Understanding HTTP and REST

Oxford University

Software Engineering Programme

Dec 2013

World Wide Web

- navigating document collections
- multimedia documents
- hypertext cross-references
- hypertext markup language
- (HTML)
- hypertext transfer protocol
- (HTTP)
- Tim Berners-Lee at CERN, 1989–1992

Evolving Web

generation	access	technology	example
first	manual	browser	arbitrary HTML
second	programmatic	screen-scraper	systematic HTML
third	standardized	web service	formally described service
fourth	semantic	semantic WS	semantically described service

The *deep web* dominates the *surface web*.

HTTP

- two-way transmission of requests and responses
- layered over TCP
- essentially stateless (but. . .)
- standard extensions for security

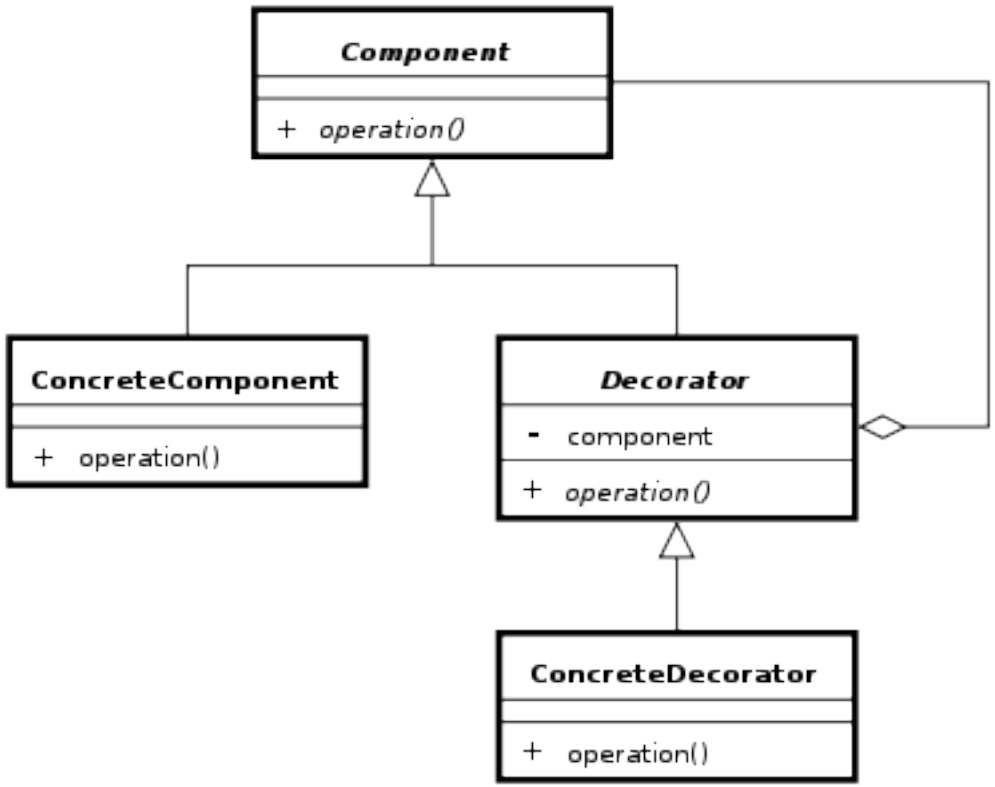
HTTP “Verbs”

