione. Non esiste un aggiornamento parziale dei contenuti. Gli utenti, quindi, devono attendere che l'intera pagina venga caricata, anche se hanno bisogno solo di una piccola porzione di dati. Questo rende l'esperienza utente inefficiente e lenta. Infine, le interazioni tra client e server sono limitate a un ciclo di richiesta e risposta, che limita la dinamicità e l'interattività delle applicazioni web.



Ecco perchè nasce Ajax

Ajax è una tecnica di sviluppo che consente la creazione di applicazioni web interattive. Permette l'aggiornamento di parti della pagina web senza doverla ricaricare completamente.

Ajax non è una nuova tecnologia, ma un pattern di sviluppo che combina diverse tecnologie esistenti per migliorare l'interattività delle applicazioni web.

- **Supporto alle Interfacce Utente**: Ajax facilita la creazione di interfacce utente più reattive e dinamiche.
- **Webpage meno leggibili/linkabili**: Le pagine web costruite con Ajax possono risultare meno leggibili dalle macchine e meno linkabili rispetto alle pagine web tradizionali.

Core feature del Web 2.0

- **Web come Piattaforma**: Il Web 2.0 vede il web come una piattaforma su cui costruire applicazioni e servizi, piuttosto che un semplice insieme di pagine statiche.
- **Intelligenza Collettiva**: Sfrutta la partecipazione degli utenti per creare valore, come avviene nei social network e nei siti di condivisione di contenuti.
- **Oltre il Livello del Singolo Dispositivo**: Le applicazioni Web 2.0 sono progettate per funzionare su una varietà di dispositivi, non limitandosi al solo computer desktop.
- **Servizi, non Software Confezionato**: L'attenzione è rivolta alla fornitura di servizi online piuttosto che alla distribuzione di software confezionato.
- **Esperienze Utente Ricche**: Le applicazioni Web 2.0 offrono esperienze utente più ricche e coinvolgenti, spesso utilizzando tecniche come Ajax per migliorare l'interattività e la reattività.

8.2.2.1 Componenti di Ajax

Ajax è composto da diverse componenti chiave che lavorano insieme per creare applicazioni web interattive e dinamiche.

1. **HTML (o XHTML) e CSS** vengono utilizzati per presentare le informazioni in modo strutturato e stilizzato.
2. Il **Document Object Model (DOM)** consente la visualizzazione dinamica e l'interazione con i contenuti della pagina.
3. Per lo scambio e la manipolazione dei dati, si utilizzano **XML** e **XSLT**, sebbene nelle applicazioni moderne **JSON** sia spesso preferito per la sua leggerezza e facilità d'uso.
4. L'oggetto **XMLHttpRequest** è fondamentale per recuperare i dati in modo *asincrono* dal server web, evitando il ricaricamento completo della pagina.
5. **JavaScript** lega insieme tutte queste componenti, facilitando l'interazione tra di esse, con **TypeScript** come alternativa fortemente tipizzata per migliorare la robustezza del codice.

JavaScript

JavaScript è un linguaggio di scripting a oggetti, non tipizzato. Viene interpretato da un *engine* e viene eseguito dal browser (Node.js permette l'esecuzione sul server).
JavaScript permette di rendere le pagine html dinamiche, cioè di inserire dei programmi che modificano il comportamento e le visualizzazioni.
Obiettivi principali:

1. La capacità di effettuare richieste in HTTP al server, in maniera trasparente all'utente
2. La funzione di rendere asincrona la comunicazione tra browser e web server.

JavaScript può richiedere dati in formato testo puro e XML. Attualmente il formato più diffuso è JSON.

