Understanding HTTP and REST

Oxford University
Software Engineering Programme
April 2021

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

HTTP

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

```
127.0.0.1:40816: clientconnect
127.0.0.1 GET http://localhost:8080/
    host: localhost:8000
    accept-encoding: gzip, deflate
    user-agent: Python-httplib2/0.9.2 (gzip)

 << 200 OK 13B
    X-Powered-By: Express
    Content-Type: application/json; charset=utf-8
    content-length: 13
    ETag: W/"d-95lxyDUPrXs/bUPZHxxiWQ"
    Date: Mon, 20 Jun 2016 09:19:05 GMT
    Connection: keep-alive

    {
        "random": 63
    }

127.0.0.1:40816: clientdisconnect
```

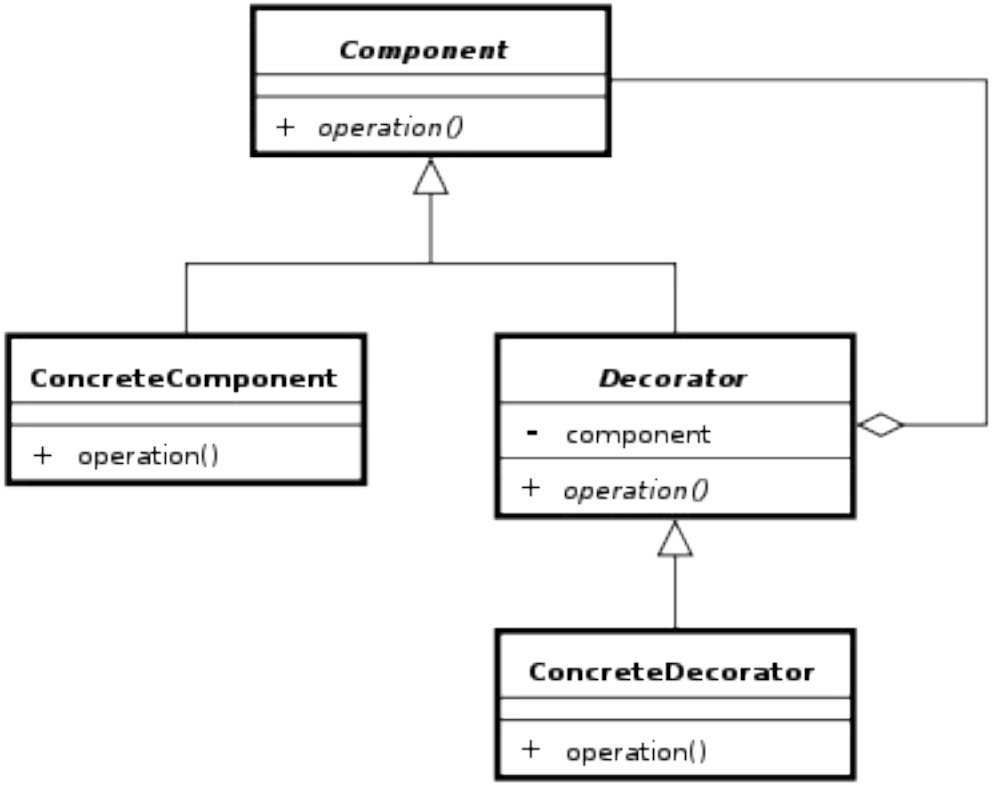
HTTP “Verbs”

