

Laboratorio di Programmazione di Rete - B

Federica Paganelli

Università di Pisa

Web Services

- Un web service è un programma che:
 - Utilizza una connessione web
 - Richiede ad un server esterno di fornire dati o eseguire un algoritmo
- Possibili tipologie
 - RESTful
 - RPC style
 - Ibrido REST-RPC

REST

- Il REpresentational State Transfer è uno *stile architettonico* per lo sviluppo di web services
 - Un insieme di vincoli applicati nella progettazione di una architettura
- Introdotto nel 2001 da Roy Fielding
- Si sostiene che possa essere visto come la base di HTTP
 - In realtà è stato distillato *ex post* da HTTP

HTTP

HTTP Request

- GET /index.html HTTP 1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:20.0)
Gecko/20100101 Firefox/20.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
- Nessun body!
 - Tipico del metodo GET

HTTP Response

- HTTP 1.1 200 OK
 - Date: Mon, 28 Oct 2013 14:00:00 GMT
 - Server: Apache
 - Content-Length: 6448
 - Content-Type: text/html
 - Content-Encoding: gzip
 - Connection: Keep-Alive

```
<!DOCTYPE html>
<html lang="en">
    <head>...
        <title>...
```

Request-response

1. Informazioni per la richiesta:
 - i. HTTP method
 - ii. URI
 - iii. HTTP request headers
 - iv. Request Entity Body
2. Formattare la richiesta ed inviarla al server
3. Parsing dei dati in risposta:
 - i. Response code
 - ii. HTTP response headers
 - iii. Response Entity Body

Metodi HTTP

- Il metodo HTTP (*HTTP method*) indica quale operazione deve essere eseguita:
 - GET (lettura di una risorsa)
 - POST (creazione di una risorsa)
 - PUT (modifica di una risorsa)
 - DELETE (eliminazione di una risorsa)
- Operazioni "CRUD"
 - Create, Read, Update, Delete

Cosa è il REST?

1. Uno *stile architetturale* per costruire sistemi debolmente accoppiati:
 - Definito da un insieme di vincoli generali
 - Il Web (URI, HTTP, HTML, XML) come *istanza* di tale stile
2. Il Web usato correttamente
 - Ovvero non usato come trasporto (come in SOAP)
 - HTTP costruito in accordo ai principi RESTful
 - I servizi sono costituiti sulla cima degli standard Web, senza usarli in modo scorretto
 - HTTP è un *protocollo applicativo*
3. NON Qualsiasi cosa *utilizzi HTTP e XML* o JSON
 - XML-RPC fu il primo approccio per questo
 - XML-RPC non ha una interfaccia uniforme

PRINCIPI BASE DEL REST

Lo stile architetturale REST

- ROA (*Resource Oriented Architecture*) è un insieme di linee guida per implementare un web service RESTful
- I principi alla base di una architettura RESTful:
 - A. Resource Identification
 - B. Uniform Interface
 - C. Self-Descriptive Messages
 - D. Hypermedia As The Engine Of Application State
 - E. Stateless Interactions

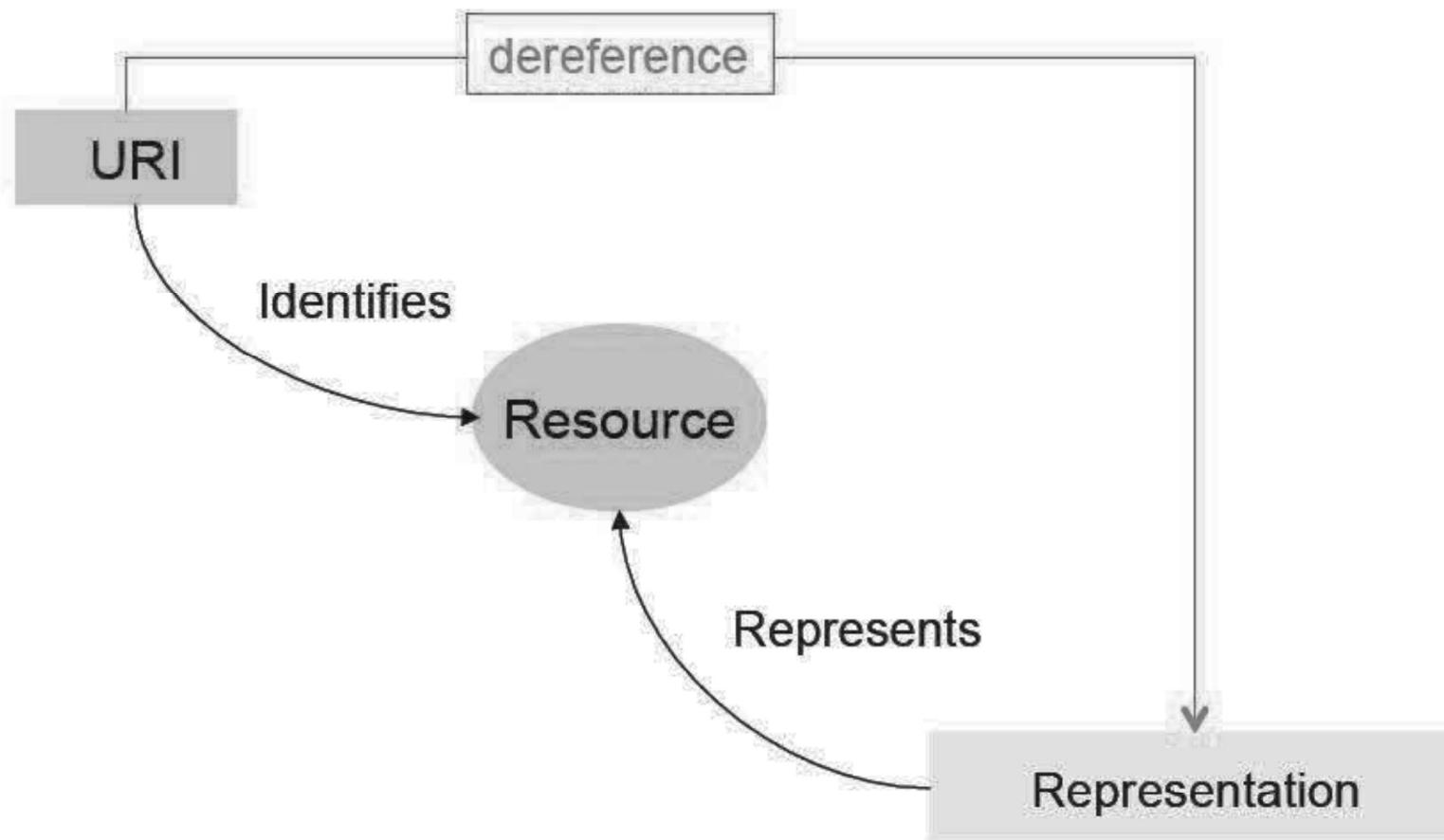
A – Resource Identification (1)