8.2.2.1 Architettura Ajax

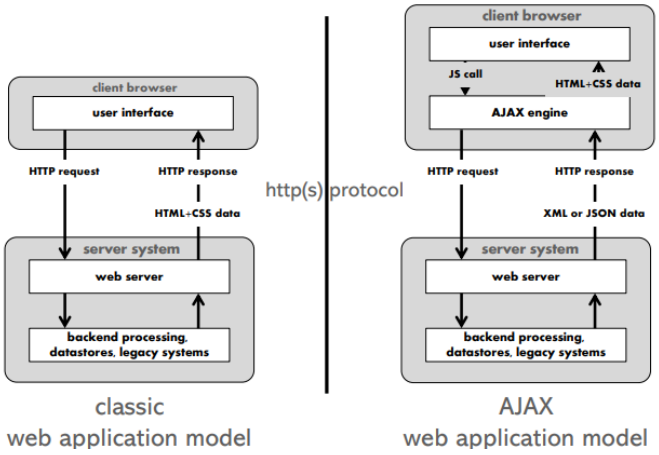
Nel modello classico delle applicazioni web:

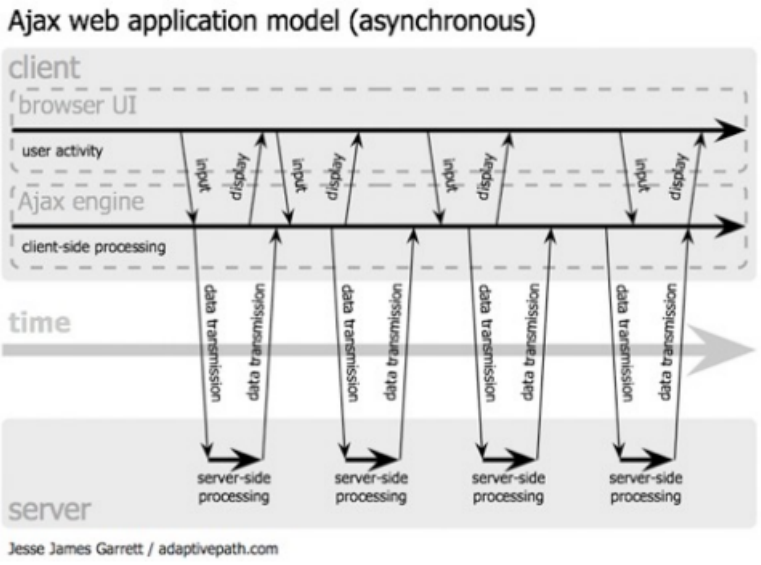
1. **Richiesta completa della pagina**: Ogni volta che l'utente interagisce con l'applicazione (ad esempio, cliccando su un link o inviando un modulo), il browser invia una richiesta HTTP al server.
2. **Ricaricamento della pagina**: Il server elabora la richiesta, esegue eventuali operazioni necessarie (come accesso a un database), genera una nuova pagina HTML e la invia al browser.

3. **Esperienza dell'utente**: L'intera pagina viene ricaricata, il che può causare interruzioni nell'esperienza dell'utente a causa dei tempi di caricamento e del fatto che la pagina scompare e riappare completamente.

Nel modello AJAX:

1. **Richiesta asincrona**: Le interazioni dell'utente non causano un ricaricamento completo della pagina. Invece, il JavaScript esegue richieste asincrone (AJAX) al server dietro le quinte.
2. **Aggiornamento parziale della pagina**: Il server risponde con dati (spesso in formato JSON o XML) e il JavaScript sul client aggiorna dinamicamente solo le parti necessarie della pagina web senza ricaricare l'intera pagina.
3. **Esperienza dell'utente**: L'utente percepisce un'esperienza più fluida e reattiva, poiché gli aggiornamenti possono avvenire in modo incrementale e senza interruzioni.





Esempio pratico

Immagina un modulo di ricerca su un sito di e-commerce.

Modello classico	Modello AJAX
Quando l'utente invia il modulo, il server elabora la ricerca e restituisce una nuova pagina con i risultati. Questo comporta un ricaricamento completo della pagina.	L'invio del modulo di ricerca comporterebbe una richiesta asincrona al server per ottenere i risultati. Una volta ricevuti, solo la sezione della pagina contenente i risultati di ricerca verrebbe aggiornata, senza ricaricare l'intera pagina.