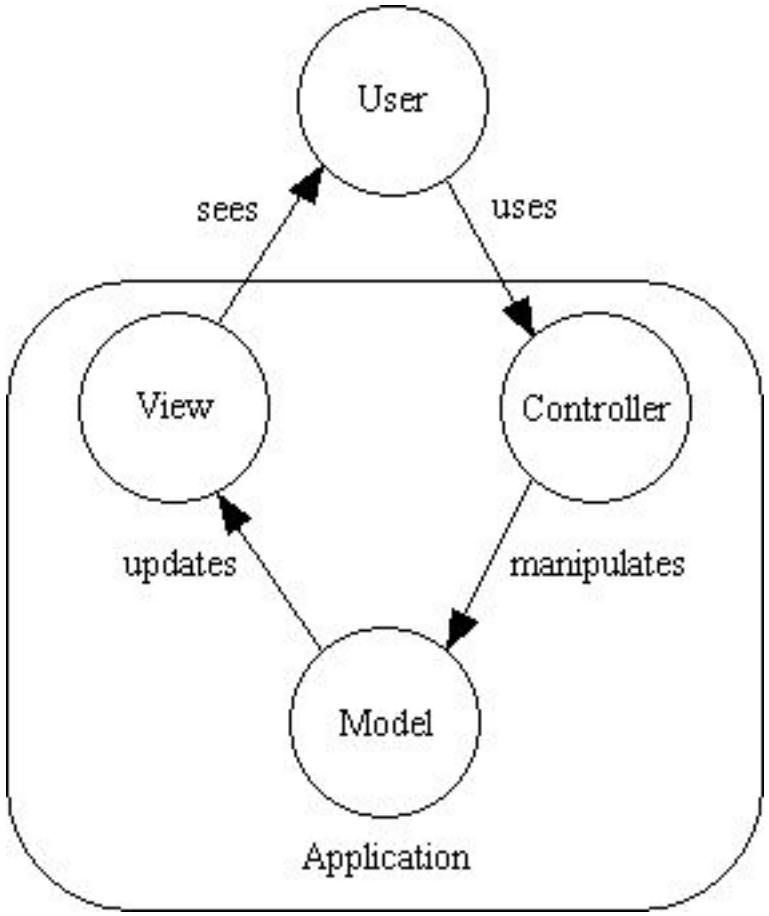
- GET uri
  - read a document; should be “safe”
- PUT uri, data
  - create or modify a resource; should be idempotent
- POST uri, data
  - create a subordinate resource
- DELETE uri
  - delete a resource; should be idempotent
- (also HEAD, TRACE, OPTIONS, CONNECT and now PATCH)

More VERBS

- HEAD - just get the metadata
- OPTIONS - which verbs are supported?
- PATCH - just send the updates

Examples of Design Patterns





REST is a design pattern

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

Doesn’t REST just mean using HTTP?