- Risorsa: qualsiasi cosa sufficientemente importante da poter essere considerata come entità a sé stante
 - Qualcosa rappresentabile in un computer
 - Un oggetto fisico (una mela, un tavolo, ecc...)
- Esempi di risorse
 - La versione 1.0 di tal software
 - Una mappa di Firenze
 - Il prossimo numero primo dopo 20
 - Le vendite del primo trimestre 2013

A – Resource Identification (2)

- Dare un nome a tutto ciò di cui si vuole parlare:
 - *Prodotti* in un negozio online
 - *Categorie* utilizzate per raggruppare i prodotti
 - *Clienti* che intendono comperare i prodotti
 - *Carrelli* dove i clienti inseriscono i prodotti
- A tal fine sono utilizzati gli URI
- Si parla anche di *URIs as resource identifiers*; poiché gli HTTP URIs appartengono ad uno spazio di indirizzabilità globale, le risorse identificate mediante un URI hanno uno *scope* globale.

Risorsa, URI, Rappresentazione (1)



Identificare risorse sul Web

- Gli URI (Uniform Resource Identifier) sono essenziali per implementare la Resource Identification
- Gli URI sono identificatori universali per le cose, generalmente leggibili dagli esseri umani
 - Taluni schemi di identificazione non sono leggibili dagli esseri umani

Struttura di un URI

- URI = schema://authority/path[?query][#fragment]
- <http://www.google.it/search?q=rest&start=10#1>
 - Schema: http
 - Authority: www.google.it
 - Path: /search
 - Query: ?q=rest&start=10
 - Fragment: #1
- I fragments non sono inviati al server

Esempi di URI

- La versione 1.0 di tal software
 - <http://www.example.com/software/versioni/1.0.zip>
- Una mappa di Firenze
 - <http://www.example.com/mappa/italia/toscana/firenze>
- Il prossimo numero primo dopo 20
 - <http://www.example.com/prossimo-primo/20>
- Le vendite del primo trimestre 2013
 - <http://www.example.com/vendite/2013/Q1>

Linee guida per gli URI

- Preferire i nomi ai verbi
 - GET /book?isbn=24&action=delete
 - DELETE /book/24
- Mantenere gli URI brevi
- Non cambiare gli URI
 - Utilizzare la redirection se davvero serve cambiarli
- **NOTA:** Gli URI REST sono identificatori che devono essere scoperti seguendo i collegamenti ipertestuali e non devono essere costruiti dal client

URI Template (1)

- Una tecnica per definire URI che includano parametri, che devono essere sostituiti prima di utilizzare l'URI
 - <http://example.com/prodotti/{id}>
- Il REST non si cura di dettagli sugli URI
- L'URI Template non è richiesto dal REST, tecnicamente parlando
 - Praticamente parlando, l'URI Template si rivela una best practice

URI Templates (2)

- Gli URI Templates specificano come costruire ed effettuare il parsing di URI parametrici
 - Sul server sono spesso usati per configurare "regole di routing"
 - Sul client sono utilizzati per istanziare URI a partire da parametri locali



