Controller/server communication

CPEN320

REST Architecture and RESTful APIs

Outline

- **What is REST ?**
- HTTP and REST
- RestFul APIs

REST

- REST - **re**presentational **s**tate **t**ransfer
- Guidelines for web app to server communications
- 2000 PhD dissertation that was highly impactful
    - Trend at the time was complex Remote Procedure Calls (RPCs) system
    - Became a must have thing: Do you have a REST API?

So what’s this REST thing ?

So what’s this REST thing ?

- REST is what you’ve been doing already in web applications. Example: accessing a URL
  - It’s an **architectural style**, NOT a standard
  - Set of **design principles** and **constraints** that characterize web-based client/server interactions

Why REST ?

- Performance
- Scalability
- Simplicity of interfaces
- Modifiability of components to meet changing needs
- Visibility of communication between components by service agents
- Portability of components by moving program code with the data
- Reliability or the resistance to failure at the system level

The six principles of REST style

- Client-Server
- Statelessness
- Cacheable
- Layered System
- **Uniform Interface (this is very important)**
- Code on Demand (Optional)

Client-Server

- Clear separation between clients and servers
- Servers and clients can be replaced and developed independently as long as the interface between them is not altered

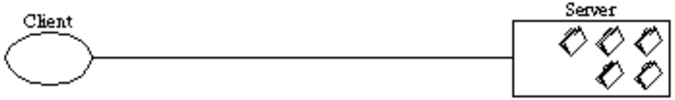


Figure 5-2. Client-Server

Stateless

- The server doesn't know about the client's application state – passed in by the client

- Server is **replaceable** and can pass session state to another server or database

- Pass representations around to change the state
  - Representation must contain all the needed info

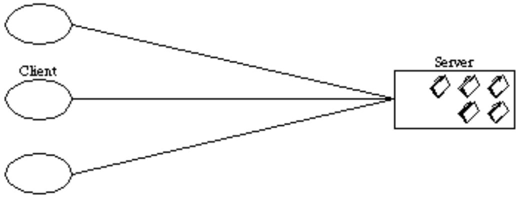


Figure 5-3. Client-Stateless-Server

Cacheable

- Caching improves performance but can compromise freshness
- Responses are assumed to be cacheable by default
- If the response does not wish to be cached, it must explicitly mark itself as such

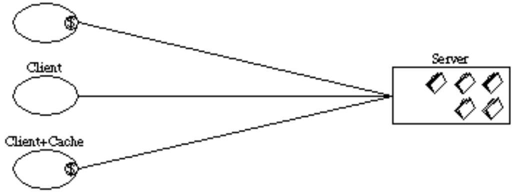


Figure 5-4. Client-Cache-Stateless-Server

Uniform Interface

- Identification of resources
- Manipulation of resources through these representations
- Self-descriptive messages
- **hypermedia** as the engine of application state

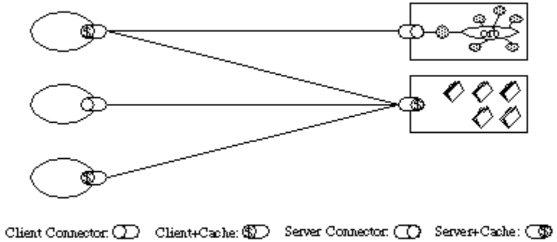


Figure 5-6. Uniform-Client-Cache-Stateless-Server

Layered System

- Client should not be able to tell if it is directly connected to server or through an intermediary (e.g., proxy, firewall etc)
- Allows scalability, e.g., through load balancing
- Security policies may be applied at proxy