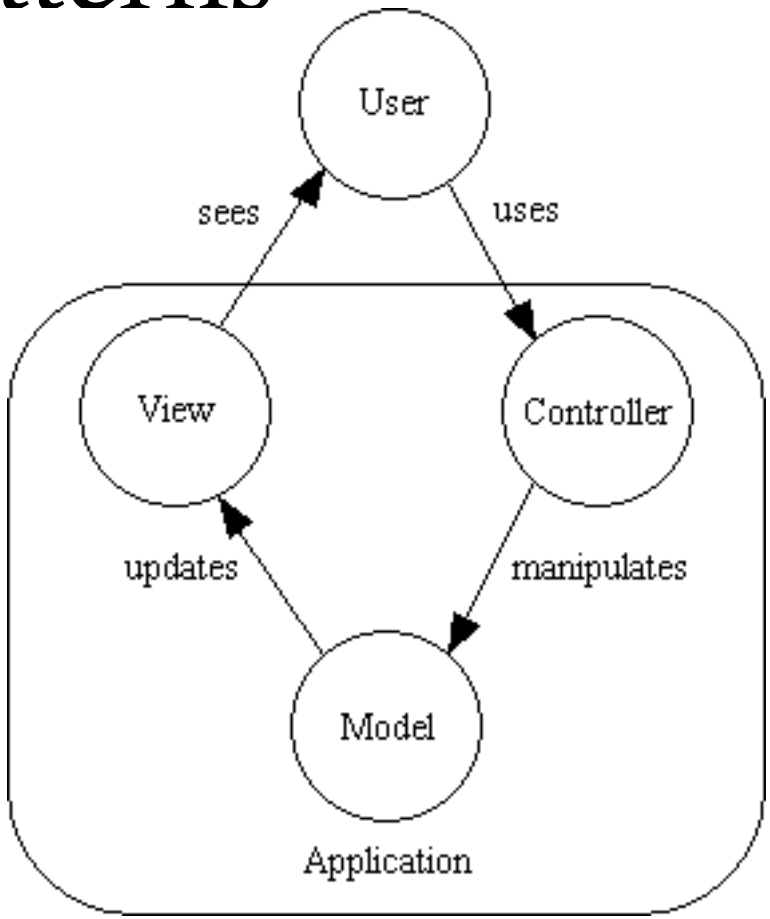
- GET uri
  - read a document; should be “safe”
- PUT uri, data
  - create or modify a resource; should be idempotent
- POST uri, data
  - create a subordinate resource
- DELETE uri
  - delete a resource; should be idempotent
- (also HEAD, TRACE, OPTIONS, CONNECT and now PATCH)

UR*

- uniform resource identifier (URI)
  - uniform resource locator (URL)
  - uniform resource name (URN)
- For a large class of schemes, the syntax is
  - <scheme>://<authority><path>?<query>
- The classical view is that URIs are partitioned into URLs (which describe a primary access mechanism, eg http:// and URNs (which do not, eg isbn:) and need a separate resolver).
- The contemporary view is that URIs may define subspaces; http: is a URI scheme, and urn:isbn: is a URN namespace. 'URL' is somewhat deprecated.

Design Patterns





REST is a design pattern

Also characterized as an Architectural Style
(aka an architecture design pattern)

REST

- Roy Fielding, a principal author of HTTP 1.1
- PhD thesis *Architectural Styles and the Design of Network-based*
- Subsequent article *Principled Design of the Modern Web Architecture* (ACM TOIT 2:2, 2002)
- Richardson & Ruby, *RESTful Web Services* architectural patterns of the web
- *Software Architectures* (2000) more about evaluation than a cookbook
- Taking HTTP seriously as a distributed computing protocol: fixed few verbs, emphasis on the nouns

Client Server (CS)

- Server offers services, listens for requests
- Client sends request, waits for response
- Transient, triggering client; persistent, reactive server
- Separation of concerns: user interface from behaviour
- Improves portability to a new user interface
- Improves scalability by simplifying components
- Improves evolvability by allowing independent evolution of components

Replicated Repository (RR) and Caching ($)

- Replicated repository: multiple servers provide same service
  - Present the illusion of a single, centralized service
  - Improves performance: latency, redundancy
  - Maintaining consistency the primary challenge
- Caching: caching responses for later reuse
  - Effectively a replication of a fragment (typically, potential data set is huge or infinite)
  - Responses explicitly or implicitly labelled cacheable or not
  - Lazy or active replication
  - Less effective than full replication, but cheaper and simpler

Stateless (S)

- Each request from client must carry all necessary context
- No session state stored on server — kept entirely on client
  - ***Resource state is a different matter***
- Improves visibility for monitoring
- Improves reliability by simplifying recovery from partial failure
- Improves scalability by allowing server to free resources quickly
- Improves evolvability by simplifying server, cache
- Decreases performance by increasing overhead