REST

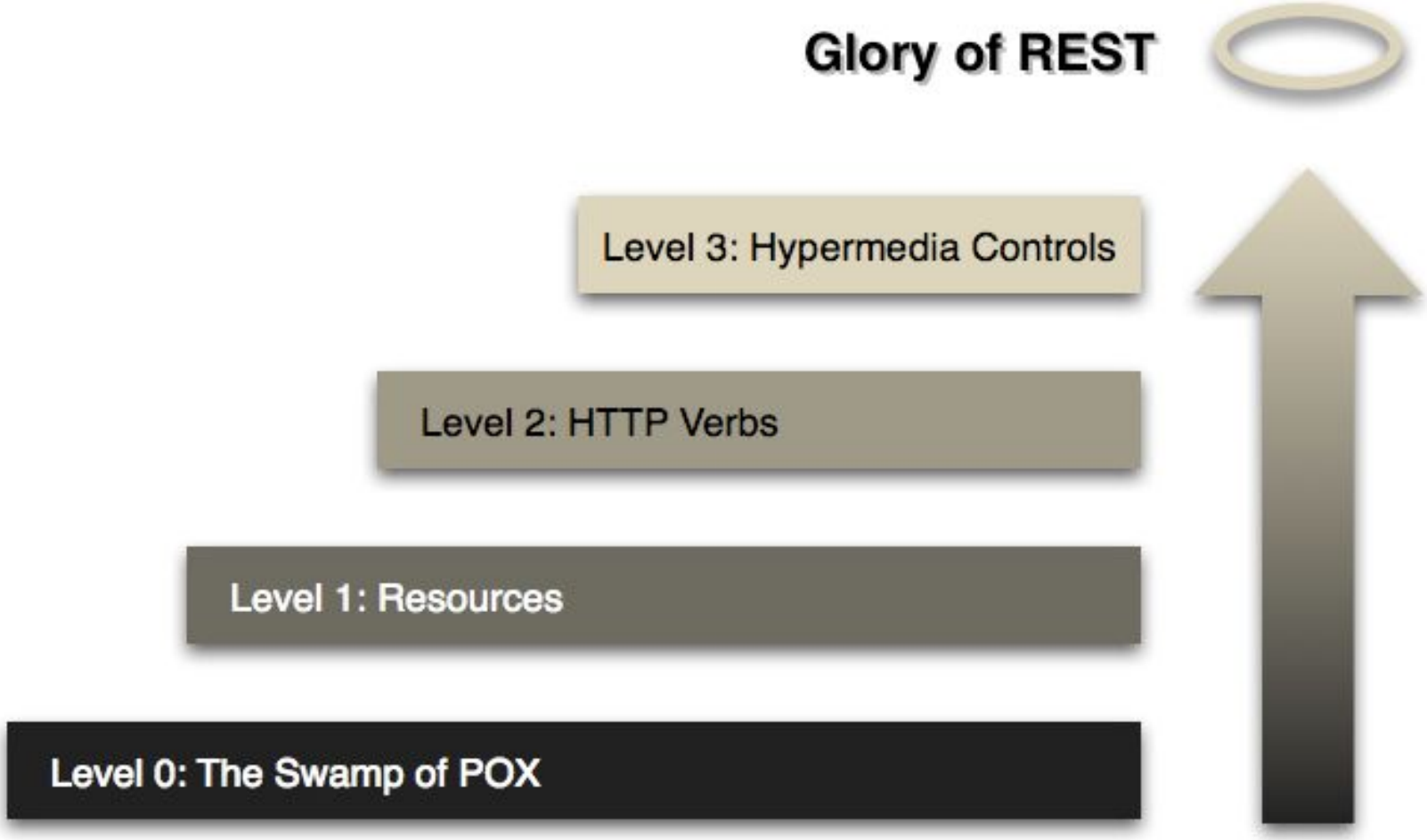
- **Roy Fielding**, a principal author of HTTP
- PhD thesis *Architectural Styles and the Design of Network-based*
- Subsequent article *Principled Design of the Modern Web Architecture* (ACM TOIT 2:2, 2002)
- Richardson & Ruby, *RESTful Web Services* architectural patterns of the web

Core ideas of REST

- "Treat HTTP seriously"

- Every "object" has a unique URL
- Use the correct "VERB":
  - GET, POST, PUT, DELETE
- Use content-types properly
- Use good HTTP return codes
- Use hyperlinks

Richardson's Maturity Model



HTTP good bad and ugly

- Good
  - GET reports/open-bugs HTTP/1.1
    - in contrast to RPC-style interaction
- Bad
  - ```
    POST /rpc HTTP/1.1
    Host: www.upcdatabase.com
    <?xml version="1.0”>
     <methodCall>
     <methodName>lookupUPC</methodName> …
    </methodCall>
    ```
- Ugly
  - http://www.flickr.com/services/rest?method=search&tags=cat
    ```

Core HTTP Verbs

- GET
  - get a representation of a resource
  - no side effects or updates to the resource
  - cacheable & idempotent
- PUT
  - update a resource
  - idempotent
- POST
  - create a new resource
- DELETE
  - remove a resource
  - idempotent

URLs for resources

http://mybank.com/account/11002123

→

```
{
    balance: 1100.10,
    transactions:
        "/account/11002123/transactions"
}
```

Hyperlinks

http://mybank.com/account/11002123/transactions/1→