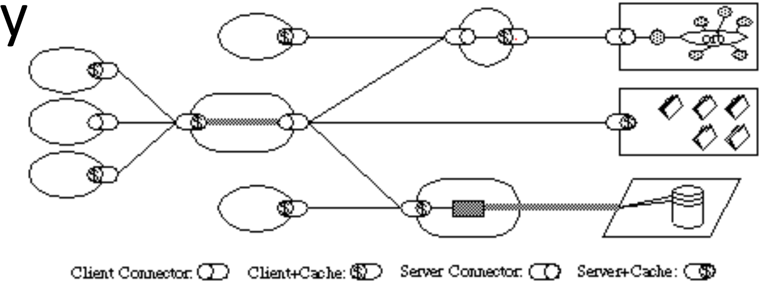


Figure 5-7. Uniform-Layered-Client-Cache-Stateless-Server

Code on Demand

- This is the only optional principle

- Extend functionality of client by transferring logic (code) to the client side

- Examples are JavaScript code, Java Applets

REST Derivation

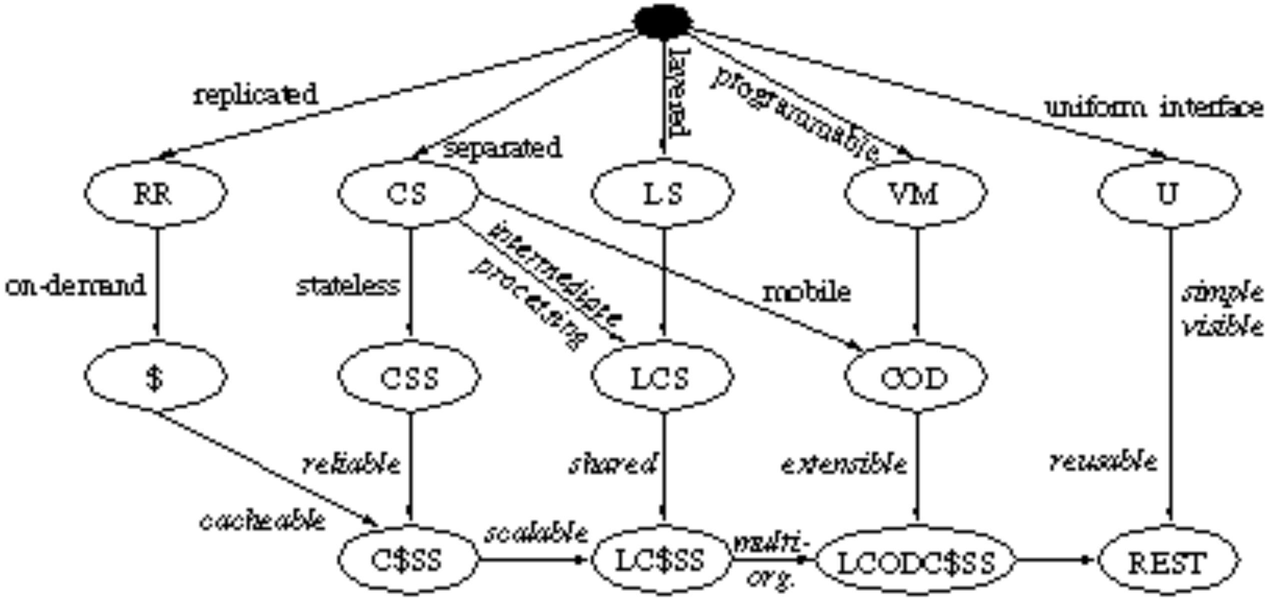


Figure 5-9. REST Derivation by Style Constraints

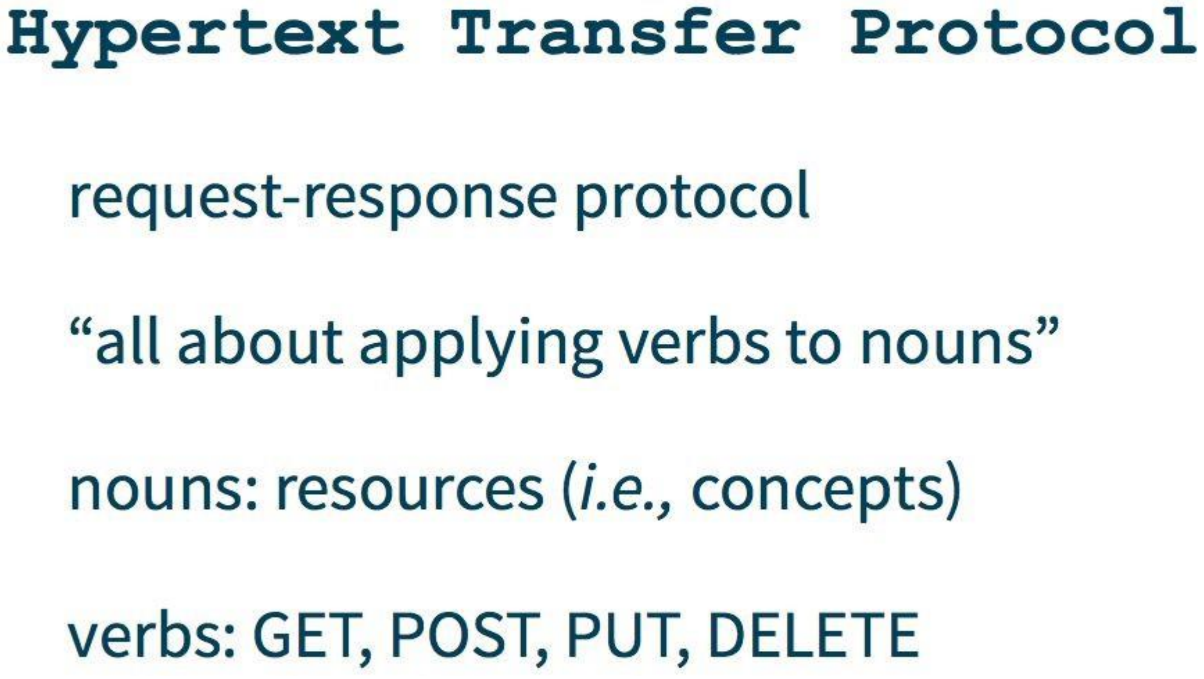
Outline

- What is REST ?

