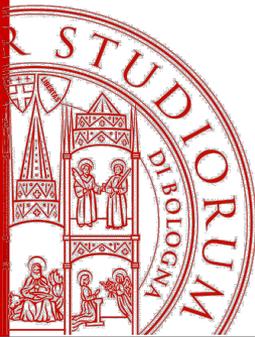


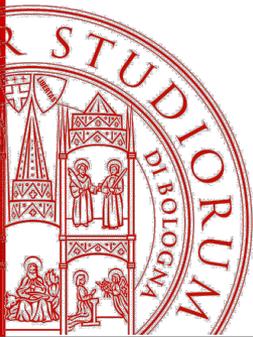
# HyperText Transfer Protocol

*Angelo Di Iorio*  
*Università di Bologna*

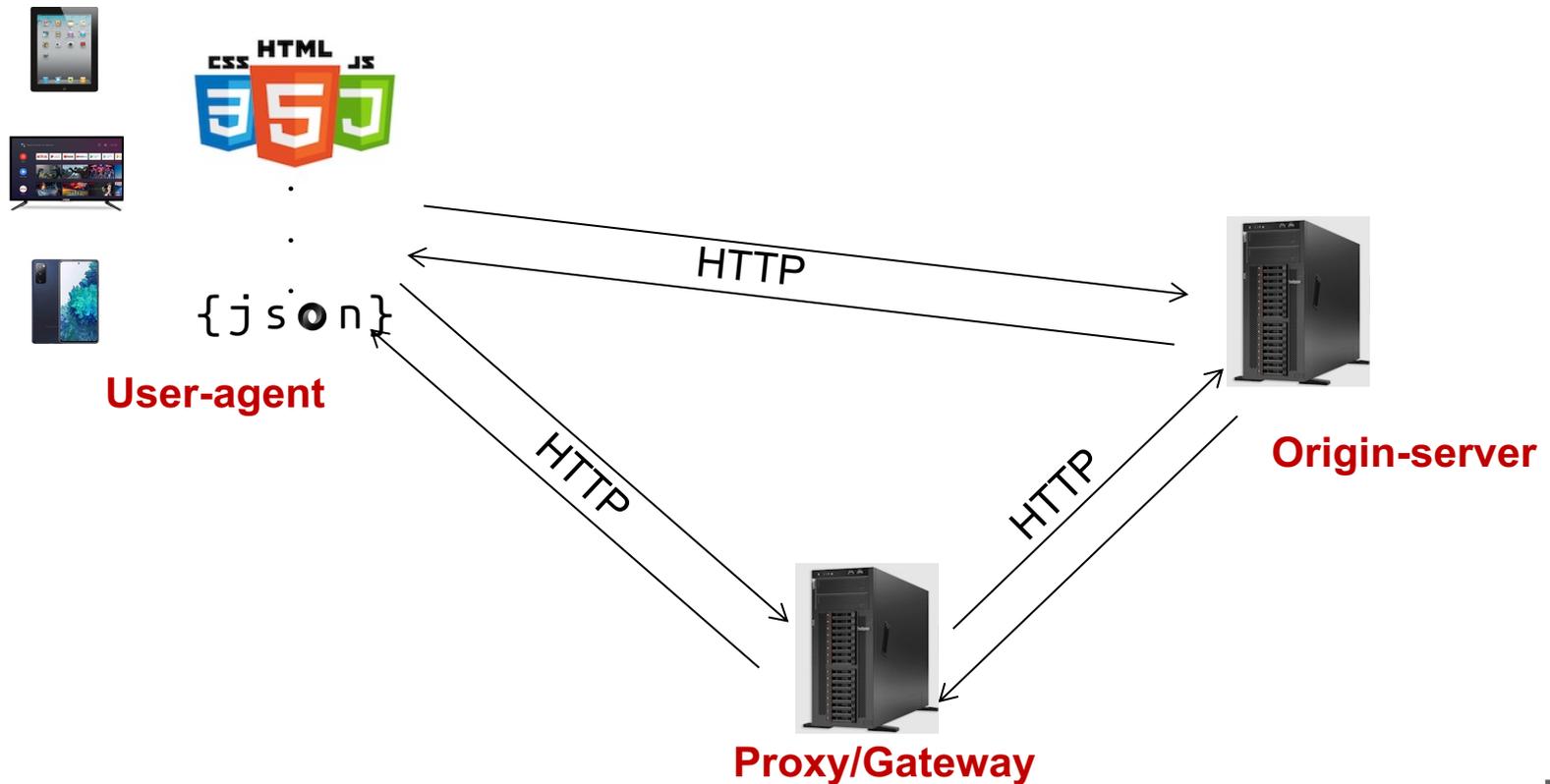


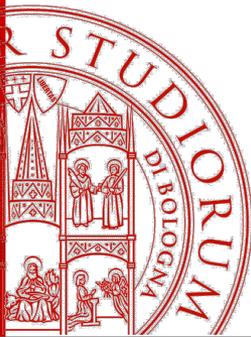
# HyperText Transfer Protocol

- HTTP é un protocollo **client-server**, **generico** e **stateless** utilizzato non solo per lo scambio di documenti Web ma in molte applicazioni distribuite
  - **Client-server**: il *client* attiva la connessione e richiede dei servizi. Il server accetta la connessione, nel caso identifica il richiedente, e risponde alla richiesta. Alla fine chiude la connessione.
  - **Generico**: HTTP è indipendente dal formato dati con cui vengono trasmesse le risorse. Può funzionare per documenti HTML come per binari, eseguibili, oggetti distribuiti o altre strutture dati più o meno complicate.
  - **Stateless**: Il server non è tenuto a mantenere informazioni che persistano tra una connessione e la successiva sulla natura, identità e precedenti richieste di un client. Il client è tenuto a ricreare da zero il contesto necessario al server per rispondere.



