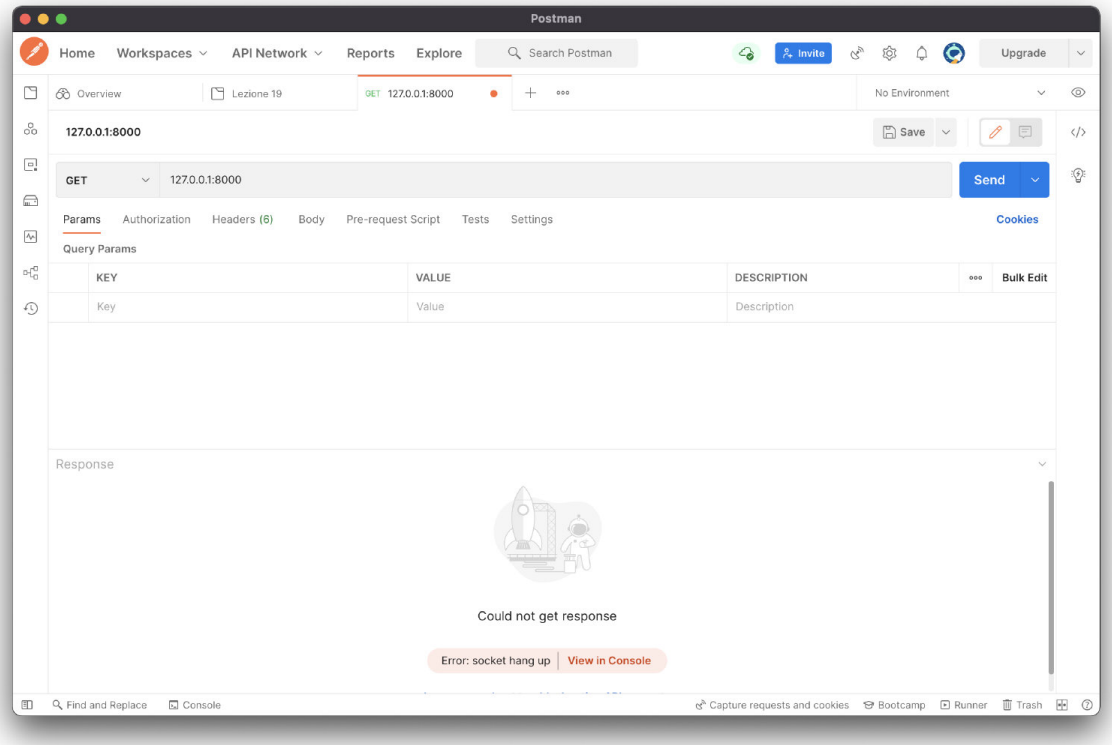
Lezione 19 - Richieste HTTP nella pratica

Abbiamo scritto un semplice server che ci permette di accettare delle richieste HTTP da un client, che può essere un browser o un agente come **postman**.

Ci basta inviare una richiesta GET su postman:



Il server non risponde alla richiesta, ma stampa in stdout delle informazioni:

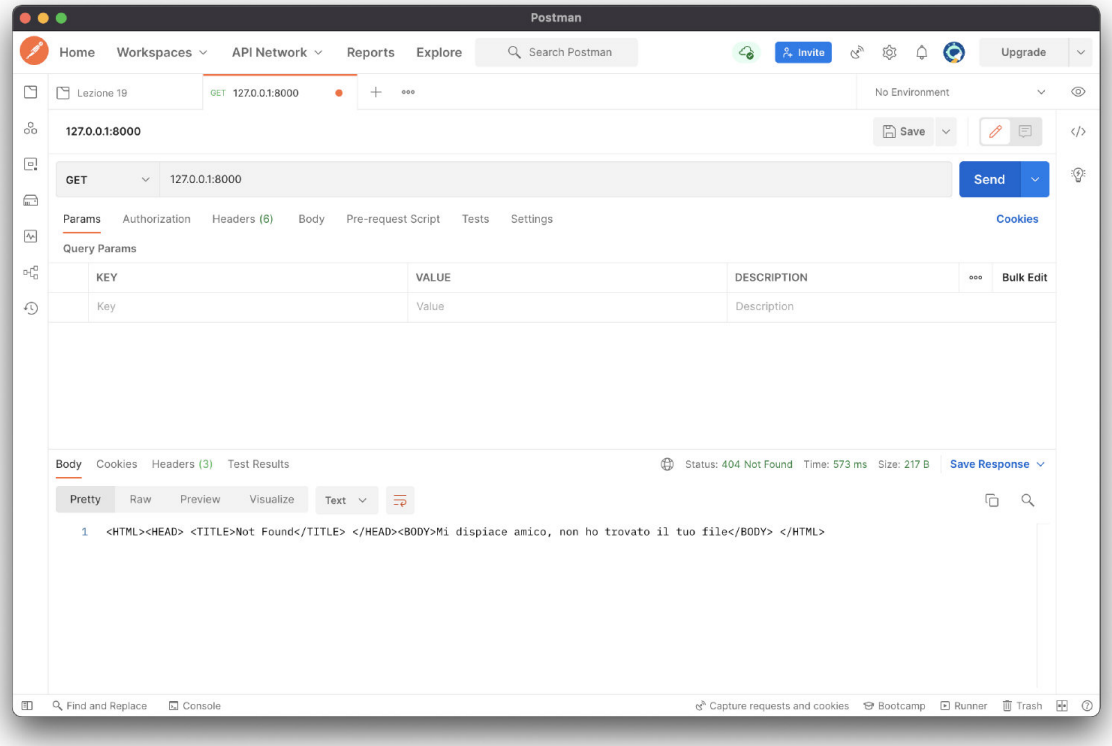
```
Richiesta ricevuta da un browser.
Client: /127.0.0.1
Thu Feb 03 19:04:47 CET 2022
Request line: GET / HTTP/1.1
User-Agent: PostmanRuntime/7.29.0
Accept: */*
Postman-Token: 926c7ff3-5714-4598-b96a-8edd11d8c343
Host: 127.0.0.1:8000
Accept-Encoding: gzip, deflate, br
Connection: keep-alive

Request ended.
Closing socket with host /127.0.0.1
```

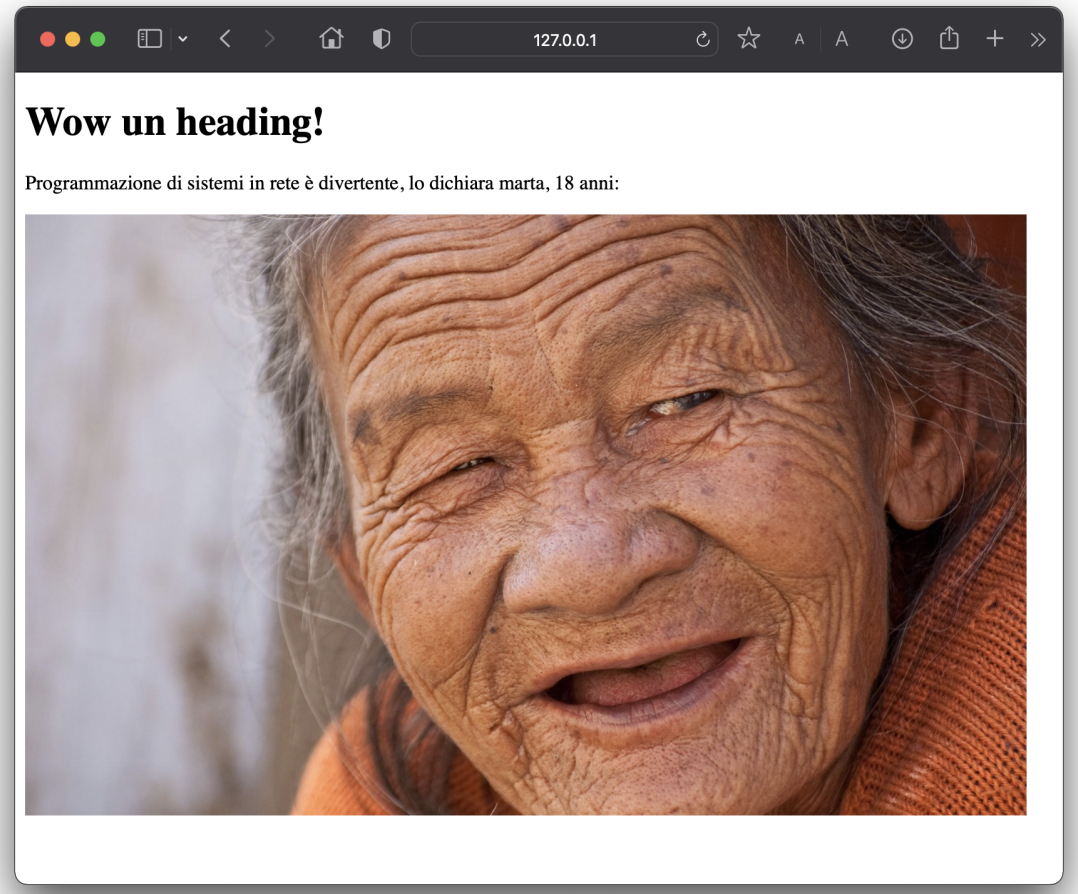
Nel secondo esercizio visto, più articolato, abbiamo un file webserver.conf dove vengono riportate le informazioni riguardanti la porta e la root:

```
port 8000
root ./root
```