```
{
    from: "http://otherbank.com/ac/50893432/",
    to: "/account/11002123"
    amount: 34.12,
    date: "1/1/2021"
}
```

PUT vs POST

- creation by either PUT to new URI or POST to existing URI
  - use PUT when client chooses URI;
  - use POST when server chooses
    - typically, create a “subordinate” resource with a POST to its parent
- successful POST returns code 201 ‘Created’ with Location header

POST example

POST /account/11002123/transactions

```
{
    to: "https://otherbank.com/ac/88819999",
    amount: "23.11",
    date: "1/1/2021"
}
```

returns
201 Created
Location: /account/11002123/transactions/2

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.
  - 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

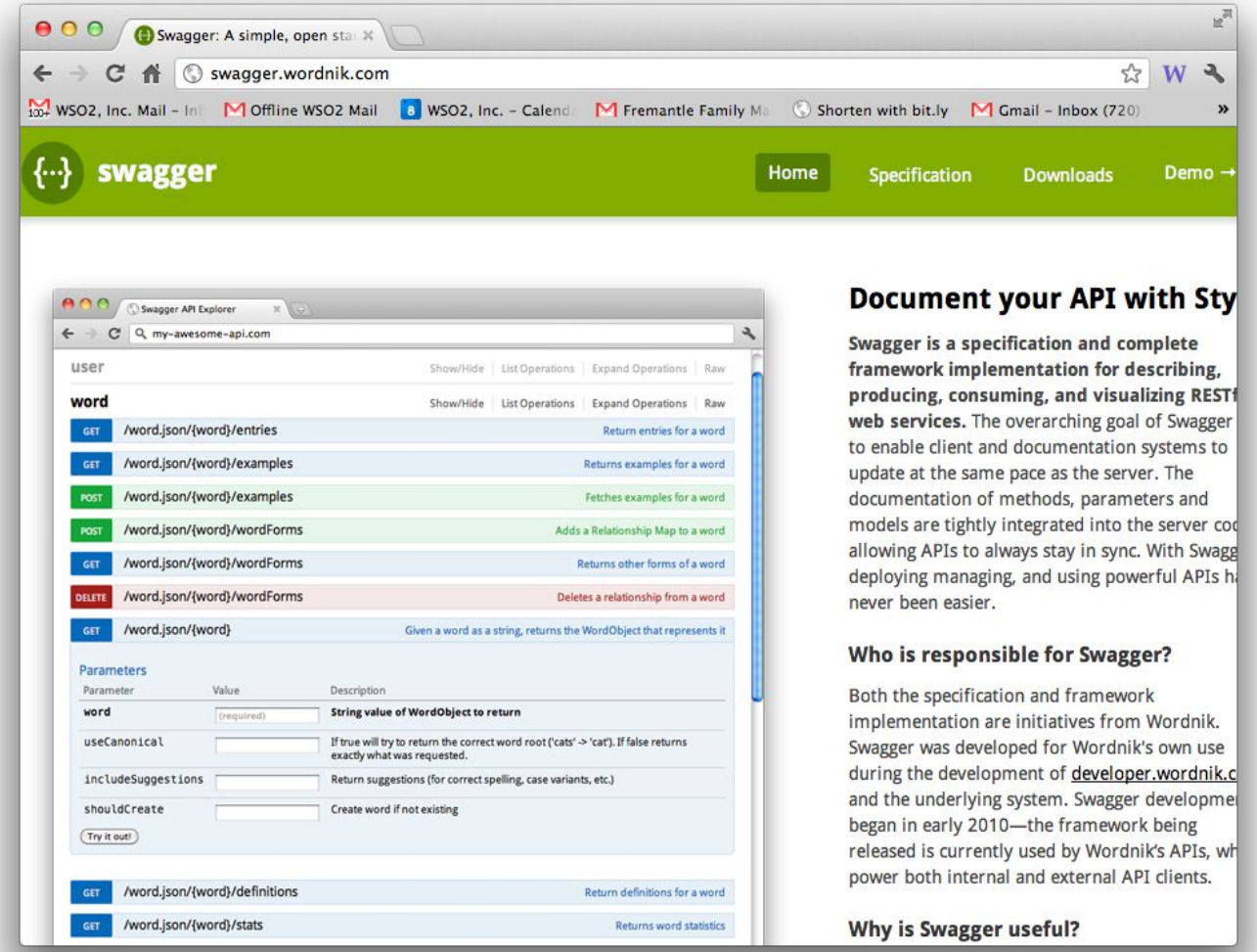
↓

- Responses contain URIs that link to further resources

↓

Beginning

REST description more later!



Return codes

- Good RESTful design means proper use of return codes...
  - Why?

HTTP return codes

	100 Continue	