Query

- I componenti delle query specificano informazioni aggiuntive
 - Informazione non gerarchica che identifica ulteriormente la risorsa
 - In molti casi può essere considerato come un *input* per la risorsa

B – Uniform Interface (1)

- Lo stesso piccolo insieme di operazioni si applica a *tutto*
- Un piccolo insieme di *verbi* applicato ad un largo insieme di *sostantivi*
- I verbi sono universali e non sono inventati in base all'applicazione
 - Se servono nuovi verbi a molte applicazioni, l'interfaccia può essere estesa

Metodi HTTP REST

Metodo	Semantica
POST	Crea una sottorisorsa
GET	Restituisce una rappresentazione della risorsa
PUT	Inizializza o aggiorna lo stato di una risorsa
DELETE	Elimina una risorsa

- Altre operazioni
 - HEAD: legge i metadati di una risorsa (GET senza response body)
 - OPTIONS: elenco di operazioni possibili su una risorsa
 - PATCH: modificare una parte di una risorsa

Metodo POST

- Il metodo POST consente di creare risorse come subordinate ad una esistente
 - In pratica una operazione di "append"
- Spesso, il server risponde con: 201 Created
- Nel response header Location è indicato l'URI della risorsa creata
 - Tale URI può essere utilizzato per future GET, PUT, DELETE

POST vs GET

- GET è una operazione di sola lettura
 - Può essere ripetuta senza generare effetti sullo stato della risorsa (*idempotente*) e cachata
 - N.B. Questo non vuol dire che ogni volta sarà restituita la stessa rappresentazione
- POST è una operazione di lettura e scrittura
 - Potrebbe modificare lo stato della risorsa
 - Potrebbe provocare effetti collaterali sul server

POST vs PUT (1)

- Quale è il modo giusto per creare le risorse?

PUT /resource/{id}

201 Created

- Problema: come assicurarsi che l'identificativo della risorsa {id} sia univoca?
- Soluzione 1: lasciare che sia il client a scegliere un id univoco

POST vs PUT (2)

POST /resource

301 Moved Permanently

Location: /resource/{id}

- Soluzione 2: lasciare che sia il server a calcolare l'id univoco

POST vs PUT (3)

- Creazione di una risorsa: POST o PUT?
 - POST viene utilizzato quando il *server* decide l'URI
 - PUT viene utilizzato quando il *client* decide l'URI
- PUT è usato anche per aggiornare risorse esistenti

	PUT a una nuova risorsa	PUT ad una risorsa esistente	POST
/weblogs	n.a. (La risorsa è già esistente)	Non ha effetto	Crea un nuovo blog
/weblogs/myblog	Crea questo blog	Modifica le impostazioni del blog	Crea una nuova entry nel blog
/weblogs/myblog/entries/1	n.a. (Come si è avuto questo URI?)	Modifica questa entry del blog	Inserisce un commento a questa entry

Metodi idempotenti

- Un metodo HTTP *idempotente* genera sul server lo stesso effetto quando applicato 1 o n volte
 - Ovvero la seconda richiesta e le successive lasciano lo stato della risorsa nello stesso stato in cui lo ha lasciato la prima richiesta
 - Aritmeticamente: $12 \times 0 \times 0 \times 0 \times 0 \times 0 \times 0$
- Le richieste idempotenti possono essere processate molte volte senza effetti collaterali
 - Se qualcosa va storto (server spento, errore del server) la richiesta può essere ripetuta fino al buon esito
- GET, PUT e DELETE sono idempotenti

Metodi safe

- Un metodo HTTP *safe* non genera effetti collaterali sul server
- I *metodi safe* possono essere ripetuti *n* volte senza effetti collaterali
 - Aritmeticamente: $12 \times 1 \times 1 \times 1 \times 1 \times 1 \times 1$
 - In pratica, *senza effetti collaterali* vuol dire senza effetti collaterali rilevanti
- GET e HEAD sono metodi safe
 - Può cambiare il contatore degli accessi, ma viene considerato effetto collaterale irrilevante

Metodi idempotenti e safe

Metodo	Semantica	Idempot.	Safe
POST	Crea una sottorisorsa	NO	NO
GET	Restituisce una rappresentazione della risorsa	SI	SI
PUT	Inizializza o aggiorna lo stato di una risorsa	SI	NO
DELETE	Elimina una risorsa	SI	NO

C – Self-Descriptive Messages (1)