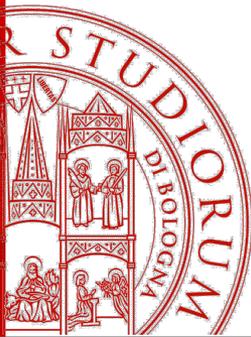
# Ruoli in HTTP (1)





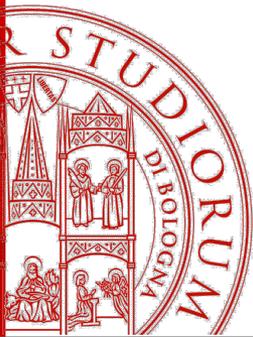
# Ruoli in HTTP (2)

- **Client:** un'applicazione che stabilisce una connessione HTTP, con lo scopo di mandare richieste.
- **Server:** un'applicazione che accetta connessioni HTTP, e genera risposte.
- **User agent:** client che inizia una richiesta HTTP (tipicamente un browser, ma può anche essere un bot).
- **Origin server:** il server che possiede fisicamente la risorsa richiesta (è l'ultimo della catena)
- **Proxy:** Un'applicazione intermediaria che agisce sia da client che da server. Le richieste sono soddisfatte autonomamente o con possibili trasformazioni o controlli
- **Gateway:** un'applicazione che agisce da intermediario per qualche altro server. A differenza della comunicazione con il proxy, un client può non essere al corrente che si tratta del gateway