- **HTTP and REST**

- RestFul APIs

HTTP



Hypertext Transfer Protocol

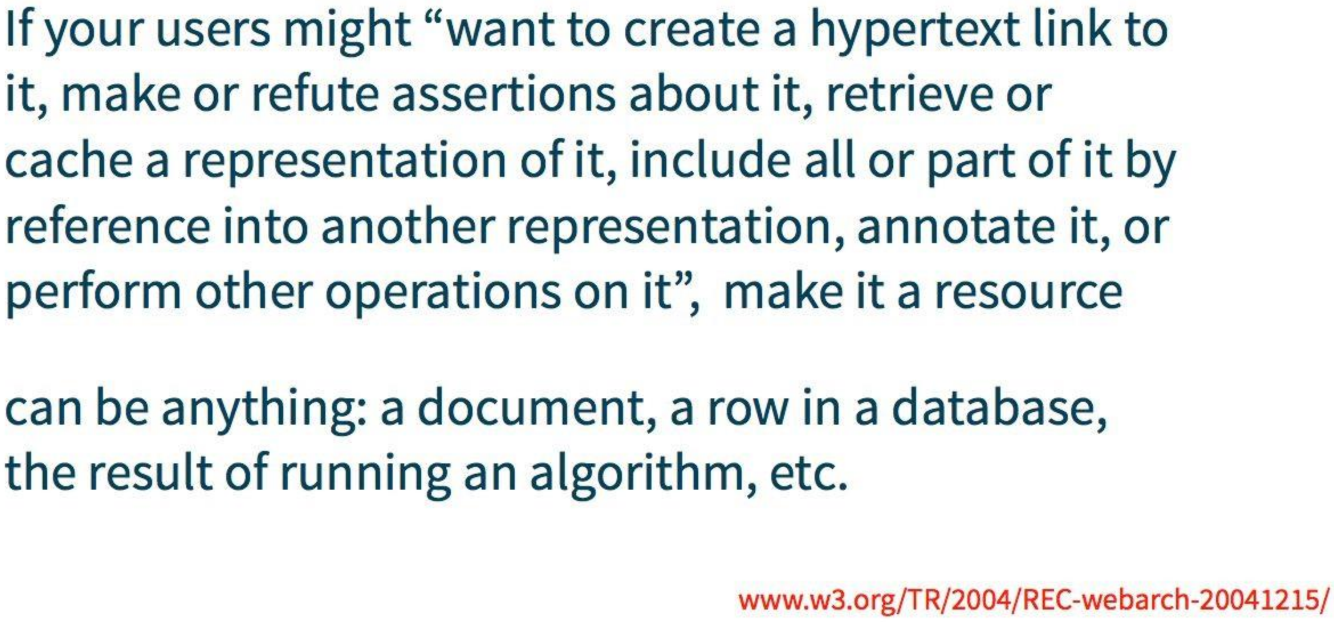
request-response protocol

“all about applying verbs to nouns”

nouns: resources (*i.e.*, concepts)

verbs: GET, POST, PUT, DELETE

Resources



If your users might “want to create a hypertext link to it, make or refute assertions about it, retrieve or cache a representation of it, include all or part of it by reference into another representation, annotate it, or perform other operations on it”, make it a resource

can be anything: a document, a row in a database, the result of running an algorithm, etc.

www.w3.org/TR/2004/REC-webarch-20041215/

URL

Uniform Resource Locator

every resource must have a URL

type of URI (Identifier)

specifies the location of a resource on a network

REPRESENTATION OF RESOURCES

when a client issues a GET request for a resource, server responds with representations of resources and not the resources themselves

any machine-readable document containing any information about a resource

server may send data from its database as HTML, XML, JSON, etc.

web.archive.org/web/20130116005443/http://tomayko.com/writings/rest-to-my-wife

Some RESTful API attributes

- Server should export **resources** to clients using unique names (**URIs**)
  - Example: `http://www.example.com/photo/` is a collection
  - Example: `http://www.example.com/photo/78237489` is a resource
- Keep servers "stateless"
  - Support easy load balancing across web servers
  - Allow caching of resources
- Server supports a set of HTTP methods mapping to **C**reate, **R**ead, **U**pdate, **D**elete (CRUD) on resource specified in the URL
  - POST method - Create resource
  - GET method - Read resource (list on collection)
  - PUT method - Update resource
  - DELETE method - Delete resource

Representational State Transfer

- Representations are transferred back and forth from client and server

- Server sends a representation describing the state of a resource

- Client sends a representation describing the state it would like the resource to have

Multiple Representations

- A resource can have more than one **representation**: different languages, different formats (HTML, XML, JSON)
- Client can distinguish between representations based on the value of **Content-Type** (HTTP header)
- A resource can have multiple representations— **one URL for every representation**

HTTP Methods

- Get
- Delete
- Post
- Put

- **Head**: just return the head information (e.g., content-type and length could be important to know for large files before doing a GET)

- **Patch**: applies partial modifications to a resource

- **Options**: client requests permitted communication options for a given URL or server (i.e. GET, POST, PUT, DELETE, etc.)

GET and Head Methods

- Retrieve representations of resources
- No side effects: not intended to change any resource state
- No data in request body
- Response codes: 200 (OK), 302 (Moved Permanently), 404 (Not Found)
- Safe method (i.e., does not modify any resources)
- Idempotent (called many times, same result **on the server side – in this case no result**)

Delete Method

- Destroy a resource on the server
- Success response codes: 200 (OK), 204 (No Content), 202 (Accepted)
- Not safe, but idempotent (i.e., can be called many times but will have same result on the server side – need not return the same value)
  - Why is this important ?
  - Can return 404 second time to indicate error