Problemi di AJAX
Rompere il pulsante "indietro"

- I browser registrano le visite alle pagine statiche

- Gli IFrames invisibili possono invocare modifiche che popolano la cronologia
  Modifica inaspettata di parti della pagina
- Dovrebbe avvenire solo in luoghi ben definiti
  Segnare un particolare "stato" diventa difficile
- JavaScript genera la pagina, non il server
  Aumenta la dimensione del codice sul browser
- Il tempo di risposta ne risente
  Difficile eseguire il debug
- Logica di elaborazione sia nel client che nel server
  Fonte visualizzabile
- Aperto agli hacker o al plagio
  Carico del server
- La richiesta asincrona può essere un'operazione "pesante

La trasmissione dei dati tra server e applicazione RIA è un aspetto critico.
Esistono diverse alternative tecnologiche: SOAP, XML-RPC, JSON, AMF etc. e la scelta del formato influenza la struttura dell'applicazione e le performance.
Questo perché un formato richiedere tempo per predisporre i dati lato server, trasferire i dati, fare il parsing dei dati lato client, fare il rendering dell'interfaccia.

Ad esempio: è meglio JSON per JavaScript poiché ha delle istruzioni built-in per il parsing di oggetti JSON.

Struttura della pagina

```
<!DOCTYPE HTML>
<html>
<body>
        HTML include:
        1. Contenuto da visualizzare
        2. Interfaccia utente
        3. Spazio di input/output

        <script>
        Javascript ha lo scopo da host per il codice.
        NOTA: potrebbe essere inserito nell'head, ma è buona
                pratica includere tutto alla fine del body per
                evitare comportamenti imprevisti dovuti
                all'elaborazione sequenziale della pagina.
        </script>
</body>
</html>
```

Come vengono visualizzati i risultati?
JavaScript non ha nessun metodo built-in per stampare o mostrare funzioni.

JavaScript può mostrare i risultati in alcuni modi:

1. **Scrittura in un elemento HTML usando innerHTML**: Questo metodo consente di inserire del testo o HTML all'interno di un elemento HTML esistente.
   Ad esempio, il codice mostrato sostituisce il contenuto di un elemento `<p>` con l'ID "demo" con il testo "Hello world!".

```
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = "Hello world!";
</script>
```

2. **Scrittura in una finestra di avviso usando window.alert()**: Questo metodo apre una finestra di avviso nel browser con il testo specificato. Ad esempio, il codice mostrato visualizza una finestra di avviso con il testo "Hello world!".

```
window.alert("Hello world!")
```

3. **Scrittura nella console del browser usando console.log()**: Questo metodo scrive messaggi di debug nella console del browser. È utile per il controllo dell'esecuzione del codice e per visualizzare informazioni durante lo sviluppo. Ad esempio, il codice mostrato scrive un messaggio nella console che indica il numero di valori letti.

```
console.log("Letti i primi " + count + " valori.");
```

4. **Scrittura nell'output HTML usando document.write()**: Questo metodo sostituisce l'intero contenuto della pagina HTML con il testo o HTML specificato. Tuttavia, è importante notare che l'uso di `document.write()` è generalmente sconsigliato perché può sovrascrivere l'intera pagina e causare problemi di rendering e di caricamento.

Controllo e input di dati

Invocazione di funzioni

Il codice all'interno di una funzione verrà eseguito quando "qualcosa" invoca (chiama) la funzione:

- Quando si verifica un **evento** (quando un utente clicca un pulsante)
- Quando viene **invocata** (chiamata) dal codice JavaScript
- **Automaticamente** (auto-invocata)

Eventi JavaScript

Gli eventi JavaScript sono "azioni" che accadono agli elementi HTML.

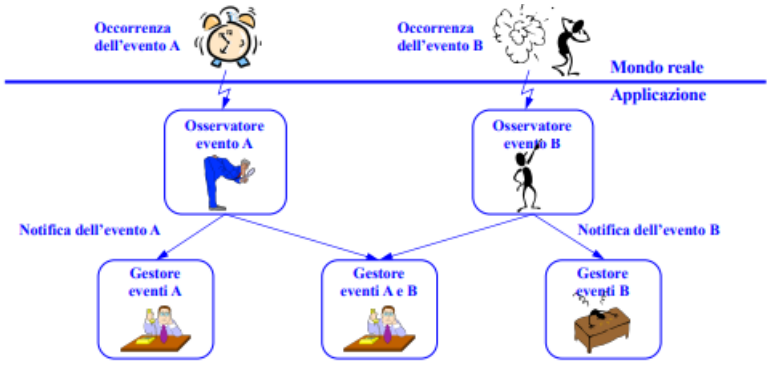
- Quando JavaScript viene utilizzato nelle pagine HTML, JavaScript può "reagire" a questi eventi.

```
<some-HTML-element some-event="some JavaScript">
```

Gli eventi JavaScript giocano un ruolo fondamentale nel facilitare l'interazione dinamica e reattiva tra l'utente, la pagina HTML e il codice JavaScript nelle applicazioni web Ajax.