Layered Systems

- Hierarchical arrangement
- Layer provides services to layer above, uses services from layer below
- Improves evolvability and reusability through abstraction
- Decreases performance through overhead, latency
- Layered-client-server (LCS) adds proxy and gateway components to CS
- Proxy acts as shared server for one or more clients, forwarding (maybe translated) requests
- Gateway appears as normal server, but forwards (maybe translated)
- Requests to lower layers: load balancing, security

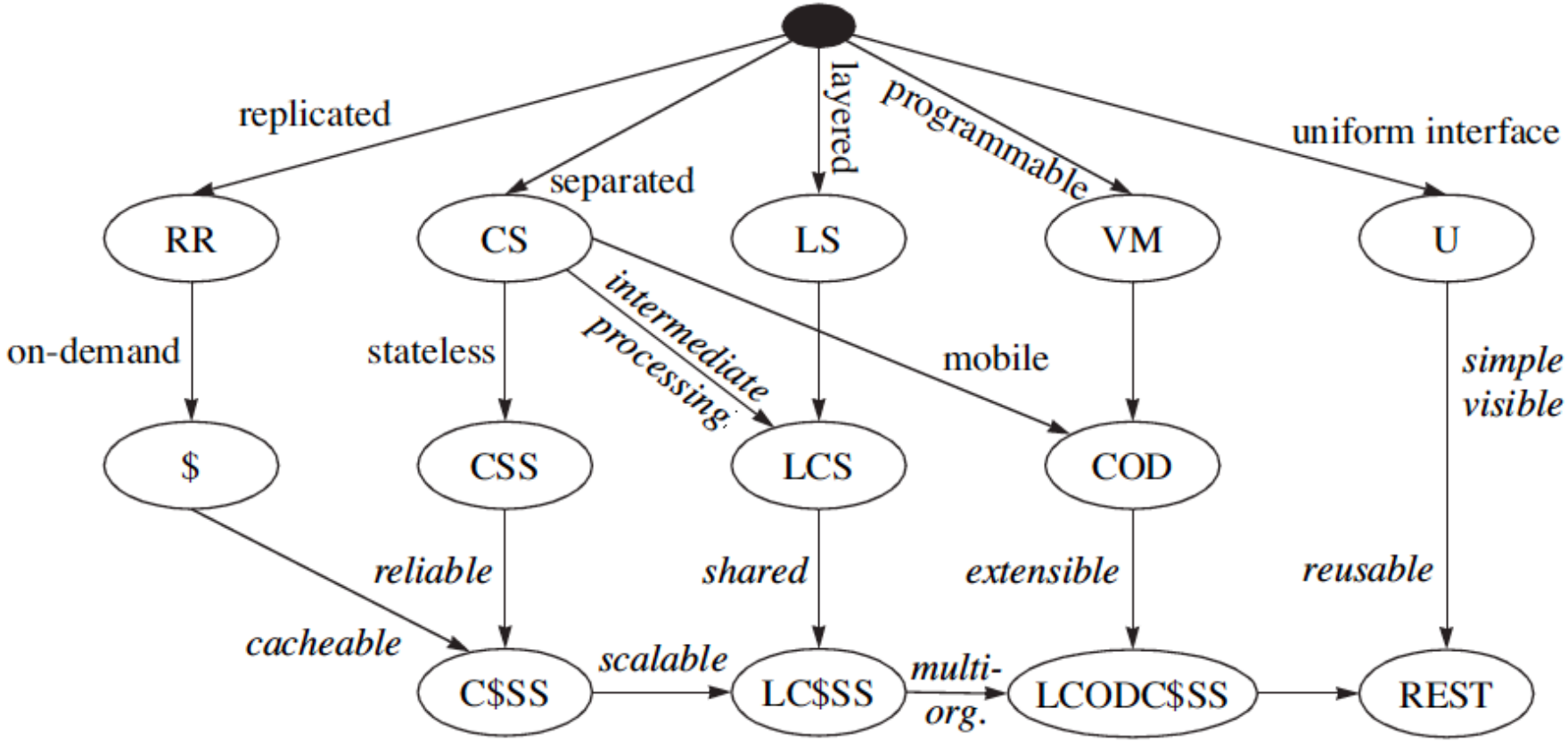
Uniform Interface

- Improves simplicity and visibility
- Decreases efficiency through possible data translations
- For REST, optimized for large-grain hypermedia data transfer
- Identification of resources
- Manipulation of resources through representations
- Self-descriptive messages
- Hypermedia as the engine of application state (more later)