	101 Switching Protocols	
Successful	200 OK	Everything is normal
Successful	201 Created	
Successful	202 Accepted	
Successful	203 Non-Authoritative Information	
Successful	204 No Content	
Successful	205 Reset Content	
Successful	206 Partial Content	
Redirection	300 Multiple Choices	
Redirection	301 Moved Permanently	Update your URL, this has moved for good.
Redirection	302 Found	
Redirection	303 See Other	
Redirection	304 Not Modified	
Redirection	305 Use Proxy	
Redirection	306 Unused	
Redirection	307 Temporary Redirect	This is temporarly moved, don't update your bookmarks.

http://www.cheat-sheets.org/saved-copy/http-response-codes-1.pdf

Client Error Codes

Client Error	400	Bad Request	Server didn't understand the URL you gave it.
	401	Unauthorized	Must be authenticated
	402	Payment Required	Not used really
	403	Forbidden	Server refuses to give you a file, authentication won't help
	404	Not Found	A file doesn't exist at that address
	405	Method Not Allowed	
	406	Not Acceptable	
	407	Proxy Authentication Required	
	408	Request Timeout	Browser took too long to request something
	409	Conflict	
	410	Gone	
	411	Lengh Required	
	412	Precondition Failed	
	413	Reqeust Entity Too Large	
	415	Unsupported Media Type	
	416	Request Range Not Satisfiable	
	417	Expectation Failed	

Server Error Codes

Server Error	500 Internal Server Error	Something on the server didn't work right.
	501 Not Implemented	
	502 Bad Gateway	
	503 Service Unavailable	Too busy to respond to a client
	504 Gateway Timeout	
	505 HTTP Version Not Supported	

Implementing REST

Just do it?!

A good answer if you already know

- HTTP coding
- JSON
- etc

Why use a framework?

- Routing
  - Separate logic for different verbs, paths, content-types
- Cacheing and content negotiation
- Data format manipulation
  - Translation to/from JSON
- Readability

Labs approach

Express + tsoa

- Built in routing, controllers, structure
- Caching, etags
- Decorations/annotations
- Swagger generation
- Security / Authentication

Clients?

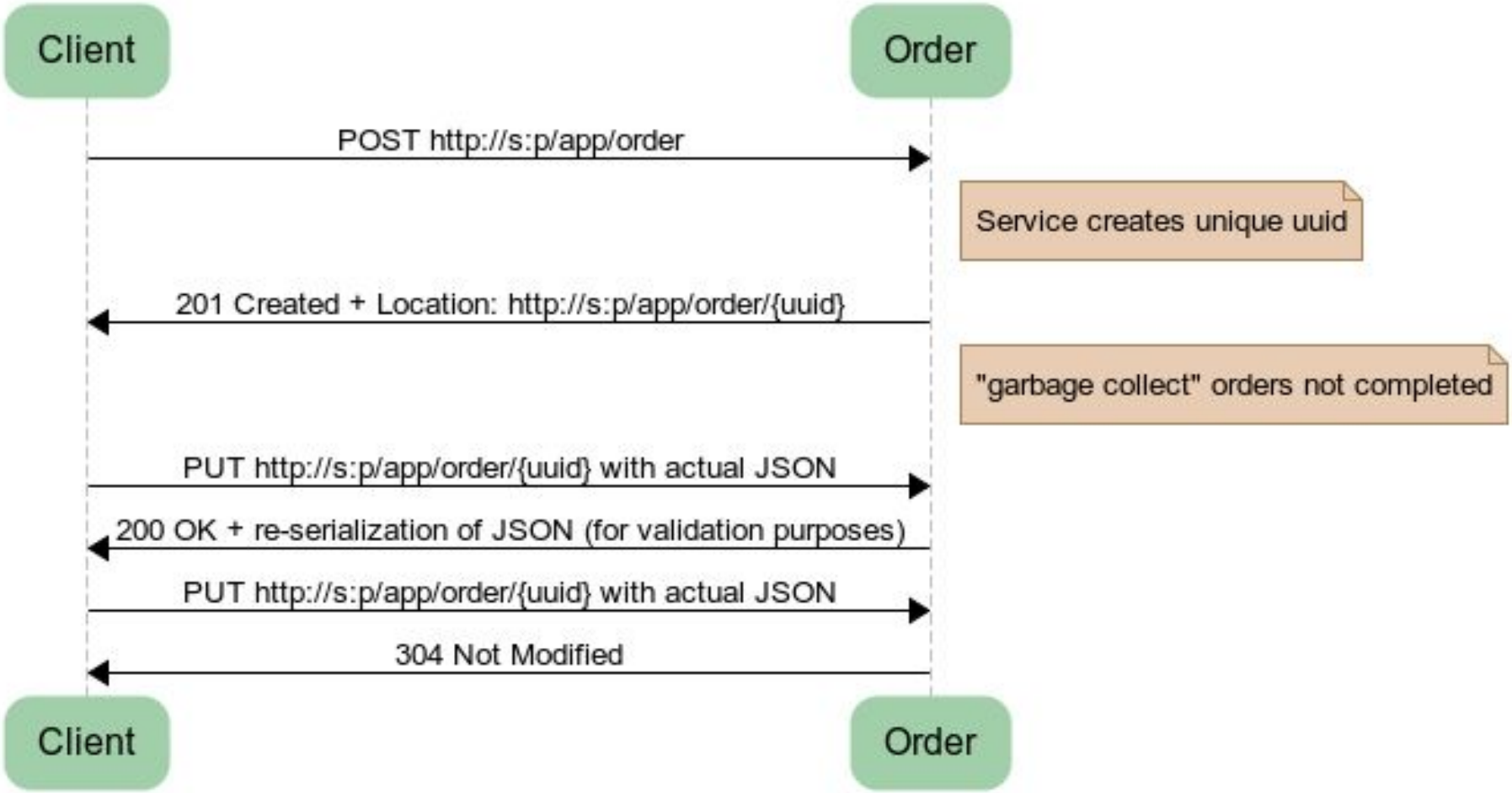