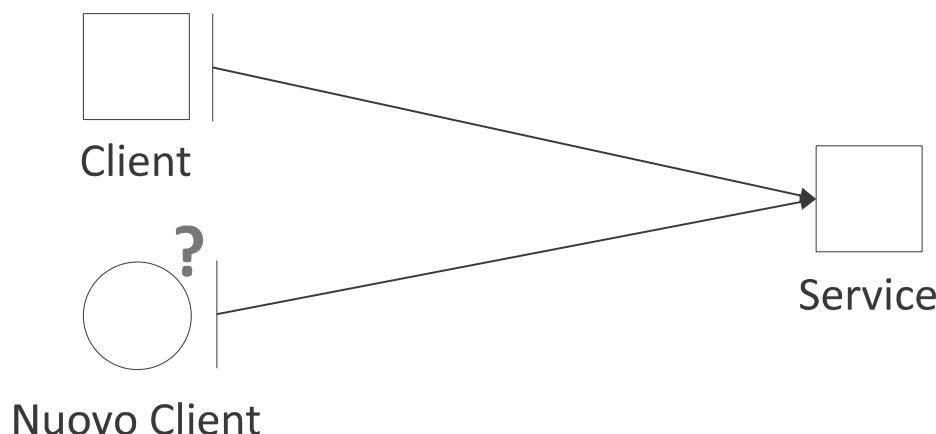
- Ogni messaggio contiene le informazioni necessarie per la propria gestione
- Le risorse sono entità astratte (non possono essere usate *di per sé*) :
 - L'identificazione delle risorse garantisce che esse siano chiaramente identificate
 - L'accesso avviene attraverso l'interfaccia uniforme

C – Self-Descriptive Messages (2)

- L'accesso alle risorse avviene usando la loro *rappresentazione*
 - Ovvero lo stato attuale della risorsa
 - Viene comunicato che tipo di rappresentazione utilizzare
 - Il formato di rappresentazione è negoziabile tra pari
- La rappresentazione delle risorse può essere basata su vincoli diversi
 - XML e JSON possono rappresentare lo stesso modello
 - Qualunque sia la rappresentazione, essa deve supportare i link

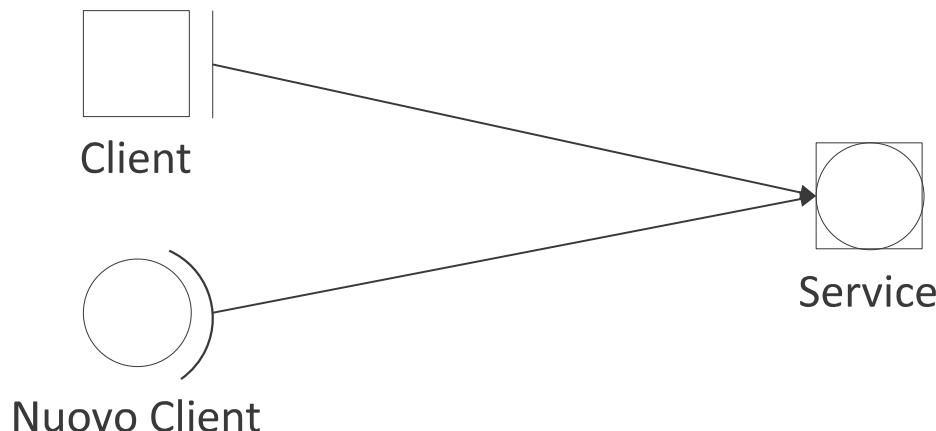
Content Negotiation (1)

- Problema: Se i consumer cambiano requisiti per il formato dei messaggi in modo non retrocompatibile?
- Un service dovrebbe supportare sia i vecchi consumer sia i nuovi, senza dovere introdurre una interfaccia specifica per ogni tipo di consumer



Content Negotiation (2)

- Soluzione: Utilizzare la Content Negotiation
- I formati di dati specifici da accettare o restituire da parte del service sono negoziati a runtime come parte dell'invocazione. Si parla di **media types**
- Benefici: minore accoppiamento, interoperabilità maggiore



Content Negotiation (3)

- Negoziare il formato dei messaggi non richiede l'aggiunta di ulteriori messaggi
- Il client invia i formati comprensibili (**MIME types**)
- Il server sceglie il formato più appropriato per la risposta

GET /resource

Accept: text/html, application/xml, application/json

200 OK

Content-Type: application/json

Content Negotiation (4)

- I fattori di qualità permettono al client di indicare un grado di preferenza per ogni rappresentazione
 - 0: non accettabile

Accept: text/html, text/*; q = 0.1

Accept: application/xhtml+xml; q=0.9, text/html;
q=0.5, text/plain; q=0.1

Content Negotiation multidimensione

- La Content Negotiation è molto flessibile e può essere utilizzata su differenti dimensioni

Request Header	Valori di esempio	Response Header
Accept	application/xml, application/json	Content-Type
Accept-Language	en, fr, de, es	Content-Language
Accept-Charset	iso-8859-5, unicode-1-1	parametro Charset di Content-Type
Accept-Encoding	compress, gzip	Content-Encoding

Rappresentazioni

- La rappresentazione scelta dipende da vari fattori
 - La natura delle risorse
 - La capacità del server
 - La capacità del mezzo di comunicazione
 - La capacità del client
 - Requisiti e vincoli dello scenario applicativo
- La negoziazione consente di ottenere la rappresentazione "migliore"

XML (eXtensible Markup Language)

- Media type: application/xml
- Il linguaggio da cui "tutto ebbe inizio"
 - Originato da SGML
 - Considerato il primo linguaggio universale per i dati strutturati
- XML è un metalinguaggio
 - Un linguaggio per rappresentare altri linguaggi
 - Molti linguaggi specifici di dominio sono definiti come *vocabolari* XML