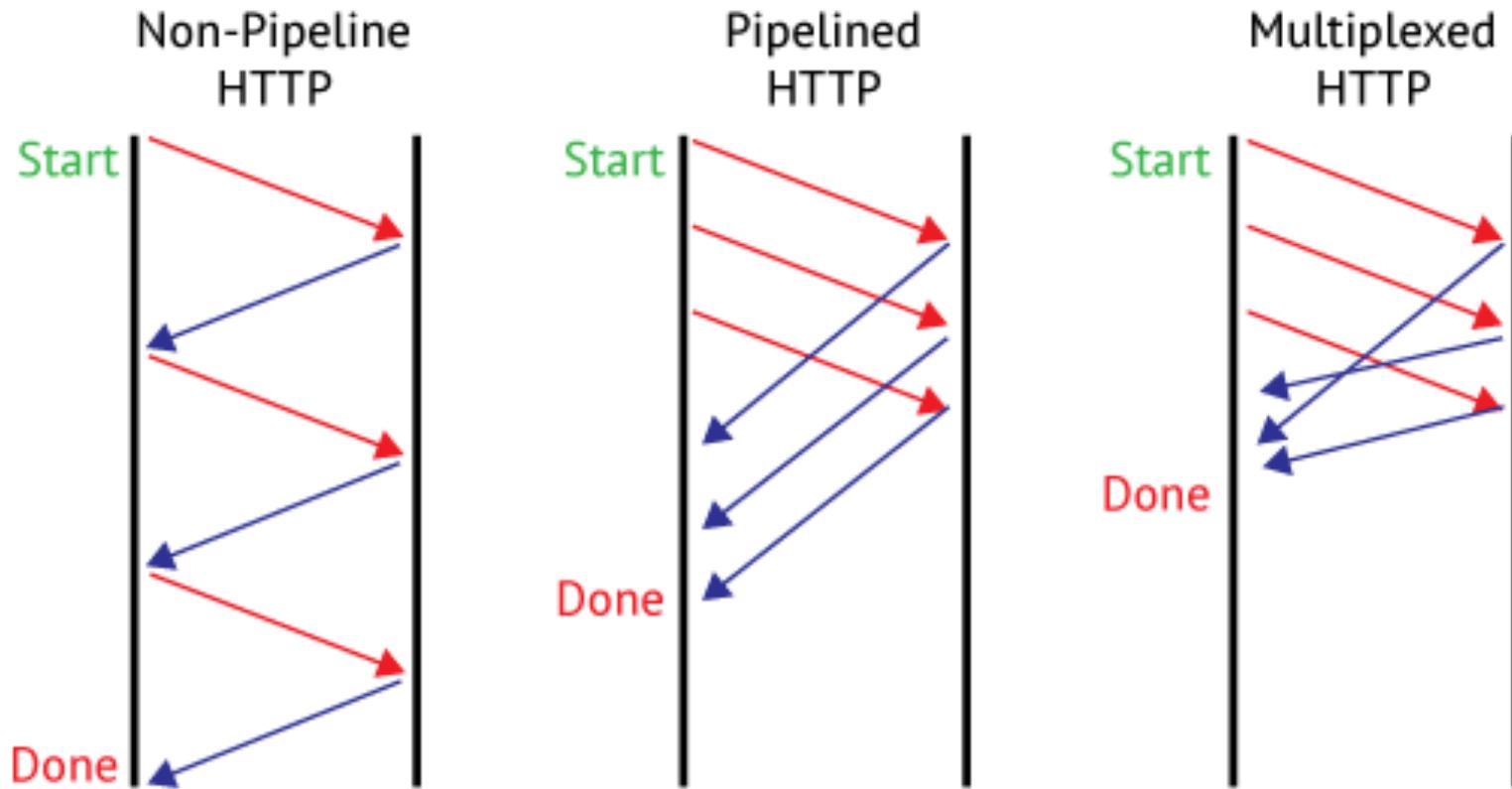


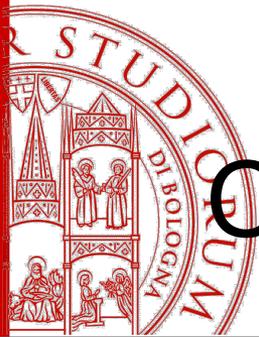
# Connessione HTTP

- Una connessione HTTP è composta da una serie di richieste ed una serie corrispondente di risposte
- Le connessioni sono **persistenti** con:
  - **Pipelining**: trasmissione di più richieste senza attendere l'arrivo della risposta alle richieste precedenti. MA le risposte sono restituite nello stesso ordine delle richieste
  - **Multiplexing**: nella stessa connessione è possibile avere richieste e risposte multiple, restituite anche in ordine diverso rispetto alle richieste e “ricostruite” nel client
- HTTP/2 ha introdotto molte novità per migliorare le performance, tra cui:
  - **Operazioni PUSH**: il server anticipa le richieste successive del client
  - **Compressione degli header**: i dati non sono trasmessi in chiaro ma compressi (e in parallelo)