Virtual Machine (VM) and Code-on-Demand (COD)

- Mobile code
- Dynamically relocate processing between data source and destination
- Improves performance by relocating code near data
- Data element must be transformed into component
- Extend client functionality by downloading applets/ scripts
- Virtual machine to provide controlled environment
- Improves simplicity and extensibility of client
- Reduces visibility
- Not a big part of REST-based SOA (yet: cf AJAX)

REST Derivation from Style Constraint



Principles of REST Architecture

- REST isn't protocol specific, but in practice means the RESTful usage of HTTP
- HTTP is actually a very rich application protocol which gives us features like content negotiation and distributed caching.
- HTTP verbs nicely map to CRUD operations of data
- RESTful web services try to leverage HTTP in its entirety using specific architectural principles.

Resources and Uniform Interface

- Addressable Resources. Every "object" on your network should have a unique ID.
- An important aspect is that each "object" or resource has its own specific URI where it can be addressed
- Anything you wish to act upon, reference, annotate, etc
- The URI should have a lifetime equivalent to the resource it represents (e.g. I've had the same bank account for 20+ years)

Representation

- State of resource captured and transferred between components
- Might be current or desired future state
- Represented as data plus metadata (name–value pairs)
- Metadata includes control data, media type
- The **Content-Type** of the resource should be useful and meaningful (self-description)
- One resource might have several representations
- Selected via separate URIs, or via content negotiation