XML (eXtensible Markup Language)

- XML è solo sintassi e non ha (quasi) semantica
 - Semantica minima (ID, URI relative)
 - La semantica è lasciata interamente ai vocabolari XML
- XML è costruito intorno ad un modello ad albero
 - Ogni documento XML è un albero e perciò limitato in struttura
 - XML RESTful introduce i collegamenti ipertestuali per trasformare gli alberi XML in grafi

Esempio XML

```
<?xml version="1.0"?>
<libri>
    <libro id="1">
        <titolo>La Divina Commedia</titolo>
        <autore>Dante Alighieri</autore>
        <anno>1321</anno>
    </libro>
    <libro id="2">
        <titolo>De Vulgari Eloquentia</titolo>
        <autore>Dante Alighieri</autore>
        <anno>1305</anno>
    </libro>
</libri>
```

JSON (JavaScript Object Notation)

- Media type: application/json
- JSON codifica dati come oggetti JavaScript
 - Più efficiente per consumer JavaScript
 - Questo modifica i servizi XML usabili genericamente in servizi orientati a JavaScript
 - Per servizi su larga scala ha senso fornire XML e JSON

Esempio XML (1)

```
<?xml version="1.0"?>
<libro id="1">
    <titolo>La Divina Commedia</titolo>
    <autore>Dante Alighieri</autore>
    <anno>1321</anno>
</libro>
```

Esempio JSON (1)

```
{ "libro" : {  
    "id" : "1",  
    "descr" : {  
        "titolo" : "La Divina Commedia",  
        "autore" : "Dante Alighieri",  
        "anno" : "1321" }  
    }  
}
```

Esempio XML (2)

```
<?xml version="1.0"?>
<libri>
    <libro id="1">
        <titolo>La Divina Commedia</titolo>
        <autore>Dante Alighieri</autore>
        <anno>1321</anno>
    </libro>
    <libro id="2">
        <titolo>De Vulgari Eloquentia</titolo>
        <autore>Dante Alighieri</autore>
        <anno>1305</anno>
    </libro>
</libri>
```

Esempio JSON (2)

```
{ "libri": [   { "id" : "1",   "descr" : {     "titolo" : "La Divina Commedia",     "autore" : "Dante Alighieri",     "anno" : "1321"   }},   {     "id" : "2",     "descr" : {       "titolo" : "De Vulgari Eloquentia",       "autore" : "Dante Alighieri",       "anno" : "1305"     }   } ] }
```

D – Hypermedia As The Engine of Application State (HATEAOS)

- Sono trasferite rappresentazioni delle risorse contenenti link
 - Il client può procedere al passo successivo dell’interazione scegliendo uno di questi link
- Le risorse e lo stato può essere utilizzato navigando i link
 - I link possono interconnettere risorse navigabili
- Le applicazioni RESTful *navigano* invece di *chiamare*
 - Le rappresentazioni contengono informazioni riguardo possibili attraversamenti
 - Le applicazioni navigano alla prossima risorsa sulla base dei link semantici
 - La navigazione può essere delegata, dato che tutti i link utilizzano identificatori

HATEOAS

GET /conto/12345 HTTP/1.1

HTTP/1.1 200 OK

```
<?xml version="1.0"?>
<conto>
  <numero>12345</numero>
  <saldo valuta="eur">500.00</saldo>
  <link rel="deposito" href="/conto/12345/deposito" />
  <link rel="prelievo" href="/conto/12345/prelievo" />
  <link rel="bonifico" href="/conto/12345/bonifico" />
  <link rel="chiusura" href="/conto/12345/chiusura" />
</conto>
```

HATEOAS

GET /conto/12345 HTTP/1.1

HTTP/1.1 200 OK

```
<?xml version="1.0"?>
<conto>
  <numero>12345</numero>
  <saldo valuta="eur">-50.00</saldo>
  <link rel="deposito" href="/conto/12345/deposito" />
</conto>
```

E – Stateless Interactions (1)

- Ogni richiesta dal client al server deve contenere le informazioni necessarie per capire completamente la richiesta, indipendentemente da qualunque richiesta precedente
 - La richiesta avviene in completo isolamento
- Non significa applicazioni stateless!
 - Per molte applicazioni REST, lo stato è una componente essenziale
 - L'idea del REST è quella di evitare transazioni di lunga durata *nelle applicazioni*

E – Stateless Interactions (2)

- Lo *stato della risorsa* è gestito sul server
 - È lo stesso per ogni client che lavora con il servizio
 - Se un client cambia lo stato della risorsa, gli altri client devono vedere questo cambio
- Lo *stato del client (applicazione)* è gestito sul client
 - Specifico per ogni client
 - Potrebbe avere effetto sull'accesso al server, ma non sulle risorse stesse
 - ad es. il fatto che l'utente abbia visualizzato la figura n.3

Stateless shopping (1)