Questo server prevede una risposta; nel primo caso non prevediamo alcun file da richiedere, quindi la risposta sarà:



Se però richiediamo una risorsa con `http://127.0.0.1:8000/index.html`, ed usiamo un browser, otteniamo:



Migliorare le performance HTTP

Abbiamo osservato come recuperare una pagina web significa non solo recuperare l'involucro ma anche il contenuto (infatti qualora avessimoa avuto delle immagini locali il browser avrebbe fatto delle richieste GET per delle specifiche risorse, per poterle renderizzare).

Nella versione 1.0 di HTTP il recupero degli oggetti (quali immagini) era gestito con la chiusura automatica delle connessioni; quindi ogni volta che veniva un messaggio di risposta al client, veniva chiusa la connessione. Per recuperare diversi oggetti diventava un task impegnativo.

Connessione persistente

Una connessione TCP viene mantenuta attiva dal momento in cui viene stabilita, non solo per il recupero del singolo oggetto, ma viene mantenuta fino a quando tutti gli oggetti **embedded della pagina involucro** sono stati recuperati. Se ad esempio abbiamo una pagina HTML con diverse immagini, ed un client richiede quella determinata pagina, la connessione non viene chiusa finchè **tutte** le immagini non sono state trasferite.

Usare ogni volta una nuova connessione, significa incorrere nello **slow start**, ovvero non sfruttare al meglio le potenzialità del canale per via del fatto che ogni volta subiamo gli effetti del **road trip time**, ovvero il tempo che passa dalla richiesta alla risposta.

Pipelining

Si può ancora migliorare usando altre tecninche; il fatto di eliminare l'attivazione delle connessioni consente di inviare richieste senza risentire dell'overhead dovuto al tempo dell'attivazione della connessione. Ma in ogni caso dopo ogni richiesta dobbiamo attendere la risposta prima di poter effettuare una nuova richiesta.

Per migliorare le prestazioni abbiamo due possibilità:

- Attivare più connessioni in parallelo
- Utilizzare la tecnica del pipelining, introdotta nella versione 1.1 di HTTP.

Con il pipelining il client non è più costretto ad attendere la risposta prima di poter inviare un nuovo messaggio, ma una volta inviata la prima richiesta (che contiene tutte le risorse da chiedere ulteriormente, come delle immagini) può inviare le altre richieste una dopo l'altra, senza dover attendere la risposta della richiesta precedente.

Lunghezza dei messaggi

L'introduzione delle connessioni persistenti, ha però introdotto un **effetto collaterale**: con il funzionamento di HTTP 1.0 (chiusura della connessione dopo la risposta) era facile quando il messaggio di risposta terminava (il browser capisce quando termina il messaggio perchè il server chiude la connessione).

Con le connesisoni persistenti la connesisone non può essere chiusa; abbiamo quindi necessità di inserire nel protocollo, quindi tra le **linee di intestazione**, una linea che ci consente di specificare la dimensione del messaggio. Quando usiamo le connessioni persistenti usiamo quindi il campo `Content-Length`, che viene usato dal client per capire fino a quanto leggere.