L'applicazione è puramente "reattiva". *Non è possibile identificare staticamente un flusso di controllo unitario*.

- Il programma principale (main) si limita a inizializzare l'applicazione, istanziando gli osservatori e associandovi gli opportuni handler (gestori);
- L'associazione osservatore-gestore può essere dinamica.



AddEventListener

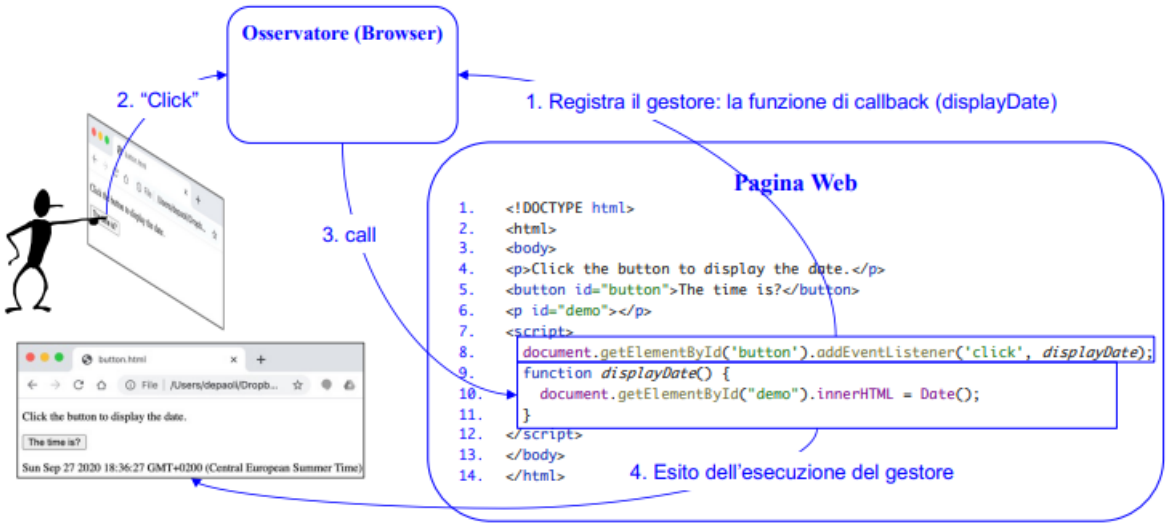
Si può associare uno o più gestori di eventi ad ogni elemento del DOM HTML che genera eventi. Viene utilizzata la

funzione `AddEventListener`.
Questo metodo permette di aggiungere più listener per lo stesso evento su un singolo elemento, e offre un controllo maggiore sugli eventi rispetto all'uso delle tradizionali proprietà degli eventi come `onclick`.

```
element.addEventListener(event, function);
```

1. Il primo parametro è il tipo di evento: "click" o "mousedown" o qualsiasi evento del DOM
   NB: omettere il prefisso "on" ("click" e non "onclick")
2. Il secondo parametro è la funzione che deve essere chiamata quando l'evento accade
   NB: può essere una funzione anonima o il nome di una funzione (senza parentesi rotonde)