Post Request

- Upload data from the browser to server
  - Usually means “create a new resource,” but can be used to convey *any* kind of change: PUT, DELETE, etc.
  - Side effects are likely
- Data contained in request body
- Success response codes:
  - 201 (Created): **Location** header contains URL for created resource;
  - 202 (Accepted): new resource will be created in the future
- Neither safe nor idempotent

Put Method

- Request to modify resource state
- Success response codes:
  - 200 (OK)
  - 204 (No Content)
  - 201 (Created) – see below
- Not safe, but idempotent
- Can also be used like POST idempotent
  - Will create the resource if it does not exist (but only once)
  - URI can be chosen by the client (may be risky)
  - Not widely used in practice

Patch Method

- Representations can be big: PUTs can be inefficient
- Send the server the parts of the document you want to change
- Neither safe nor idempotent

HERE

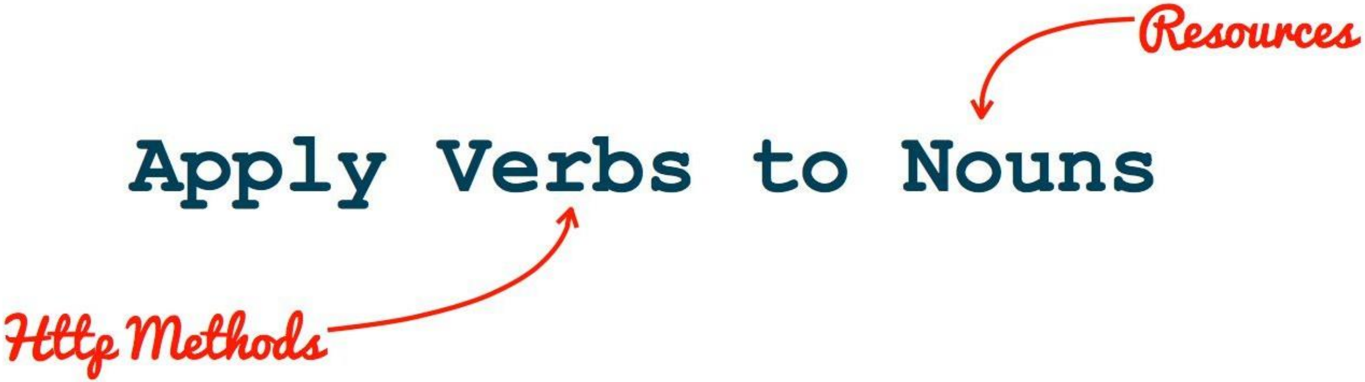
Outline

- What is REST ?
- HTTP and REST
- **RestFul APIs**

Restful APIs: Features

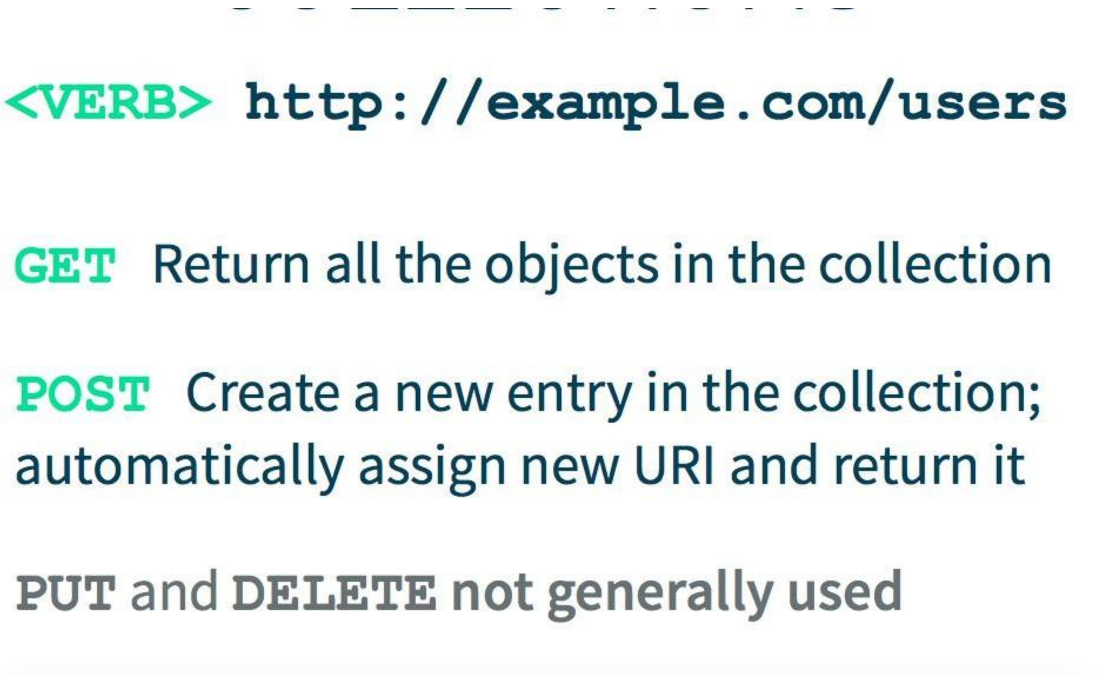
- Application program interface to a defined request-response message system between clients and servers
- Accessible via standard HTTP methods
- Request URLs that transfer representations (JSON, XML)