Per segnalare al client quanti byte devono essere letti, abbiamo un terzo metodo chiamato **Chunked encoding**:

Chunked Encoding

Questa codifica permette di inviare il body del messaggio di risposta non più come un unico blocco, ma come tanti **chunk** (blocchi); ogni chunk è preceduto da una linea consentente il numero di byte che ci dice di quanti è composto il seguente chunk.

Il vantaggio di questa soluzione possiamo inviare un blocco di byte non appena disponibile, specificando la dimensione solo di quel blocco; non dobbiamo dire al client la **dimensione complessiva**, ma solo del blocco corrente. Di conseguenza non dobbiamo **conoscere a priori la dimensione del contenuto da inviare**.

Evitare di trasferire dati già noti : caching

Sempre per motivi di prestazioni, HTTP prevede di gestire il **caching**; le risorse che chiediamo al server che chiameremo **origin server** (ovvero il server autorevole rispetto alla risorsa che chiediamo, ovvero è il server originario della risorsa) non sono sempre *complete*.

In alcuni casi la risorsa non viene recuperata dall'origin server, ma potrebbe essere recuperato da un server intermedio sul percorso che va dall'origin server al client. Questo perchè ci sono risorse che non variano molto velocemente, ed il loro "deterioramento" è lento.

Sd esempio, il **logo di un'azienda** varia molto lentamente, infatti potrebbe essere adottato anche per anni, quindi potremmo usarlo ripetutamente senza doverlo aggiornare.

La cache potrebbe essere collocata in diversi punti del web, non solo nel nostro browser. La cache è presente anche negli origin server, così come nel percorso dal server al client. La cache si trova anche all'interno dei **proxy**.

I **proxy** sono sostanzialmente un'intermediario tra il client ed il server, che si comporta per il client come se fosse il server. Nasconde di fatto l'interazione verso il server.

Ci sono anche delle **cache distribuite**, realizzate attraverso i **CDN - Content Delivery Network**, ovvero dei vari server che fungono da cache.

Problema dell'inconsistenza

Il problema principale del caching è **l'inconsistenza**, ovvero trovare delle risorse che non sono aggiornate, ovvero che non coincidono con la copia presente **sull'origin server**. Dobbiamo quindi essere sicuri che la copia in cache sia in Lina con quella originale. L'HTTP offre alcuni campi (alcuni già visti) che ci consentono di *combattere* l'inconsistenza:

Last-Modified, IfModified-Since, Expires, ecc.

Last modified è un campo che indica quando quella risorsa è stata modificata; possiamo quindi verificare se la risorsa in cache è allineata a quella presente nell'origin server, usando nella richiesta GET il campo **If-Modified-Since**, con la data recuperata dal campo **Last-Modified**.

Cosa cambia tra una cache locale ed una cache presente nella rete in proxy ed una cache presente nell'origin server?

Per capire lo svantaggio di avere una cache sempre più vicina al browser, pensiamo al concetto di **search hit/miss**; infatti ogni volta che un qualsiasi sistema cerca all'interno di una cache, potrebbe non trovare la risorsa che sta cercando. Di conseguenza, avere una cache più grande (opportunamente organizzata in modo che sia quanto più veloce possibile), è sicuramente un vantaggio.

Se ad esempio abbiamo una cache all'interno di un proxy condiviso da diversi computers, è molto più probabile che il client possa trovare la entry che sta cercando all'interno della cache del proxy, essendo essa molto più grande.