Gestione degli eventi con AddEventListener

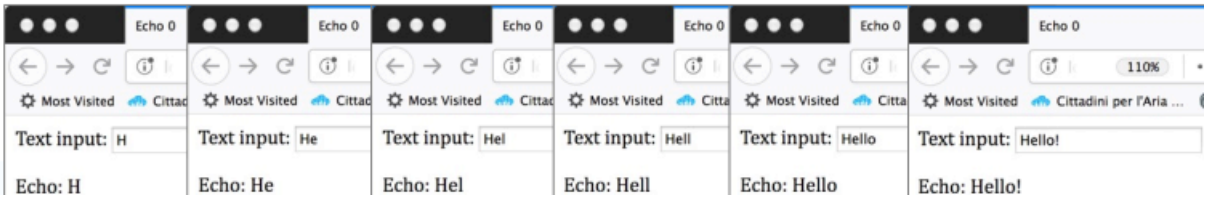


Passaggio di dati con function calls

```
<html>
<body>
Text input:<input type="text" id="txt1" onkeyup="echo1(this.value)">
<p>Echo: <span id="demo"></span></p>

<script>
function echo1(str) {
        document.getElementById("demo").innerHTML = str;
}
</script>
</body>
</html>
```

Ad ogni evento release-button, chiamata la funzione chiamata echo1 passando this.value. Il risultato diventa



Un'alternativa può essere:

```
<html>
<title> Echo 1 </title>
<body>
<form action="">
```

```
        Text input: <input type="text" id="txt1" onkeyup="echo()">
</form>
<p>Echo: <span id:"demo"></span></p>

<script>
        function echo() {
                document.getElementById("demo").innerHTML = document.getElementById("txt1").value;
        }
</script>
</body>
</html>
```

8.2.3 JavaScript

8.2.3.1 Dichiarazione delle variabili

- Le variabili sono dichiarate **var** e valgono per tutto il programma (global scope) o nelle funzioni (function scope)
- La parola riservata **let** permette di definire variabili che valgono solo nel blocco in cui sono dichiarate (block scope)
- La parola riservata **const** permette di definire variabili let con un valore costante (cioè che non può essere modificato) (block scope)
  ATTENZIONE: le const sono contenitori costanti, ma i valori contenuti possono cambiare
  • Le variabili **semplici** sono **immodificabili**
  • I valori delle **variabili** composte possono **cambiare**

Buona regola: usare var solo per le variabili globali, let e const in tutti gli altri casi

8.2.4 Interazione con il server

8.2.4.1 loadDoc()

Questa funzione tipicamente utilizza l'oggetto `XMLHttpRequest` o, più recentemente, l'API Fetch per inviare richieste HTTP asincrone e aggiornare dinamicamente il contenuto di una pagina web senza ricaricarla completamente.

```
<!DOCTYPE html>
<html>
<head>
  <title>Esempio di loadDoc con XMLHttpRequest</title>
</head>
<body>

<button type="button" onclick="loadDoc()">Carica Documento</button>
<div id="demo"></div>

<script>
function loadDoc() {
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
        // Utilizza onreadystatechange per segnalare un nuovo stato
        // del ciclo di scrittura/lettura che il programma può conoscere
        // attraverso la variabile readyState
    if (this.readyState == 4 && this.status == 200) {
      document.getElementById("demo").innerHTML = xhttp.responseText;
    }
  };
  // Apre la connessione con la connect() e crea la prima riga
  // di richiesta (obbligatoria) in formato HTTP
```

```
  // Può popolare l'header del messaggio con funzioni dedicate
  // NOTA: header di default vengono aggiunti automaticamente
  xhttp.open("GET", "ajax_text.txt", true);
  // Chiude il messaggio, lo invia utilizzando write(), e avvia
  // la lettura della risposta con read()
  // NOTA: invia senza body perché è una GET
  xhttp.send();
}
</script>

</body>
</html>
```

Quando si fa clic sul pulsante, viene richiamata la funzione JavaScript loadDoc() viene chiamata e l'effetto è quello di sostituire il testo nel div "demo".

La funzione **loadDoc()** svolge le seguenti task:

1. Caricare il file ajax_text.txt
2. Se la nuova risorsa viene caricata correttamente (codice di stato codice 200)
3. Il testo ricevuto viene visualizzato come contenuto del div "demo"

La condizione `this.readyState == 4 && this.status == 200` significa che la richiesta è completata (`readyState == 4`) e la risposta dal server è stata ricevuta con successo (`status == 200`).