- Uno scenario tipico di sessione può essere mappato sulle risorse
 - Client: Mostrami i prodotti
 - Server: Qui c'è una lista di tutti i miei prodotti
 - Client: Compro 1 di http://ex.org/prodotto/X, sono "Pippo", "password"
 - Server: Aggiunto 1 di http://ex.org/prodotto/X a http://ex.org/utenti/pippo/carrello
 - Client: Compro 1 di http://ex.org/prodotto/Y, sono "Pippo", "password"
 - Server: Aggiunto 1 di http://ex.org/prodotto/Y a http://ex.org/utenti/pippo/carrello

Stateless shopping (2)

- Continua...
 - Client: Rimuovi 1 di http://ex.org/prodotto/X, sono "Pippo", "password"
 - Server: Rimosso 1 di http://ex.org/prodotto/X a http://ex.org/utenti/pippo/carrello
 - Client: OK, ho finito, sono "Pippo", "password"
 - Server: Ecco il costo degli oggetti nel carrello
http://ex.org/utenti/pippo/carrello
- Questo è più che rinominare da "sessione" a "risorsa"
 - Tutti i dati rilevanti sono conservati nel server in modo persistente
 - L'URI del carrello può essere utilizzato da altri servizi, per lavorare con il relativo contenuto
 - Invece di nascondere il carrello nella sessione, esso viene esposto come una risorsa

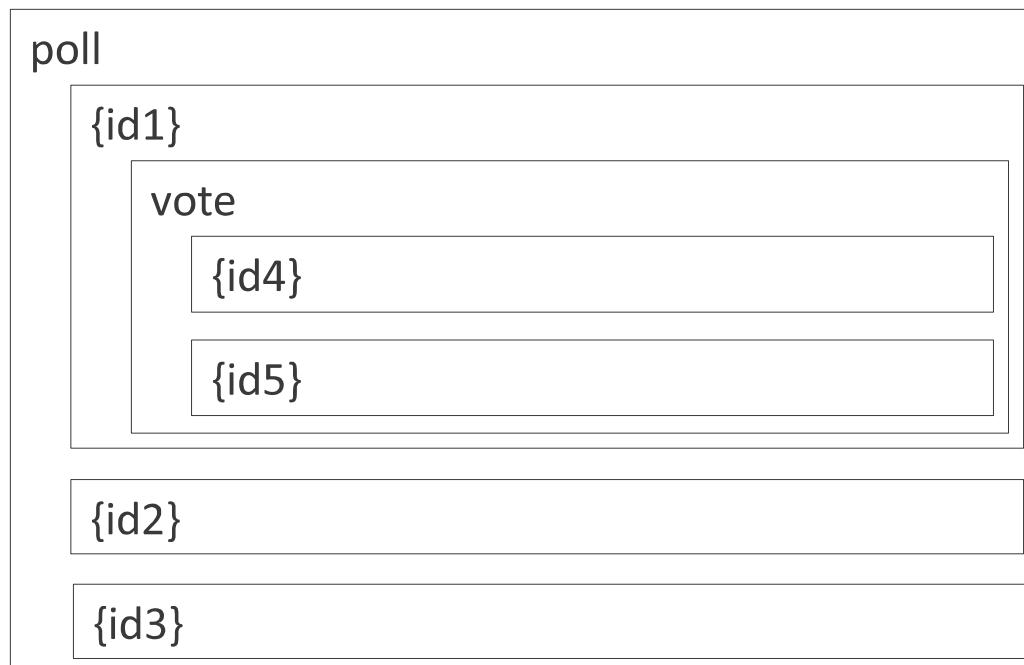
METODOLOGIA DI PROGETTAZIONE

Metodologia di progettazione

1. Identificare le risorse che devono essere esposte come servizio (es. catalogo di libri, ordini di acquisto, urne e voti)
2. Modellare le relazioni (es. contenimento, riferimento, transizione di stato) tra le risorse con collegamenti ipertestuali che possono essere seguiti per ottenere maggiori dettagli (o eseguire transazioni di stato)
3. Definire URI "carini" per indirizzare le risorse
4. Capire cosa significa eseguire GET, POST, PUT e DELETE per ogni risorsa (e se è ammissibile o meno)
5. Progettare e documentare la rappresentazione delle risorse
6. Implementare ed effettuare il deploy su server web
7. Testare con un browser

Esempio: semplice API Doodle (1)

1. Risorse: urne e voti
2. Relazioni di contenimento



Esempio: semplice API Doodle (2)

3. Gli URI contengono gli ID delle risorse figlio

- /poll
- /poll/{id}
- /poll/{id}/vote
- /poll/{id}/vote/{id}

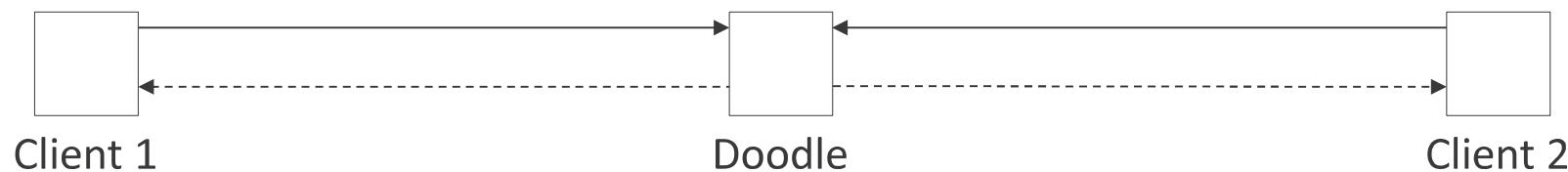
Esempio: semplice API Doodle (3)