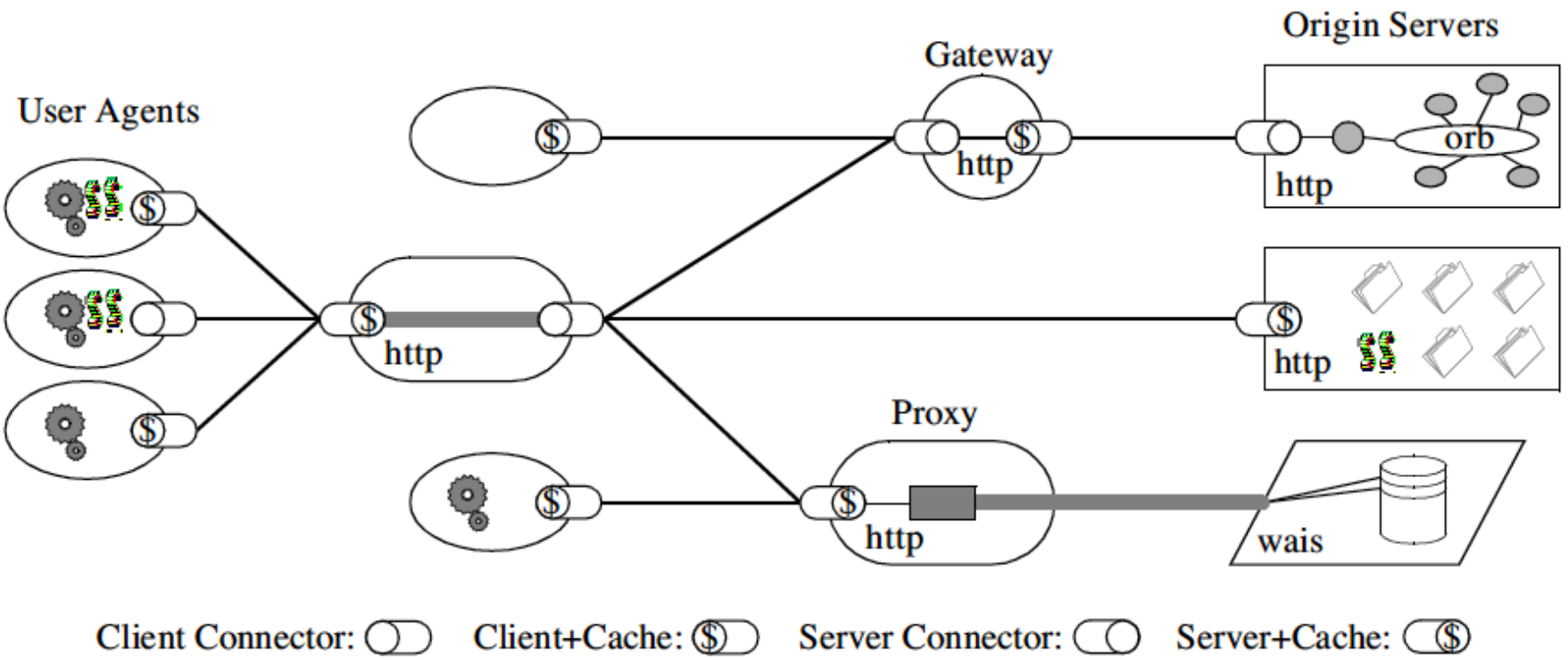
Stateless Interaction

- Abstract interface for component communication
- Stateless interactions:
  - connectors need not retain application state between requests
  - interactions can be processed in parallel, naively
  - intermediary may view and understand request in isolation
  - reusability of cached response can be determined from response itself
- Request parameters: control data, target URI, optional representation
- Response parameters: control data, optional resource metadata, optional representation

Uniform Interface

- A Uniform, Constrained Interface. When applying REST over HTTP, stick to the methods provided by the protocol
  - GET, POST, PUT, and DELETE.
- These should be used properly
  - GET should have no side effects or change on state
  - PUT should update the resource “in-place”

REST Architecture



Resource Oriented Architecture

- Resource-oriented architecture
  - after Richardson & Ruby, RESTful WS
  - action identified in HTTP method, not in payload
  - scoping information in URI
- GET reports/open-bugs HTTP/1.1
  - in contrast to RPC-style interaction
- POST /rpc HTTP/1.1
  Host: www.upcdatabase.com
  <?xml version="1.0”>
  <methodCall>
  <methodName>lookupUPC</methodName> …
  </methodCall>
- . . . or hybrid
  - http://www.flickr.com/services/rest?method=search&tags=cat

PUT vs POST

- PUT vs POST
  - creation by either PUT to new URI or POST to existing URI
  - typically, create a subordinate resource with a POST to its parent
- use PUT when client chooses URI; use POST when server chooses
- successful POST returns code 201 ‘Created’ with Location header
- (POST also sometimes used for form submission, but this can be non-uniform)

Resource Representations and States

- Interact with services using representations of resources.
  - An XML representation
  - A JSON representation
- An object referenced by one URI can have different formats available. Different platforms need different formats.
  - A mobile application may need JSON
  - A Java application may need XML.
- Utilize the Content-Type header
  - And the Accept: header
- Communicate in a stateless manner
  - Stateless applications are far more scaleable

Hypertext as the Engine of Application State

- Resources are identified by URIs

↓

- Clients communicate with resources via requests using a
  - standard set of methods

↓

- Requests and responses contain resource representations
  - in formats identified by media types

↓

- Responses contain URIs that link to further resources

↓

Beginning

Async

- HTTP is synchronous: request–response
  - What about long-running requests? deferred synchronous interaction
- Client POSTs request (because not idempotent)
- POST /queue HTTP/1.1
- Host: jobservice.com
  - Please tell me whether 2^43,112,609 - 1 is prime
  - server queues task, returns code 202 'Accepted' with
    - URI Location: http://jobservice.com/queue/job11a4f9
    - 202 Accepted
- Client polls resource:
  - GET /queue/job11a4f9 HTTP/1.1
  - getting either status report or result
- Of course WebSockets could be used to push the response
  - Also see new Push API from W3C

URI Design

- URIs should be meaningful and well-structured
- Some believe client should be able to construct URI to access a resource (increases surface area)
- Others say URIs should be opaque!
  - Discuss?!
- Use paths to separate elements of hierarchy, general to specific
- use punctuation to separate items at same hierarchical level
  - commas when order matters (eg coordinates), semicolons otherwise
  - use query variables only for 'arguments'
- URIs denote resources, not operations (unless the operation is itself something you might CRUD)

REST Standards

- **HTTP 1.1**
- URI
- URI Template
- **WebSockets**
- **XML, JSON, etc**
- Atom/AtomPub
- OData
- OpenId
- **OAuth 1 / 2**
- **SAML/SAML2**
- JSON Web Tokens
- WADL
- Swagger
- Json Home
- Json Web Encryption
- Json Web Signature
- Json Patch
- SPDY
- HTTPbis
- HTTP Link Header
- Microformats
- RDDL
- …