L'oggetto **XMLHttpRequest** viene utilizzato per:

- Aggiornare una pagina web senza ricaricarla
- Richiedere dati da un server - dopo il caricamento della pagina
- Ricevere dati da un server - dopo il caricamento della pagina
- Inviare dati a un server - in background

Creazione di un oggetto XMLHttpRequest:
variable = new XMLHttpRequest();
Sintassi per mandare una richiesta:
xhttp.open("GET", "ajax_text.txt", true);
xhttp.send();

Metodo	Parametri	Cosa fa
`open(method, url, async)`	- `method` : Specifica il metodo HTTP da usare per la richiesta. In questo caso, "GET" è utilizzato per recuperare dati. - `url` : Specifica l'URL del file o risorsa che si vuole recuperare. In questo caso, "ajax_text.txt" è il file che vogliamo ottenere dal server. - `async` : Indica se la richiesta deve essere asincrona (`true`) o sincrona (`false`). Usare `true` (asincrono) è generalmente consigliato per non bloccare l'interfaccia utente mentre la richiesta è in corso.	La chiamata `xhttp.open("GET", "ajax_text.txt", true);` prepara l'oggetto `XMLHttpRequest` per inviare una richiesta GET asincrona al file "ajax_text.txt".
`xhttp.send()`		Questo metodo invia la richiesta HTTP preparata con il metodo `open` . In caso di una richiesta GET, non è necessario includere alcun corpo nel metodo `send()` ; quindi, viene chiamato senza argomenti.

La condizione `this.readyState` è utilizzata nel contesto dell'oggetto `XMLHttpRequest` in JavaScript per determinare lo stato corrente della richiesta HTTP.

8.2.4.2 Risposte dal server

Proprietà	Descrizione
`onreadystatechange`	Definisce una funzione da richiamare quando la proprietà readyState cambia
`readyState`	Mantiene lo stato della richiesta XMLHttpRequest. **0 (UNSENT)**: La richiesta non è stata ancora inizializzata. **1 (OPENED)**: La richiesta è stata aperta, ma non è stata ancora inviata. **2 (HEADERS_RECEIVED)**: I header della richiesta sono stati ricevuti. **3 (LOADING)**: Il corpo della risposta è in fase di ricezione. **4 (DONE)**: La richiesta è completata e la risposta è pronta.
`status`	**200**: "OK" **403**: "Forbidden" **404**: "Page not found" Ce ne sono altri.
`statusText`	Ritorna la status-text (e.g. "OK" or "Not Found")

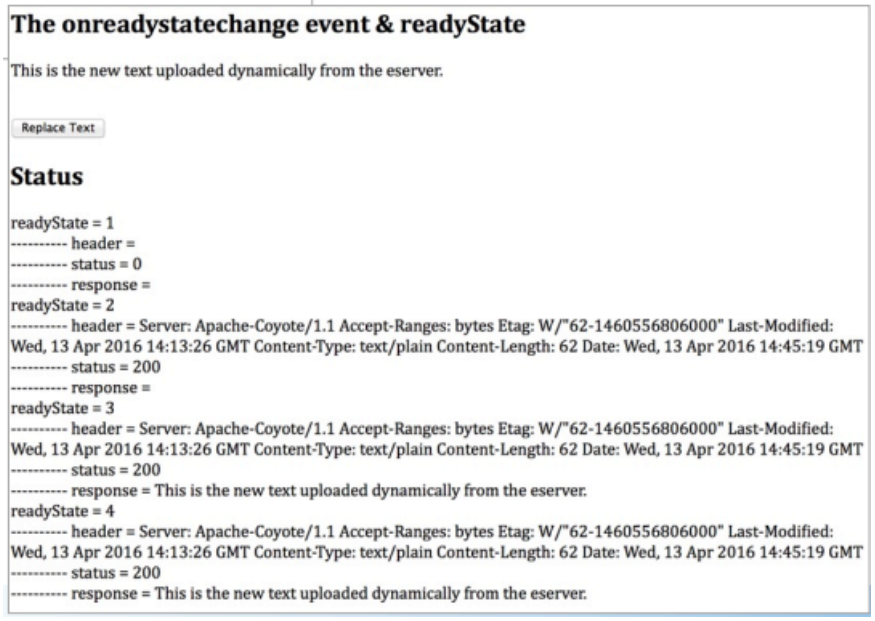
```
<!DOCTYPE html>
<html>
<head>
  <title>Esempio XMLHttpRequest Dettagliato</title>
</head>
<body>

<button type="button" onclick="loadDoc()">Carica Documento</button>
<div id="status"></div>
<div id="demo"></div>

<script>
function loadDoc() {
  var xhttp = new XMLHttpRequest();
  var s = "<h2>Status</h2>";
  xhttp.onreadystatechange = function() {
    s = s + "readyState = " + xhttp.readyState;
    s = s + "<br> ---------- header = " + xhttp.getAllResponseHeaders();
    s = s + "<br> ---------- status = " + xhttp.status;
    s = s + "<br> ---------- response = " + xhttp.responseText;
    s = s + "<br>";
    if (xhttp.readyState == 4 && xhttp.status == 200) {
      document.getElementById("demo").innerHTML = xhttp.responseText;
    }
    document.getElementById("status").innerHTML = s;
  };
  xhttp.open("GET", "ajax_text.txt", true);
  xhttp.send();
}
</script>

</body>
</html>
```

