Esercizio 9.3

In questo esercizio usiamo i cookie:

```
Cookie[] cookies = request.getCookies();
```

Successivamente verifichiamo se vi sono dei cookies, e sopratutto se tra questi è presente il cookie di autenticazione:

```
if(cookie != null){
  for(int i = 0; i<cookie.length && !authenticated; i++){
    authenticated = cookie[i].getName().equals("authToken") &&
cookie[i].getValue().equals("0192837465");
  }
}
```

Verifichiamo che il codice di quel cookie è il codice di autenticazione.

Se l'utente si autentica, allora:

```
if(authenticated)
  pw.print("welcome");
else{
  pw.print("Non sei autenticato!");
}
```

Esercizio 9.4

Implementiamo l'esercizio 9.3 con HttpSession invece che con i cookies.

🏁 1:02

Paradigma REST

Il paradigma REST - REpresentational State Transfer; con questo paradigma si provava a caratterizzare il comportamento di un'applicazione web. REST nasce come un paradigma che ha l'intenzione di evocare di come un'immagine di un'applicazione ben progettata si comporta.

Una rete di pagine web, dove l'utente avanza attraverso un'applicazione (selezionando dei link e quindi avanzando nella macchina a stati) e la selezione di questi link ha come risultato l'entrata in una pagina successiva.

Con la definizione non troviamo nulla di diverso rispetto a ciò che ci è già noto per le applicazioni WEB.

REST **è un paradigma che specializza il paradigma client-server**. Le richieste e risposte vengono impiegate per trasferire **rappresentazioni di risorsi**. Un esempio di rappresentazione di risorsa può essere l'HTML.

Principi alla base del paradigma REST

Il concetto principale è che tutte le informazioni che vengono scambiate sono organizzate come risorse; le risorse possono essere contenuti informativi di qualsiasi tipo identificabili mediante **un'URI**.

Le risorse astratte possono essere concretizzate in modi diversi (HTML può essere rappresentato in json, xml, plain, ecc). Le componenti applicative sono le stesse componenti che abbiamo visto nel web:

- Lato client: user agents (non per forza browsers)
- Lato server: origin servers

Quando pensiamo al paradigma REST la possibilità di scrivere applicazioni client-server in cui il protocollo di comunicazione usato tra client e server è un protocollo con delle **operazioni standardizzate**;

Le interazioni tra client e server possono essere mediate, ovvero possiamo avere la presenza di componenti intermedie per migliorare le prestazioni.

Vincoli architetturali REST

REST è il risultato dell'applicazione in successione di vincoli.

Primo vincolo - Client Server

Il primo vincolo applicato è il vincolo client-server. Quindi la prima cosa da cui partiamo è il fatto che **REST è un paradigma client-server**; il vincolo è stato introdotto perchè la separazione del codice di un'applicazione in client e server consente di separare gli aspetti: ad esempio possiamo avere un codice del server standardizzato che possiamo usare in diverse applicazioni.

La separazione degli aspetti tra server e client è gestita attraverso un'interfaccia, quindi l'unica cosa che il client deve conoscere per usare le funzionalità che il server mette a disposizione, **sono le operazioni esposte dall'interfaccia del server**.

Questa separazione consente anche di far evolvere l'applicazione client-server in mdo separato: ad esempio il server può essere aggiornato mentre il client rimane lo stesso della versione precedente.

Secondo vincolo - Stateless

La comunicazione è stateless. Stateless significa che lo stato conversazionale tra client e server non viene memorizzato dai server, ma solo dai client, con il supporto dei cookie.

Abbiamo dei riscontri positivi: la scalabilità è maggiore con questo tipo di applicazione; con scalabilità intendiamo la capacità del server di mantenere le prestazioni inalterati all'aumentare del numero dei client.

Abbiamo degli effetti positivi anche sotto il punto di vista dell'affidabilità: pensiamo ad un server che mantiene informazioni di stato che per qualche motivo diventa inutilizzabile. Se un server va giù possiamo ripristinarlo riavviandolo. Se questo server aveva le informazioni di sessione non salvate, vengono completamente perse.

Terzo vincolo - Caching

L'altro vincolo introdotto è il caching; in questo caso le applicazioni divengono sempre più efficienti e scalabili, perchè le informazioni non vengono fornite per forza dagli origin servers ma anche da server intermedi.

In termini di scalabilità riduciamo il numero di computazioni che il server è costretto ad effettuare, e quindi l'efficienza aumenta. Ancora una volta abbiamo il problema della consistenza visto con HTTP/1.1, e le soluzioni analizzate in quel paradigma sono applicabili anche in REST

Quarto vincolo - Interfaccia uniforme

L'idea è che le applicazioni client server possono essere realizzate con un set minimale di operazioni; non dobbiamo pensare per ogni applicazione delle nuove operazioni, ma le operazioni che dobbiamo prevedere sono operazioni di tipo **CRUD**: ovvero operazioni solitamente eseguite su un database; creazione/lettura/aggiornamento/cancellazione.