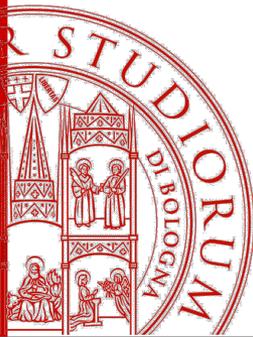
# Connessioni HTTP



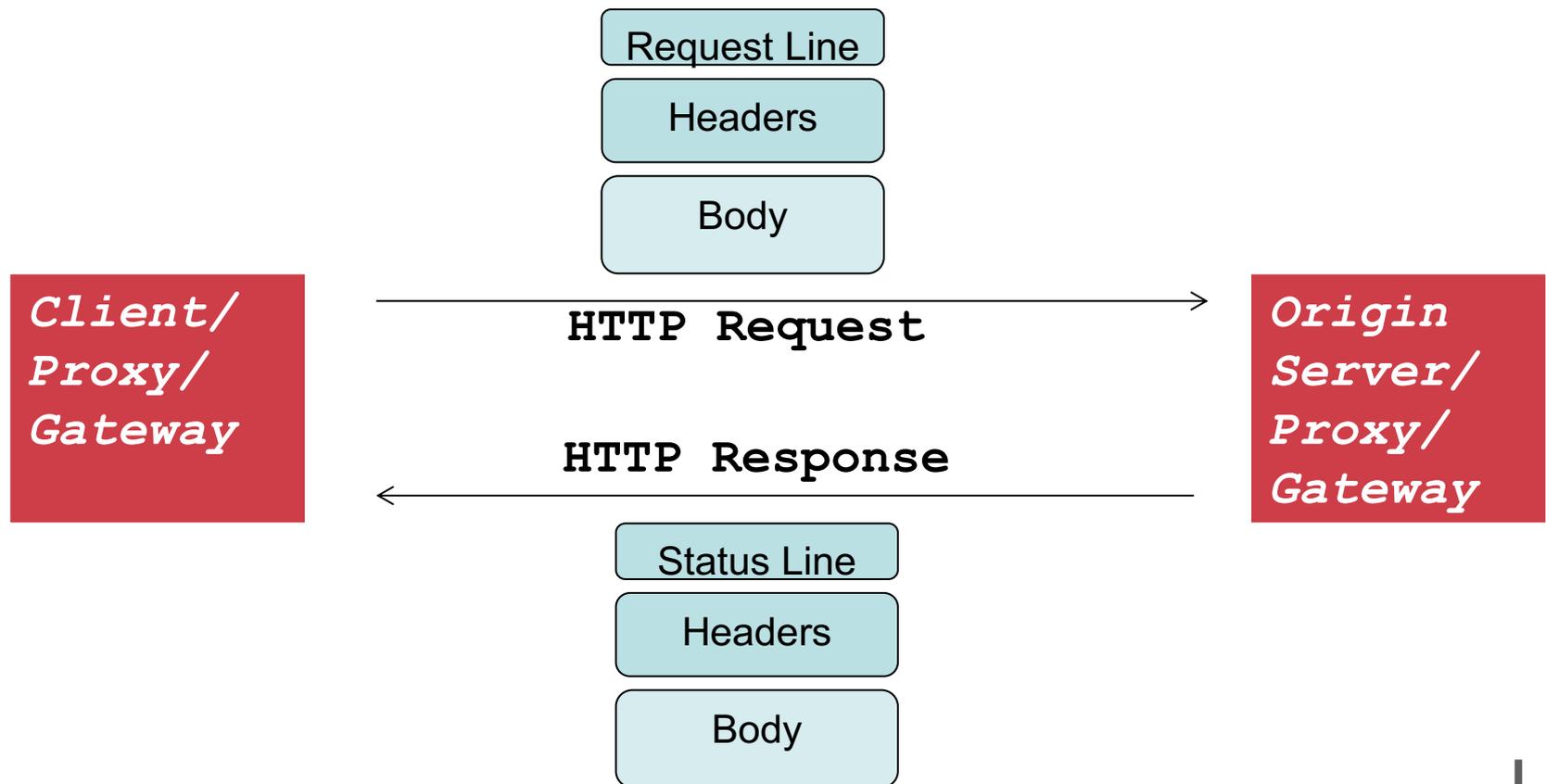


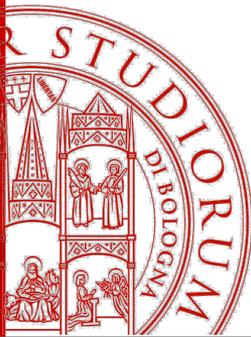
# Concetto chiave: risorse HTTP

- HTTP permette lo scambio di **risorse** identificate da URI
- Separa nettamente le risorse dalla loro **rappresentazione** e fornisce meccanismi di **negoziamento del formato di dati**, cioè la possibilità di richiedere e ricevere (la rappresentazione di) una stessa risorsa in formati diversi;
- HTTP implementa inoltre sofisticate **politiche di caching** che permettono di memorizzare **copie delle risorse sui server (proxy, gateway, etc.)** coinvolti nella trasmissione e controllare in modo accurata la validità di queste copie
- Un uso corretto di questi meccanismi migliora notevolmente le performance delle applicazioni



# Richieste e risposte HTTP

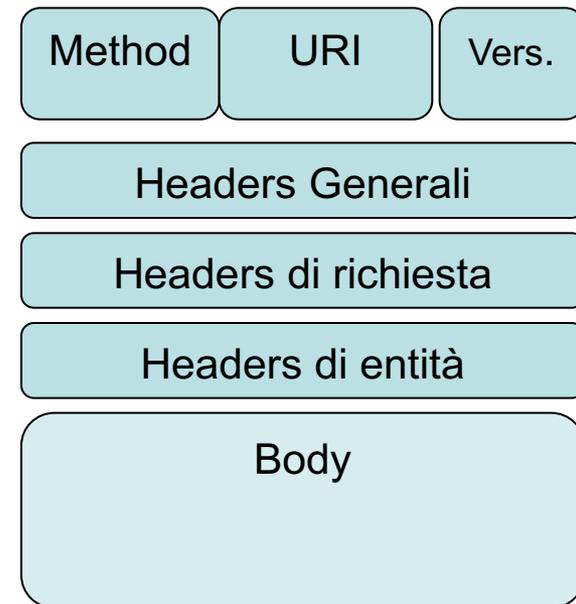


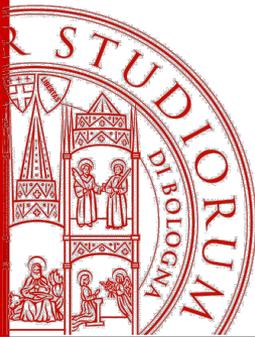


# La richiesta HTTP

La richiesta HTTP si compone di:

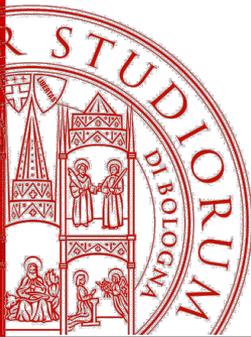
- **Method:** azione del server richiesta dal client
- **URI:** identificativo della risorsa locale al server
- **Version:** numero di versione di HTTP
- **Header** sono linee RFC822 divise in:
  - header generali
  - header di entità
  - header di richiesta
- **Body** è un messaggio MIME





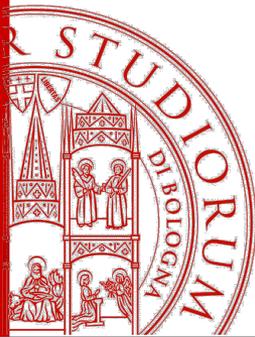
# Un esempio di richiesta

```
GET /beta.html HTTP/1.1
Referer: http://www.alpha.com/alpha.html
Connection: Keep-Alive
User-Agent: Mozilla/4.61 (Macintosh; I; PPC)
Host: www.alpha.com:80
Accept: image/gif, image/jpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1, *, utf-8
```