Quick look at the Sample Service

JSON

- A simple notation that originated in JavaScript

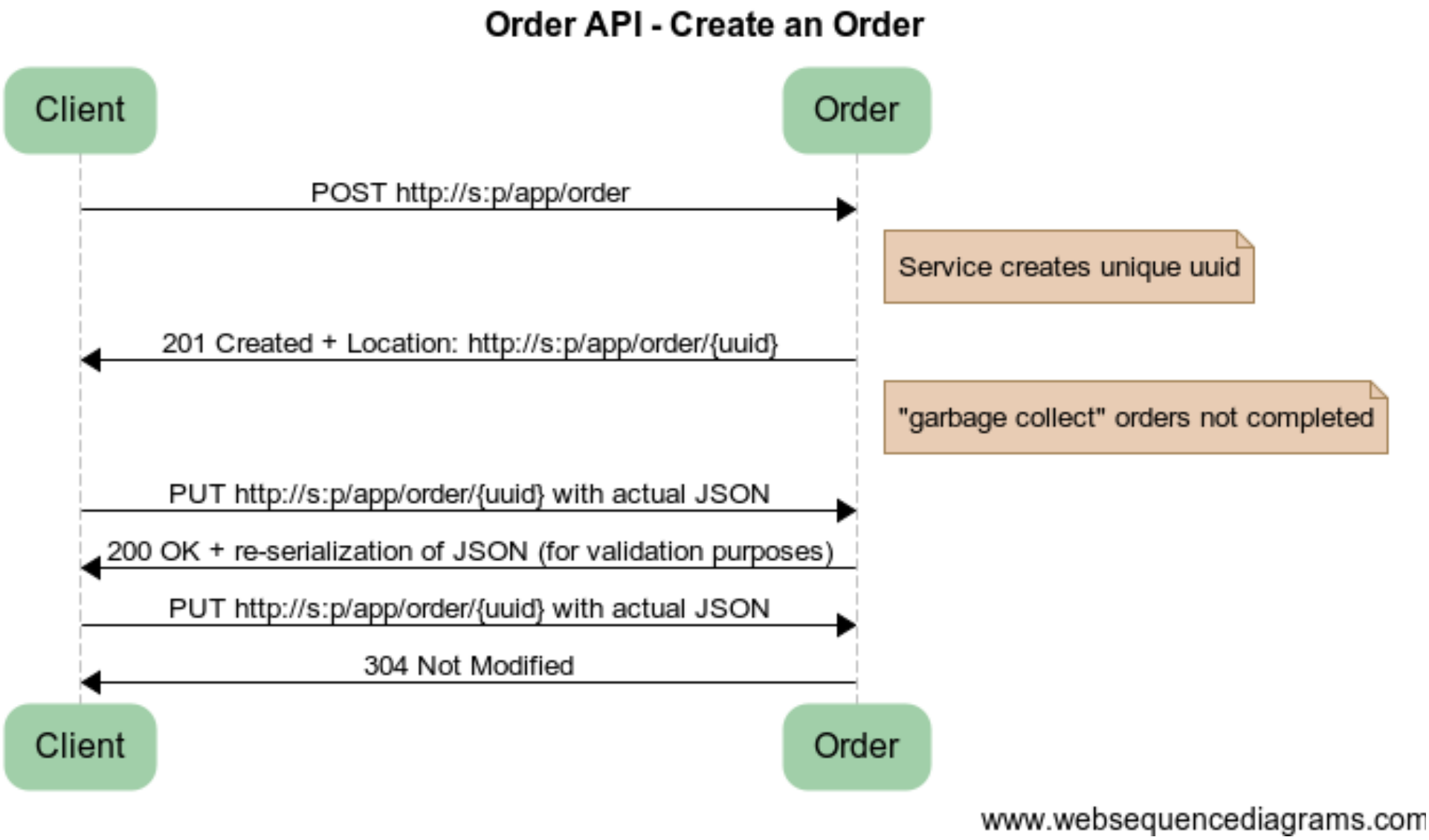
```
var x = {a:1, b:2, c:3}
```

- equivalent to:

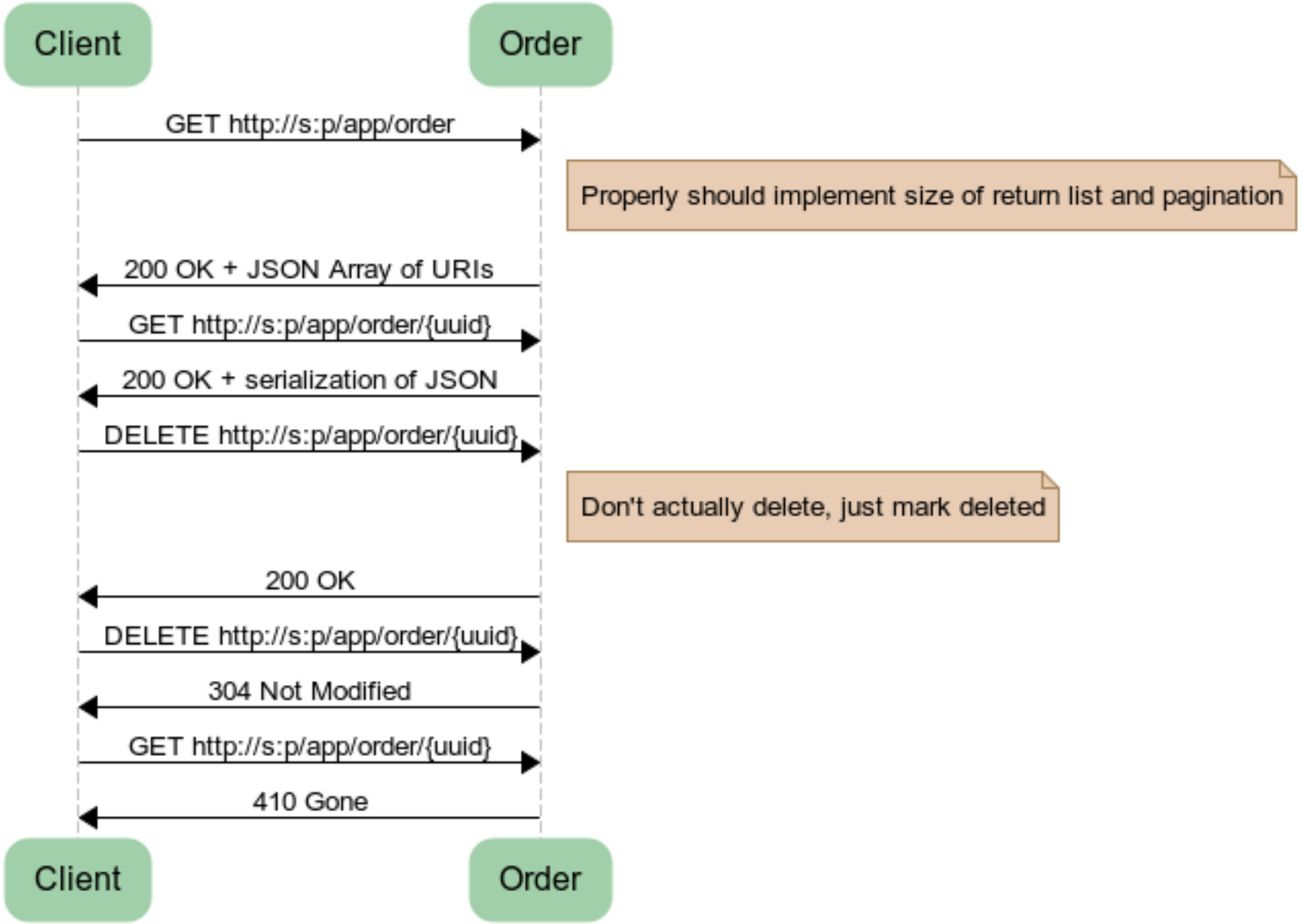
```
x.a = 1; x.b = 2; x.c = 3
```

- Can be done "dynamically"

```
var x = "{a:1, b:2, c:3}"
// imagine this actually
// comes from a webserver
var z = eval('('+x+')')
assert(z.a == 1)
```



Order API - Deal with an Order



HOW TO INSULT A DEVELOPER

Questions?