Action	Verb
Create	POST
Retrieve	GET
Update	PUT
Delete	DELETE

> Abbiamo delle associazioni riportate sopra

Ciò che cambia tra un'applicazione e l'altra è il tipo di risorsa, e non il tipo di operazioni.

Quinto vincolo - Sistema a strati

Il sistema che viene costruito a partire dalle componenti di base (client e server) è composto da diversi strati e tra il client e server possiamo prevedere degli strati intermedi: ad esempio nel caching avviene una stratificazione. Possiamo ad esempio prevedere che un componente intermedio faccia un **bilanciamento del carico**, che ha il compito di distribuire le richeiste che vengono dal client verso diverse istanze del server.

Sesto vincolo - Code on demand

Un altro possibile vincolo è il codice su richiesta: vede la possibilità che le componenti possano essere arricchite funzionalmente con del codice che viene recuperato dai server; possiamo quindi aggiungere funzionalità in maniera semplice, potendo quindi aggiornare le funzionalità senza dover aggiornare il client.

Ad esempio possiamo eseguire del codice javascript (o come veniva fatto anni fa, codice java) su richiesta.

Quindi

Il concetto da memorizzare è che le applicazioni client-server possono essere sviluppate in accordo con questo paradigma, sfruttando sempre lo **stesso protocollo**, e quindi usando sempre lo stesso **set di applicazioni**.

Il paradigma REST è un paradigma diffuso di recente, ma nato molti anni fa.

RESTFUL service

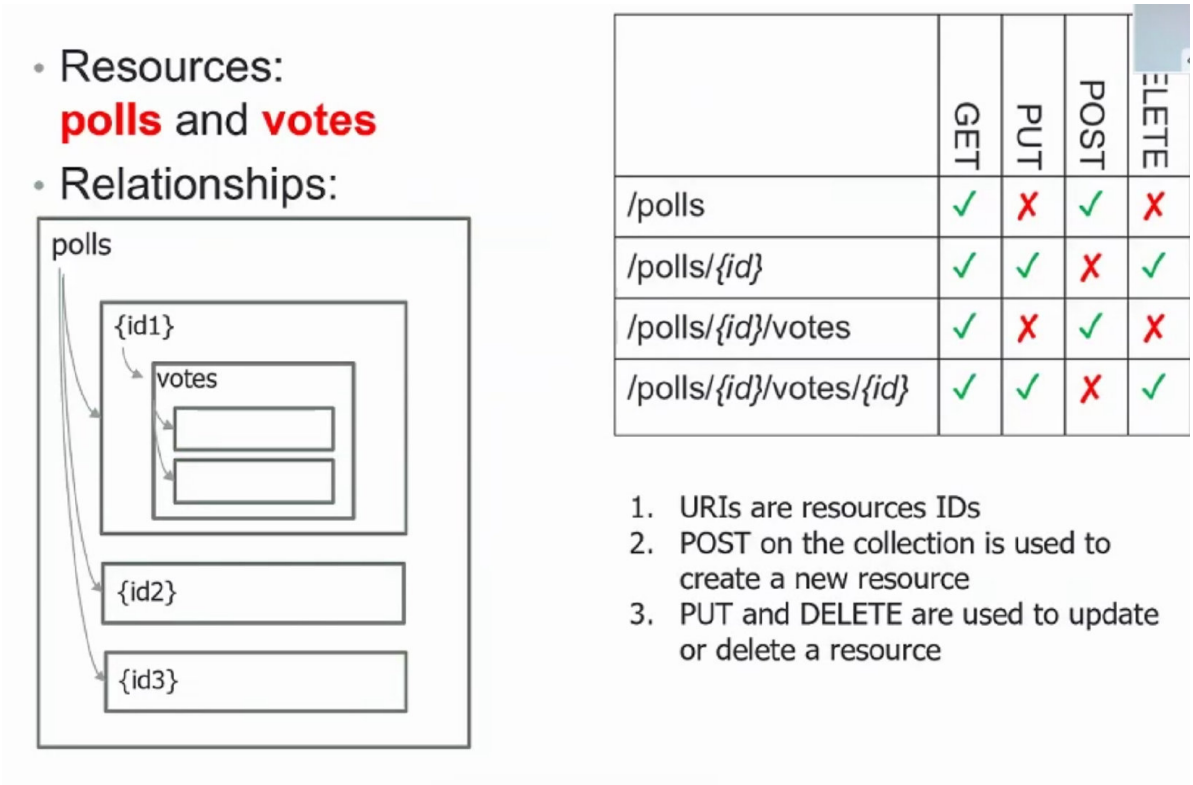
Un servizio RESTFUL è una connessione di risorse. Quando parliamo di servizi, escludiamo in prima istanza l'idea che il client debba essere il browser, ma il client può essere una qualsiasi componente software, che potrebbe anche escludere l'utente. Di conseguenza le risorse recuperate non devono per forza essere renderizzate per l'utente, ma potrebbero anche essere processate per altri scopi.