# I metodi di HTTP

- I metodi indicano l'azione che il client richiede al server sulla risorsa, o meglio sulla rappresentazione della risorsa o, ancora meglio, sulla copia della rappresentazione della risorsa
- Chiamati anche **verbi HTTP** per evidenziare l'idea che esprimono azioni da eseguire sulle risorse, identificate a loro volta da nomi (URI)
- Un uso corretto dei metodi HTTP aiuta a creare applicazioni interoperabili e in grado di sfruttare al meglio i meccanismi di caching di HTTP
- I metodi principali: GET, HEAD, POST, PUT, DELETE, OPTIONS, PATCH
- Guardiamo esempi dei più usati, GET e POST, e poi torneremo sui metodi a fine lezione visto la loro importanza per REST



# Esempi di GET e POST

- GET è il metodo più frequente, ed è quello che viene attivato facendo click su un link ipertestuale di un documento HTML, o specificando un URL nell'apposito campo di un browser.

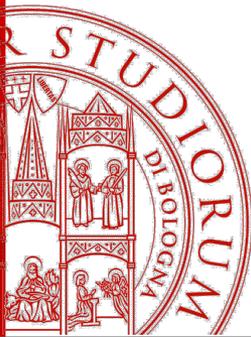
```
GET /courses/tw.html
```

```
GET /students/123456/exams/
```

- Il metodo POST serve per trasmettere informazioni dal client al server relative alla risorsa identificata nell'URI

```
POST /courses/1678
```

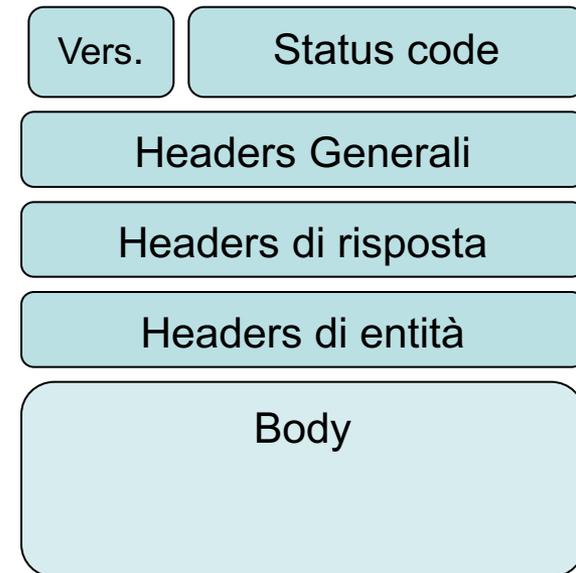
```
{  
  "titolo": "Tecnologie Web",  
  "descrizione": "Il corso..bla..bla.."  
}
```

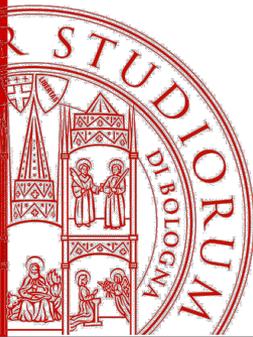


# La risposta HTTP

La risposta HTTP si compone di:

- **Status code:** indica se la richiesta è andata a buon fine o meno
- **Version:** numero di versione di HTTP
- **Header:** come per la richiesta:
  - header generali
  - header di entità
  - header di risposta
- **Body** è un messaggio MIME





# Esempio di risposta

```
GET /index.html HTTP/1.1  
Host: www.cs.unibo.it:80
```

```
HTTP/1.1 200 OK
```

```
Date: Fri, 26 Nov 2007 11:46:53 GMT
```

```
Server: Apache/1.3.3 (Unix)
```

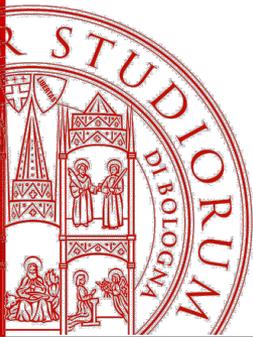
```
Last-Modified: Mon, 12 Jul 2007 12:55:37 GMT
```

```
Accept-Ranges: bytes
```

```
Content-Length: 3357
```

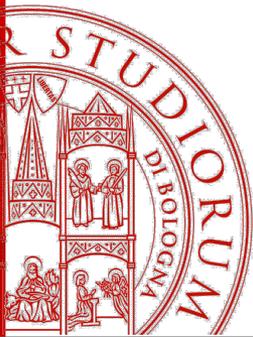
```
Content-Type: text/html
```

```
<HTML> ... </HTML>
```