Una volta che la readyState sarà di valore 4, in status verrà stampato questo:



responseText

Per ottenere la risposta da un server, utilizzare la proprietà responseText o responseXML dell'oggetto XMLHttpRequest.

Proprietà	Descrizione
responseText	La response data è una string
responseXML	La response data è un XML data

Funzione di callback

Una funzione di callback è una funzione passata come parametro a un'altra funzione.
Se avete più di un task AJAX sul vostro sito web, dovreste creare UNA funzione standard per creare l'oggetto XMLHttpRequest e chiamarla per ogni task AJAX.
La chiamata alla funzione deve contenere l'URL e cosa fare al momento del cambio di stato (che probabilmente è diverso per ogni chiamata).

```
function loadDoc(cFunc) {
// The actual parameter will be a function to process the
// data received from the server
        const xhttp = new XMLHttpRequest();
        xhttp.onreadystatechange = function() {
                if (xhttp.readyState == 4 && xhttp.status == 200) {
                // Call the actual function to process the data answer
                        cFunc(xhttp);
                }
        };
        xhttp.open("GET", "ajax_text.txt", true);
        xhttp.send();
}
```

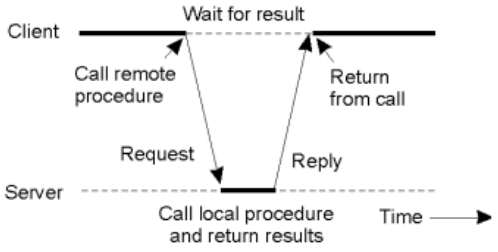
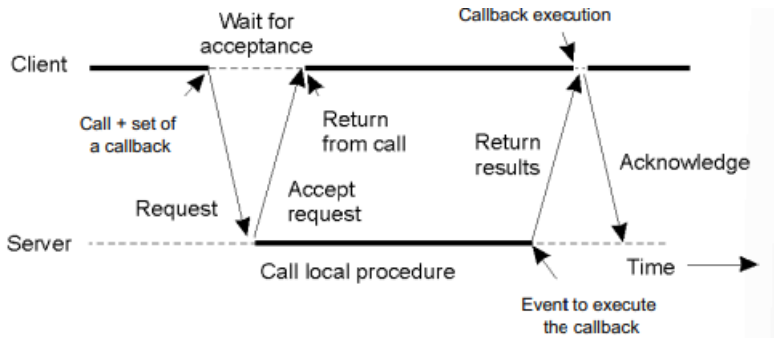
8.2.5 Programmazione lato server

Node.js è una piattaforma che permette di realizzare applicazioni web veloci e scalabili. Usa un modello di I/O (input e output) non bloccante e ad eventi.