Possiamo pensare ad un caso particolare in cui il client è un browser, ed il servizio RESTFUL è un sito web; in quel caso l'endpoint usato per raggiungere il sito web è il punto di accesso al servizio, e le risorse scambiate sono rappresentate sono rappresentate in HTML.
In generale però un servizio è pensato per fornire contenuti informativi in una rappresentazione che non è in HTML.

Come realizzare un servizio RESTFUL

- Abbiamo parlato di **collezioni di risorse**, e di fatto un servizio RESTful può essere visto come una **componente che gestisce una collezione di risorse**. Per definire un servizio RESTFUL dobbiamo capire quali sono le collezioni da trattare.
- Le risorse che afferiscono a queste collezioni possono avere delle relazioni, quindi dobbiamo modellare le relazioni che esistono tra le risorse utilizzando degli **hyperlink**; un po' come si fa con la definizione di un modello **entità relazione**.
- Una volta che abbiamo individuato le relazioni, un aspetto importante della definizione di un servizio RESTFUL è la definizione degli **URI**, ovvero degli identificatori che usiamo per raggiungere le diverse risorse. Quando definiamo gli URI, e diamo quindi dei nomi alle collezioni, **utilizziamo il plurale** (in relazione alle collezioni).
  Se ad esempio gestiamo un **catalogo di libri**, chiamiamo questa collezione **books**, quindi usiamo l'identificatore **books** nell'URI.
  Inoltre, dobbiamo avere altri elementi (nell'URI) che ci permettono di distinguere una collezione di libri da un'altra collezione di libri.
- Una volta che abbiamo definito le collezioni, ed abbiamo associato a ciascuna collezione un'identificatore, dobbiamo **definire le operazioni (metodi) che sono significative per quelle collezioni**.
  Di conseguenza, tra i metodi **GET, POST, PUT, DELETE** dobbiamo capire quali possiamo applicare alle collezioni o alle singole risorse delle collezioni.
- Dobbiamo definire la rappresentazione; se con books identifichiamo la collezione, con book identifichiamo la singola risorsa, posso ottenere una risorsa di tipo book usando GET, posso creare una risorsa di tipo book usando POST, ma cosa scambiamo?
  Poichè lo scambio tra client e server prevede l'utilizzo di rappresentazioni, dobbiamo decidere quale rappresentazione usare; ad esempio, un libro che ha una serie di informazioni (titolo, autori, ISBN), queste informazioni possono essere rappresentate in modi diversi (XML, JSON, ecc.)
  Quindi una fase importante nella progettazione di un servizio RESTFUL è la definizione della rappresentazione che useremo.
- Fatto tutto questo, dobbiamo implementare le azioni che il server metterà a disposizione, che sono previste in risposta alla ricezione di messaggi che contengono i metodi visti in precedenza; se arriva un messaggio di tipo GET eseguiamo un'azione che ha come effetto la **produzione di una risorsa**, se arriva un messaggio di tipo POST eseguiamo un'azione che ha come effetto la **creazione di una risorsa**, e così via per PUT e DELETE.
- L'ultimo passaggio è ovviamente la fase di **testing dell'applicazione** con un client.

Esempio - Servizio di votazioni



Per prima cosa identifichiamo le risorse ed effettuiamo delle relazioni tra entità che danno vita al mio servizio. Abbiamo una collezione di votazioni (polls), e per ciascuna votazione identificata da un $\{id_n\}$, abbiamo dei voti (votes); all'interno della collezione votes abbiamo i singoli voti.

Avendo scelto per individuare le collezioni gli identificatori **polls e votes**, usiamo un'URI contenente **/polls**; a sinistra dell'identificatore saranno presenti altri elementi.
A partire da una collezione il metodo POST viene usato per chiedere la creazione di un elemento di quella collezione: se viene formulata una richiesta di tipo POST con all'interno il messaggio di richiesta il metodo POST stiamo dicendo che in questa collezione vogliamo creare una nuova risorsa.

Allo stesso modo con GET chiediamo la restituzione di tutte le risorse presenti in quella colelzione.

Fine lezione 33