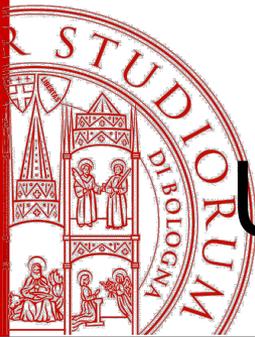
# Status code

- Lo status code è un numero di tre cifre, di cui la prima indica la classe della risposta, e le altre due la risposta specifica.
- Esistono le seguenti classi:
  - **1xx: Informational.** Una risposta temporanea alla richiesta, durante il suo svolgimento.
  - **2xx: Successful.** Il server ha ricevuto, capito e accettato la richiesta.
  - **3xx: Redirection.** La richiesta è corretta, ma sono necessarie altre azioni da parte del client per portare a termine la richiesta.
  - **4xx: Client error.** La richiesta del client non può essere soddisfatta per un errore da parte del client (errore sintattico o richiesta non autorizzata).
  - **5xx: Server error.** La richiesta può anche essere corretta, ma il server non riesce a soddisfarla per un problema interno



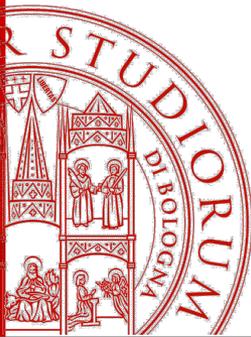
# Esempi di status code

- 100 Continue** (se il client non ha ancora mandato il body)
- 200 Ok** (GET con successo)
- 201 Created** (PUT con successo)
- 301 Moved permanently** (URL non valida, il server conosce la nuova posizione)
- 400 Bad request** (errore sintattico nella richiesta)
- 401 Unauthorized** (manca l'autorizzazione)
- 403 Forbidden** (richiesta non autorizzabile)
- 404 Not found** (URL errato)
- 500 Internal server error** (tipicamente un errore nel codice server-side)



# Utilità dello status code HTTP

- L'uso corretto degli status code aiuta a costruire API chiare e più semplici da usare
- Il client non ha necessità di leggere il body della risposta ma già dallo status code può capire se la richiesta è andata a buon fine
  - Può trovare dettagli nel body ma la dipendenza con il server è ridotta
- Permette a tutte le entità coinvolte nella comunicazione di capire meglio cosa succede, e sfruttare i meccanismi di caching e redirectione di HTTP
- Migliora uniformità e interoperabilità



# Gli header

- Gli header sono righe di testo (RFC822) che specificano informazioni aggiuntive
- Sono presenti sia nelle richieste che nelle risposte e ne descrivono diversi aspetti

**Header Generali**  
Informazioni sulla  
trasmissione

**Header di entità**  
Informazioni sulla risorsa  
e i dati trasmessi

**Header di richiesta**  
Informazioni sulla  
richiesta effettuata

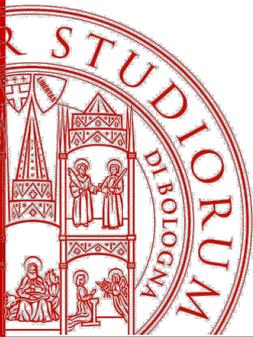
**Richiesta**

**Header Generali**  
Informazioni sulla  
trasmissione

**Header di entità**  
Informazioni sulla risorsa  
e i dati trasmessi

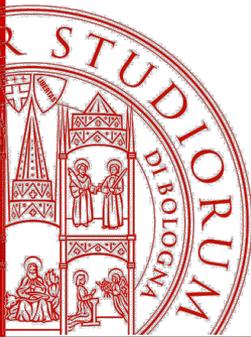
**Header di risposta**  
Informazioni sulla risposta  
generata

**Risposta**



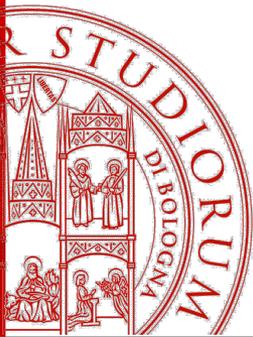
# Header generali

- Gli header generali si applicano solo al messaggio trasmesso e si applicano sia ad una richiesta che ad una risposta, ma non necessariamente alla risorsa trasmessa.
- Alcuni esempi:
  - **Date**: data ed ora della trasmissione
  - **Transfer-Encoding**: il tipo di formato di codifica usato per la trasmissione
  - **Cache-Control**: il tipo di meccanismo di caching richiesto o suggerito per la risorsa
  - **Connection**: il tipo di connessione da usare (tenere attiva, chiudere dopo la risposta, ecc.)



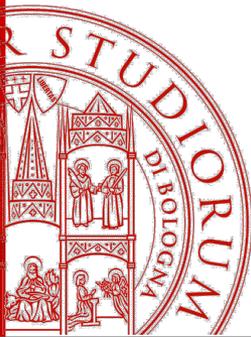
# Header dell'entità

- Gli header dell'entità danno informazioni sul body del messaggio, o, se non vi è body, sulla risorsa specificata.
- Alcuni esempi:
  - **Content-Type**: il tipo MIME dell'entità nel body. Questo header è obbligatorio in ogni messaggio che abbia un body.
  - **Content-Length**: la lunghezza in byte del body. Obbligatorio, soprattutto se la connessione è persistente.
  - **Content-Encoding, Content-Language, Content-Location, Content-MD5, Content-Range**: la codifica, il linguaggio, l'URL della risorsa specifica, il valore di digest MD5 e il range richiesto della risorsa.