- Typically HTTP clients
- OpenAPI/Swagger can help you generate client code automatically
- Multiple choices again
  - Python: httplib2, http.client, requests
  - Node: http, axios, superagent, etc
  - Java: JAXRS, Apache HTTPClient, builtin

Summary

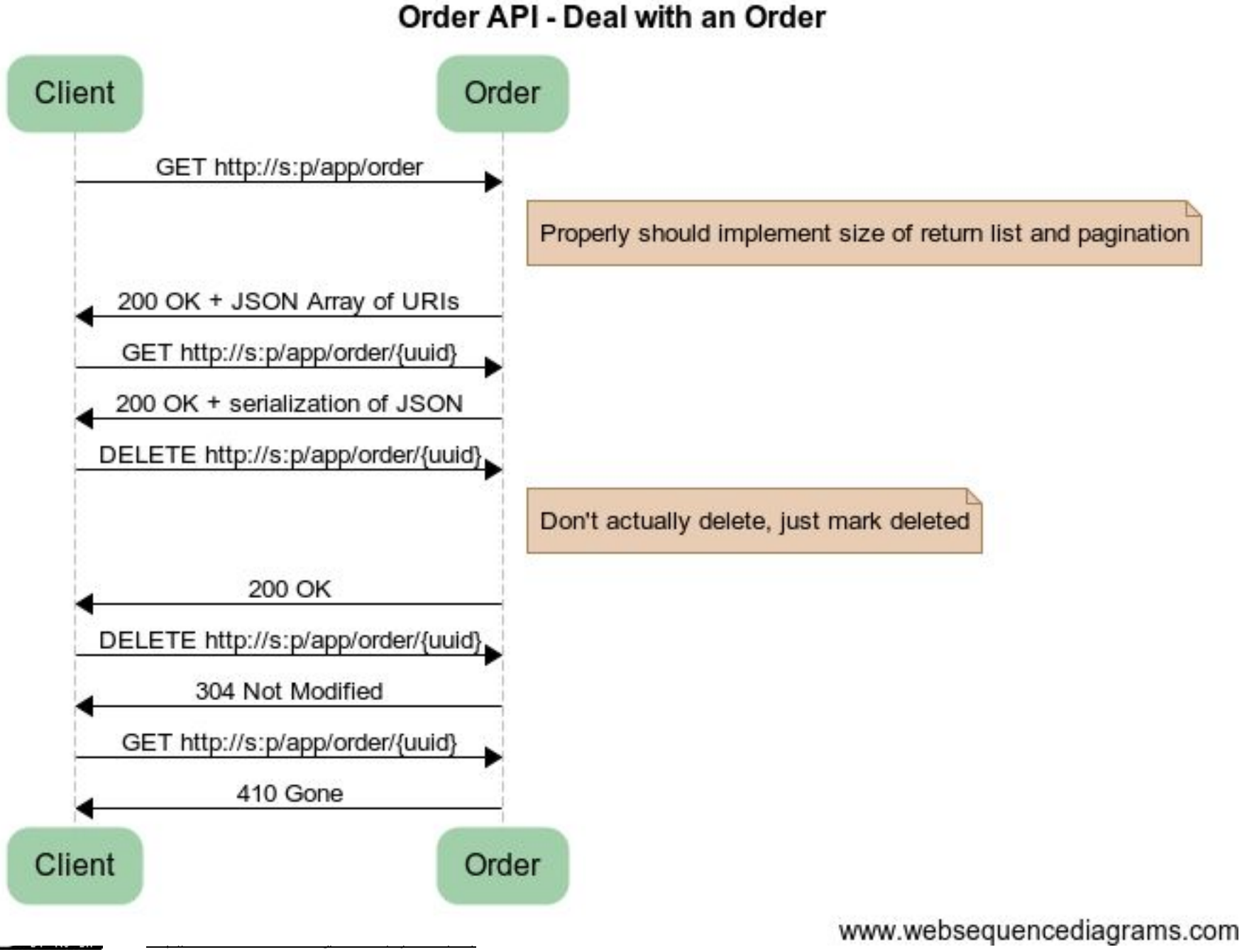
- Basic REST concepts:
  - Use the right VERB
  - Use the right return code
  - Use well defined media types
    - Resource representation
  - Use hyperlinks for HATEOAS

Our sample Purchase service

Order API - Create an Order



www.websequencediagrams.com


Order API - Deal with an Order
Client
Order
GET http://s:p/app/order
Properly should implement size of return list and pagination
200 OK + JSON Array of URIs
GET http://s:p/app/order/{uuid}
200 OK + serialization of JSON
DELETE http://s:p/app/order/{uuid}
Don't actually delete, just mark deleted
200 OK
DELETE http://s:p/app/order/{uuid}
304 Not Modified
GET http://s:p/app/order/{uuid}
410 Gone
Client
Order
www.websequencediagrams.com

Questions?