4. POST sui contenitori utilizzata per creare risorse figlie; PUT e DELETE per aggiornare e rimuovere le risorse figlie
-

	GET	PUT	POST	DELETE
/poll	✓	✗	✓	✗
/poll/{id}	✓	✓	✗	✓
/poll/{id}/vote	✓	✗	✓	✗
/poll/{id}/vote/{id}	✓	✓	✗	✓

Esempio: semplice API Doodle (4)

1. Creazione di un'urna
2. Lettura di un'urna



POST /poll
<options>A, B, C</options>

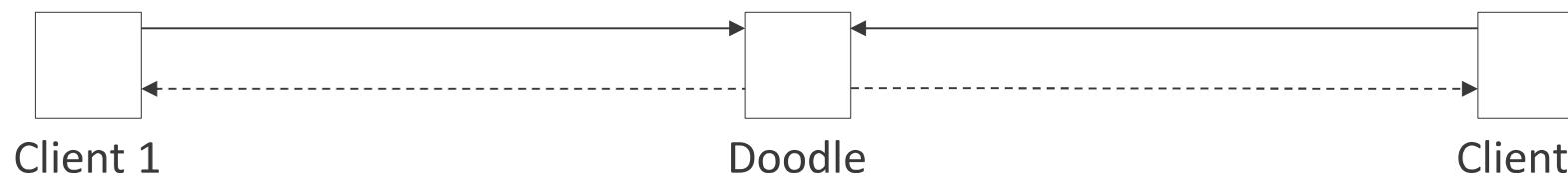
201 Created
Location: /poll/131028x

GET /poll/131028x

200 OK
<options>A, B, C</options>
<votes href="/vote" />

Esempio: semplice API Doodle (5)

3. Partecipare ad un sondaggio creando una nuova sottorisorsa voto



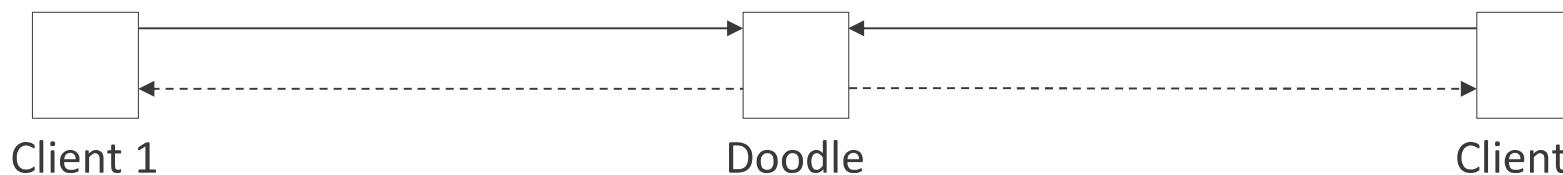
POST /poll/131028x/vote
<name>Mario Rossi</name>
<choice>B</choice>

201 Created
Location: /poll/131028x/vote/1

GET /poll/131028x
200 OK
<options>A, B, C</options>
<votes>
 <vote id="1">
 <name>Mario Rossi</name>
 <choice>B</choice>
 </vote>
</votes>

Esempio: semplice API Doodle (6)

4. Aggiornamento dei voti esistenti



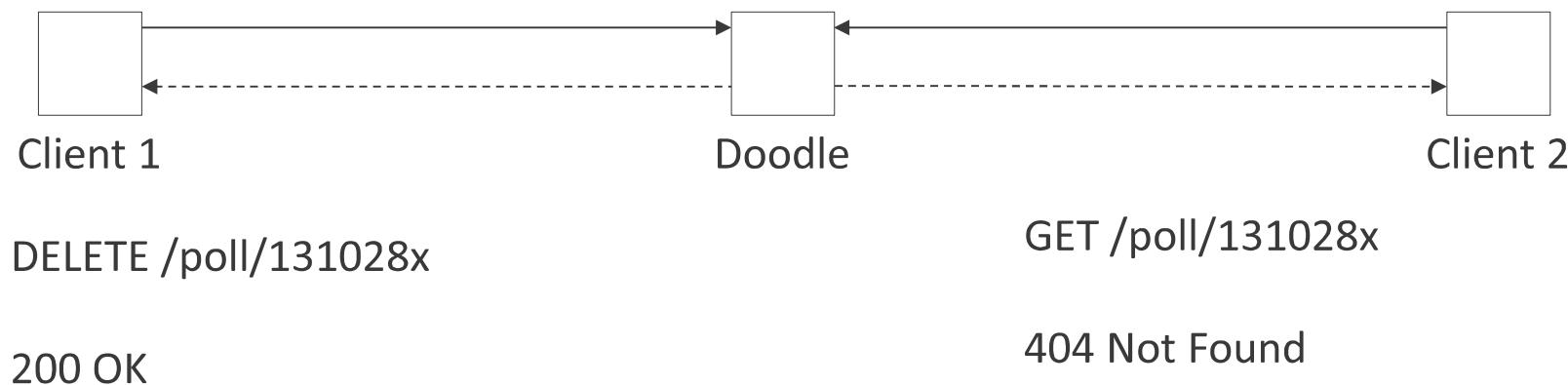
PUT /poll/131028x/vote/1
<name>Mario Rossi</name>
<choice>C</choice>