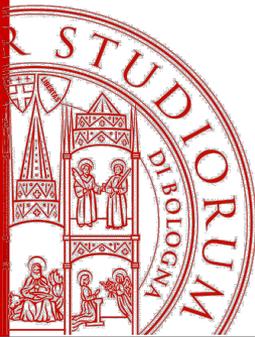
# Header della richiesta

- Gli header della richiesta sono posti dal client per specificare informazioni sulla richiesta e su se stesso al server.
- Esempi
  - **User-Agent:**
    - una stringa che descrive il client che origina la richiesta; tipo, versione e sistema operativo del client, tipicamente.
  - **Referer:**
    - l'URL della pagina mostrata all'utente mentre richiede il nuovo URL.
  - **Host:**
    - il nome di dominio e la porta a cui viene fatta la connessione.
- Altri header della richiesta sono usati per gestire la cache e i meccanismi di autenticazione



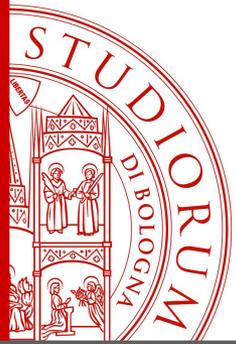
# Header della risposta

- Gli header della risposta sono posti dal server per specificare informazioni sulla risposta e su se stesso al client
- Alcuni esempi :
  - **Server:**
    - una stringa che descrive il server: tipo, sistema operativo e versione.
  - **WWW-Authenticate:**
    - challenge utilizzata per i meccanismi di autenticazione

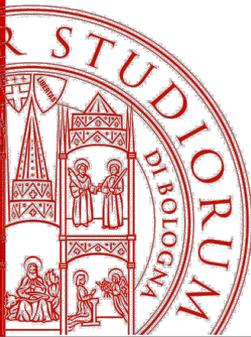


# Importanza del content-type

- Tra gli header della risposta sono particolarmente utili quelli che definiscono il tipo di dato contenuto nella risposta
- Queste informazioni permettono al client di processare correttamente la (rappresentazione di una) risorsa
- Se viene fornita un'entità in risposta, infatti, gli header `Content-type` e `Content-length` sono obbligatori
  - E' solo grazie al **content type** che lo user agent sa come visualizzare l'oggetto ricevuto
  - E' solo grazie al **content length** che lo user agent sa che ha ricevuto tutto l'oggetto richiesto.

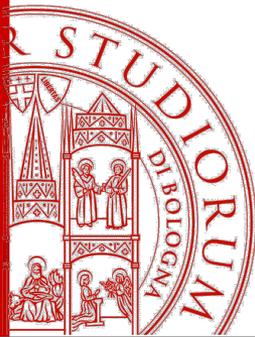


# Metodi HTTP



# I metodi di HTTP

- I metodi indicano l'azione che il client richiede al server sulla risorsa, o meglio sulla rappresentazione della risorsa o, ancora meglio, sulla copia della rappresentazione della risorsa
- Chiamati anche **verbi HTTP** per evidenziare l'idea che esprimono azioni da eseguire sulle risorse, identificate a loro volta da nomi (URI)
- Un uso corretto dei metodi HTTP aiuta a creare applicazioni interoperabili e in grado di sfruttare al meglio i meccanismi di caching di HTTP
- I metodi principali: GET, HEAD, POST, PUT, DELETE, OPTIONS, PATCH
- Guardiamo esempi dei più usati, GET e POST, e poi torneremo sui metodi a fine lezione visto la loro importanza per REST



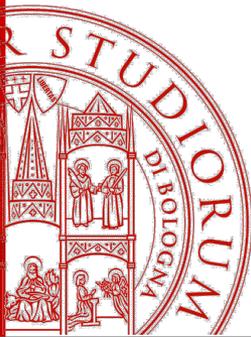
# Due proprietà importanti

- **Sicurezza**

- Niente a che vedere con password, hacking e privacy.
- Un metodo è sicuro se non genera cambiamenti allo stato interno del server (a parte ovviamente nei log).
- Un metodo sicuro può essere eseguito da un nodo intermedio (es. una cache) senza effetti negativi.

- **Idempotenza:**

- Un metodo è idempotente se l'effetto sul server di più richieste identiche è lo stesso di quello di una sola richiesta (a parte ovviamente i log).
- Un metodo idempotente può essere ri-eseguito da più agenti o in più tempi diversi senza effetti negativi.