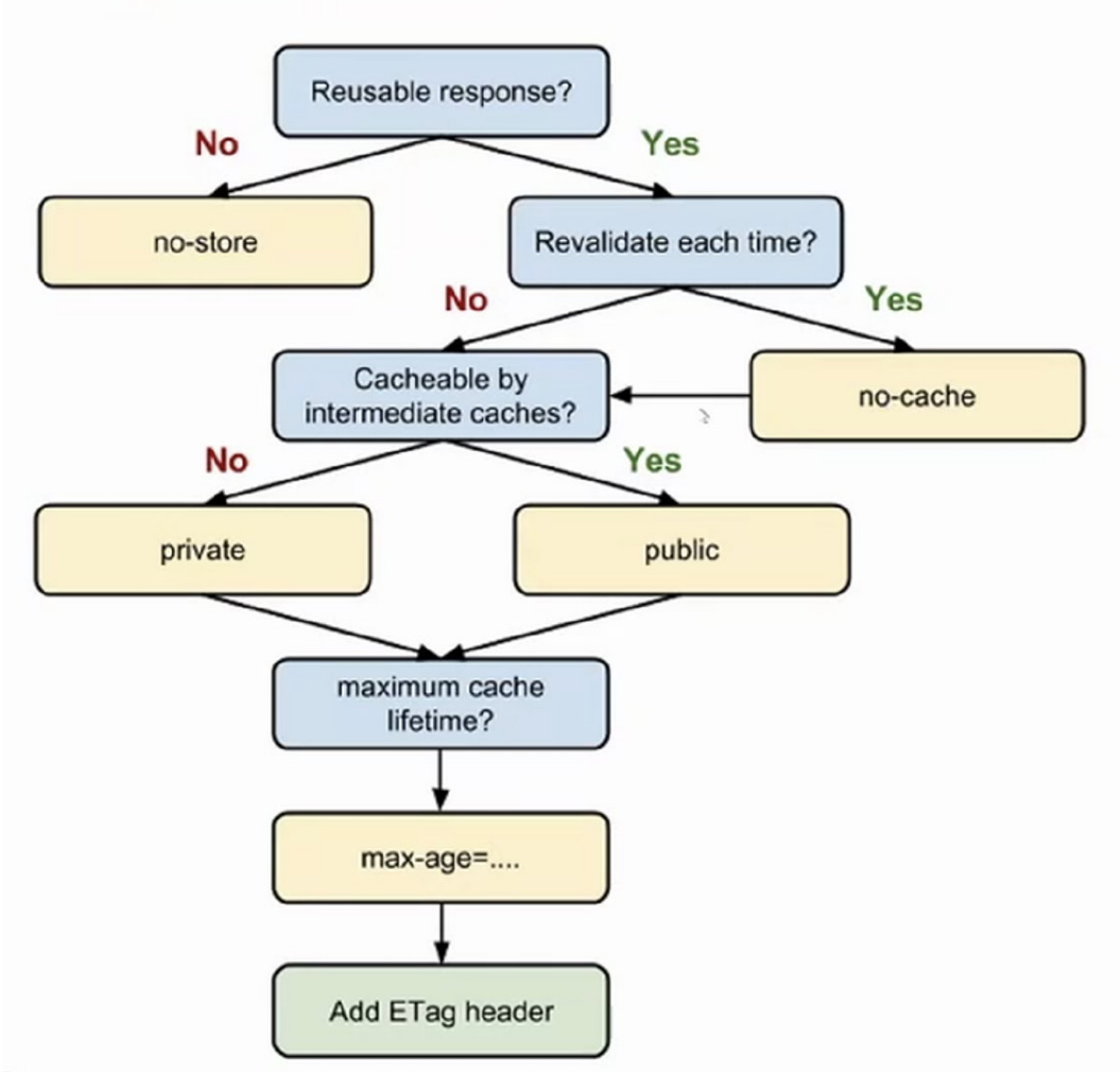
Caching Control

Per il controllo della consistenza della cache, usiamo un campo chiamato **Cache-Control**, che consente di specificare quando la risorsa deve essere memorizzata da una cache, quando deve essere recuperata da una cache, ecc.

Consente quindi di gestire le cache sia dal punto di vista del popolamento sia dal punto di vista dell'utilizzo. Il cache control può essere seguito da diversi valori:

- Evitare il caching
    - no-cache: è utile per specifiacre che il contenuto non deve essere prelevato da una cache
    - no-store: evita che la cache salvi del contenuto
- Favorire il caching
    - Cache-Control: può essere usato epr abilitare la cache con i seguenti valori:
        - <absent> senza header, ogni contenuto può essere salvato in cache
        - **private**: il contenuto può essere usato da uno specifico user e conseguentemente può essere salvato solo nel browser dell'user
        - **public**: il contenuto può essere salvato in cache publiche.

Overview - Cache



Se la risposta non può essere riutilizzabile (deciso dal server) non viene salvata (no-store); se è riutilizzabile ma deve essere rivalidata, perchè potrebbe cambiare, allora anche in questo caso non viene salvata.

Se invece la risposta non deve essere rivalidata, allora possiamo usare degli attributi specifici per indicare che la risorsa deve essere valida per diverso tempo, usando **max-age**.

---

fine lezione 19