200 OK

GET /poll/131028x
200 OK
<options>A, B, C</options>
<votes>
<vote id="1">
<name>Mario Rossi</name>
<choice>C</choice>
</vote>
</votes>

Esempio: semplice API Doodle (6)

5. Una volta presa la decisione, l'urna può essere cancellata



API Doodle

Doodle

- Doodle:
 - <http://doodle.com>
- Vera API di Doodle:
 - <http://doodle.com/xsd1/RESTfulDoodle.pdf>

ANTI-PATTERNS

Tunneling su GET/POST

- Effettuare un tunneling sul metodo GET

GET /api?method=addCustomer&name=Mario+Rossi

GET /api?method=deleteCustomer&id=42

GET /api?method=getCustomerName&id=42

GET /api?method=findCustomer&name=Mario*

- Vantaggi: facile da testare con web browser
- Problemi:
 - Cosa succede se salvo il link nei preferiti?
 - Un crawler potrebbe causare problemi
- Analogamente non è corretto il tunneling su POST (stile SOAP)

Ignorare gli status code

- Spesso è utilizzato un set ridotto di status code
 - 200 OK, 404 Not Found, 500 Internal Server Error
- Usando tutti gli status code client e server comunicano con un livello semantico più ricco
 - 201 Created: è stata creata una nuova risorsa, all'indirizzo Location
 - 409 Conflict: esiste un conflitto, ad esempio si è usato PUT con dati basati su una vecchia versione della risorsa
 - 412 Precondition Failed: il server non può soddisfare le attese del client

Dimenticare gli hypermedia

- L'hypermedia (collegare le cose tra loro) è il concetto alla base del Web
 - Un insieme connesso di risorse, dove le applicazioni si spostano da uno stato all'altro seguendo i collegamenti
- Casi possibili:
 - assenza di link nelle rappresentazioni: il server non invia informazioni, il client costruisce gli URI
 - Misto tra costruzione URI lato client e link che rappresentano le relazioni del sottostante data model
 - Idealmente gli URI e le modalità per costruirli (es. query) devono essere comunicati via hypermedia come link nelle rappresentazioni delle risorse
 - Es. Atom Publishing Protocol dispone dei service documents

Rompere la self-descriptiveness

- Self-descriptiveness: ogni HTTP request e response deve contenere tutta l'informazione necessaria per poter essere processata dal client/server
- Ogni volta che si utilizzano headers, formati o protocolli non standard si rompe tale assioma

CODICI DI STATO

Codici di stato HTTP

- 1xx Meta
- 2xx Success
- 3xx Redirection
- 4xx Client-Side Error
- 5xx Server-Side Error

1xx - Meta

- 100 Continue
- 101 Switching Protocols

2xx - Success

- 200 OK
- 201 Created
- 202 Accepted
- 203 Non-Authoritative Information
- 204 No Content
- 205 Reset Content
- 206 Partial Content

3xx - Redirection

- 300 Multiple Choices
- 301 Moved Permanently
- 302 Found
- 303 See Other
- 304 Not Modified
- 305 Use Proxy
- 307 Temporary Redirect

4xx – Client-Side Error

- 400 Bad Request
- 401 Unauthorized
- 402 Payment Required
- 403 Forbidden
- 404 Not Found
- 405 Method Not Allowed
- 406 Not Acceptable
- 407 Proxy Authentication Required
- 408 Request Timeout
- 409 Conflict
- 410 Gone
- 411 Length Required
- 412 Precondition Failed
- 413 Request Entity Too Large
- 414 Request-URI Too Long
- 415 Unsupported Media Type
- 416 Requested Range Not Satisfiable
- 417 Expectation Failed

5xx – Server-Side Error

- 500 Internal Server Error
- 501 Not Implemented
- 502 Bad Gateway
- 503 Service Unavailable
- 504 Gateway Timeout
- 505 HTTP Version Not Supported

Bibliografia

R. Billero, REST, seminario per il corso di Sistemi Telematici,
Elementi di Telematica, Fondamenti di Telematica. Maggio
2015

Leonard Richardson e Sam Ruby

RESTful Web Services

O'Reilly Media

Maggio 2007

Erik Wilde, "RESTful Web Services: Principles, Patterns,
Emerging Technologies", ICWE 2010, Vienna

Cesare Pautasso, "RESTful Service Design", ICWE 2010, Vienna