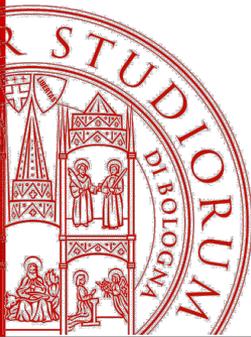
# Il metodo GET

- Il più importante (ed unico in v. 0.9) metodo di HTTP è GET, che richiede una risorsa ad un server.
- Questo è il metodo più frequente, ed è quello che viene attivato facendo click su un link ipertestuale di un documento HTML, o specificando un URL nell'apposito campo di un browser.
- *GET è sicuro ed idempotente*

**GET** /courses/tw.html

**GET** /students/123456

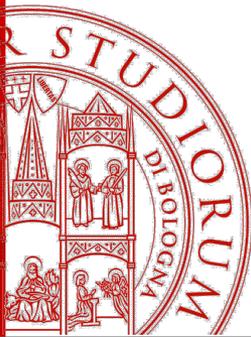
**GET** /students/123456/exams/



# Il metodo HEAD

- Il metodo HEAD è simile al metodo GET, ma il server deve rispondere soltanto con gli header relativi, senza il corpo.
- Viene usato per verificare validità, accessibilità e coerenza in cache di un URI
- *HEAD è sicuro e idempotente*

`HEAD /courses/tw.html`

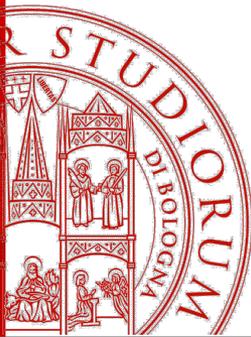


# Il metodo POST

- Il metodo POST serve per trasmettere informazioni dal client al server relative alla risorsa identificata nell'URI
- Può essere usato anche per creare nuove risorse
- Viene usato per esempio per spedire i dati di un form HTML ad un'applicazione server-side
- *POST non è sicuro né idempotente*

`POST /courses/1678`

```
{  
  "titolo": "Tecnologie Web",  
  "descrizione": "Il corso..bla..bla.."  
}
```

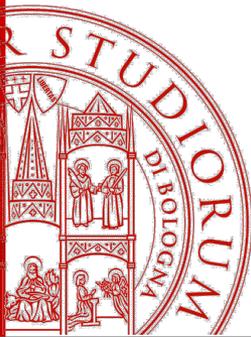


# Il metodo PUT

- Il metodo PUT serve per trasmettere delle informazioni dal client al server, creando o sostituendo la risorsa specificata.
- In generale, l'argomento del metodo PUT è la risorsa che ci si aspetta di ottenere facendo un GET in seguito con lo stesso nome
- Non offre garanzie di controllo degli accessi o locking
- *PUT è idempotente ma non sicuro*

**PUT /courses/1678**

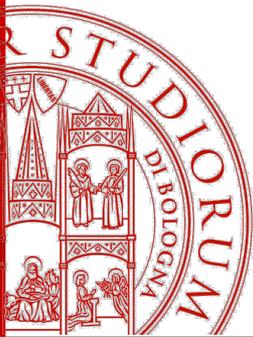
```
{  
  "id":1678,  
  "titolo":"Tecnologie Web",  
}
```



# Il metodo DELETE

- Il metodo DELETE serve per rimuovere le informazioni connesse con una risorsa.
- Dopo l'esecuzione di un DELETE, la risorsa non esiste più e ogni successiva richiesta di GET risulterà in un errore 404 Not Found
- Il metodo DELETE su una risorsa già non esistente è lecito e non genera un errore.
- *DELETE è idempotente e non sicuro.*

**DELETE /courses/1678**

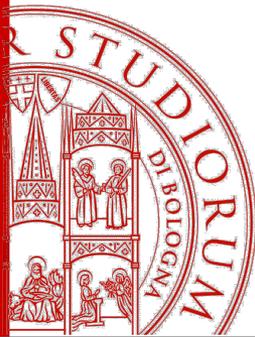


# Il metodo PATCH

- Il metodo PATCH è usato per aggiornare parzialmente la risorsa identificata dall'URI
- Usato per indicare quindi modifiche incrementali e non sovrascrivere risorse al server, come ad esempio nel caso di PUT. Indica quindi le modifiche da effettuare su una risorsa esistente
- *PATCH non è né sicuro né idempotente*

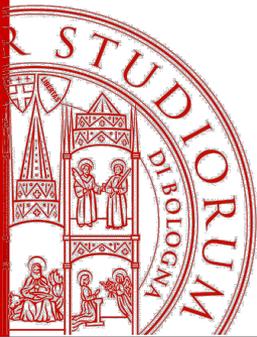
```
PATCH /courses/1678
```

```
{  
  "op" : "update"  
  "cfu" : "6"  
}
```



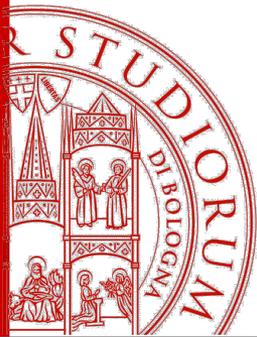
# Il metodo OPTIONS

- OPTIONS viene usato per verificare opzioni, requisiti e servizi di un server, senza implicare che seguirà poi una altra richiesta.
- Usato per il problema del *cross-site scripting*: CORS, ci torneremo parlando di AJAX



# Riassumendo

HTTP Methods	IDEMPOTENT	SAFE METHOD
GET	YES	YES
HEAD	YES	YES
OPTION	YES	YES
DELETE	YES	NO
PUT	YES	NO
PATCH	NO	NO
POST	NO	NO



# Conclusioni

- Oggi abbiamo visto le caratteristiche principali di HTTP
- In particolare l'uso corretto di metodi in ottica REST