Approccio asincrono

Node.js usa un modello ad eventi e un sistema di I/O non bloccante che lo rende leggero ed efficiente.
Sincrono: aspetto che l'effetto di una funzione sia completato prima di eseguire la prossima
Asincrono:

- non aspetto che l'effetto di una funzione sia completato prima di eseguire la prossima
- creo una funzione (di callback) che completi l'effetto desiderato mentre proseguo l'esecuzione delle istruzioni successive

Esecuzione sincrona	Esecuzione asincrona
- Il client attende (si blocca) la risposta dal chiamato - Prosegue solo quando ha ottenuto il risultato richiesto 	- Il client non attende (non si blocca) la risposta dal chiamato - Quando arriva la risposta si genera un evento e viene eseguita la callback (se definita) 

Modello ad eventi

L'esecuzione dei programmi in Node si basa su un **Single Event Loop**, che preleva un evento da una singola coda e lo serve eseguendo le operazioni previste (esecuzione callback, accesso a file, alla rete, ...).
Il **Single Event Loop** è **logico**, cioè le operazioni sono eseguite logicamente in sequenza.
A livello fisico ogni operazione è eseguita da un thread (flusso di controllo o di esecuzione) autonomo che permette esecuzione parallela e concorrente.
Questo modello • si base sull'esecuzione di operazioni stateless; • disaccoppia la programmazione dall'esecuzione favorendo la scalabilità

1. Singolo thread
2. Loop di eventi: esecuzione di eventi temporizzati se scaduti o di eventi in testa alla coda
3. Eventi aggiunti da azioni dell'utente, risposte del server, eventi generici

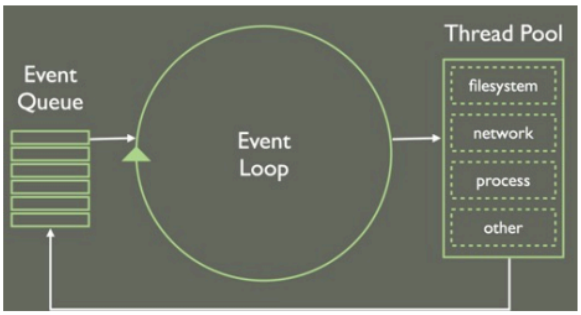
Concetti chiave:

- Ogni evento viene eseguito fino al completamento prima dell'evento successivo.
- La risposta alla richiesta Ajax non è sempre immediata.

Vantaggi	Svantaggi
Non è necessario preoccuparsi della mutua esclusione	- Un evento di lunga durata blocca l'interfaccia utente - I timeout possono arrivare in ritardo - Quando arriva la risposta Ajax, il contesto potrebbe essere cambiato

Esempio

1. *App*: Inserisce nella coda la richiesta (evento) «leggi file ed esegui callback»
2. *Loop*: Estrae la richiesta (evento – leggi file)
3. *Loop*: Assegna a un esecutore (thread) la richiesta e la callback
4. *Esecutore*: Esegue il compito «leggi file» e al termine inserisce la richiesta (evento) di callback
5. *Loop*: Estrae la richiesta (evento – callback)
6. *Loop*: Assegna a un esecutore (thread) la callback
7. *App*: Percepisce l'effetto della esecuzione della richiesta originale



Il modulo express.js è un framework web che fornisce gli strumenti per realizzare un server che può ospitare sia risorse statiche, sia applicazioni che generano rappresentazioni dinamiche

7.2.5.1 Struttura di un applicazione MVC

- *bin*: Il file all'interno di bin, chiamato www, è il file di configurazione principale della nostra applicazione.
- *public*: La cartella public contiene i file che devono essere resi pubblici per l'uso, come i file JavaScript, i file CSS, le immagini ecc.
- *routes*: La cartella routes contiene i file che contengono i metodi di navigazione verso le diverse aree della mappa. Contiene vari file js.
- *views*: La cartella view contiene vari file che costituiscono la parte di vista dell'applicazione.
  Esempio: la homepage, la pagina di registrazione, ecc.
- *app.js*: il file app.js è il file principale, che costituisce la testa di tutti gli altri file. Qui devono essere "richiesti" i vari pacchetti installati. Oltre a questo, serve per molti altri come la gestione di router, middleware, ecc.
- *package.json*: il file package.json è il file manifest di qualsiasi progetto Node.js e applicazione express.js. Contiene i metadati del progetto, come i pacchetti e le loro versioni utilizzati nell'applicazione (chiamati dipendenze), vari script come start e test (eseguiti da terminale come 'npm start'), il nome dell'applicazione, la descrizione dell'applicazione, la versione dell'applicazione, ecc.