Designing Restful APIs



Apply Verbs to Nouns

Collections



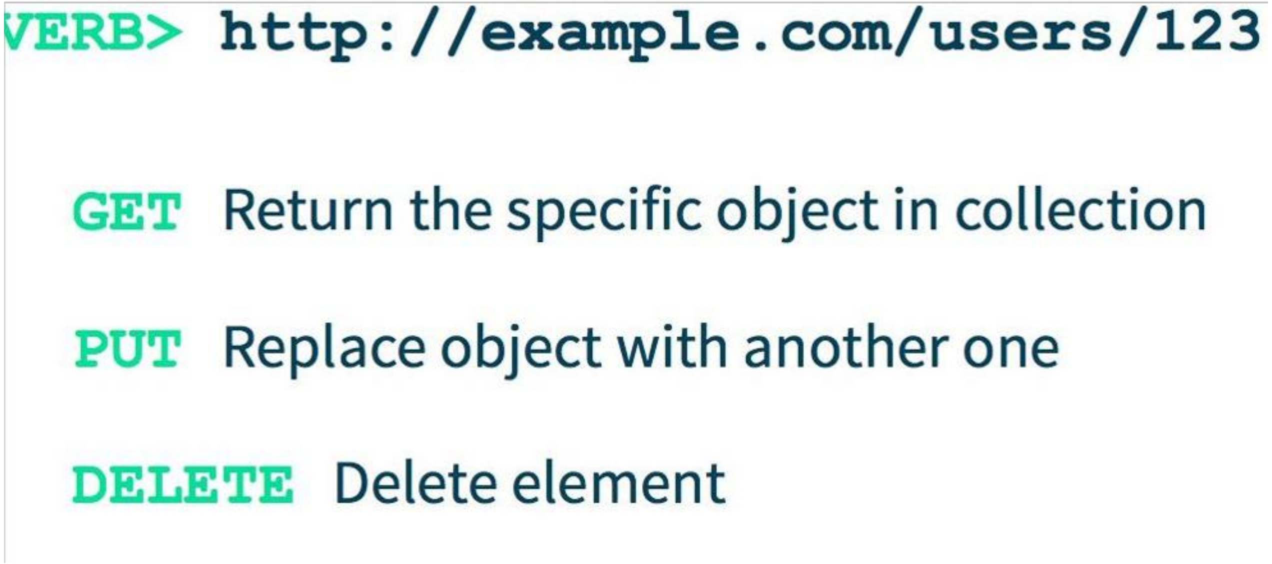
<VERB> http://example.com/users

GET Return all the objects in the collection

POST Create a new entry in the collection; automatically assign new URI and return it

PUT and **DELETE** not generally used

Elements



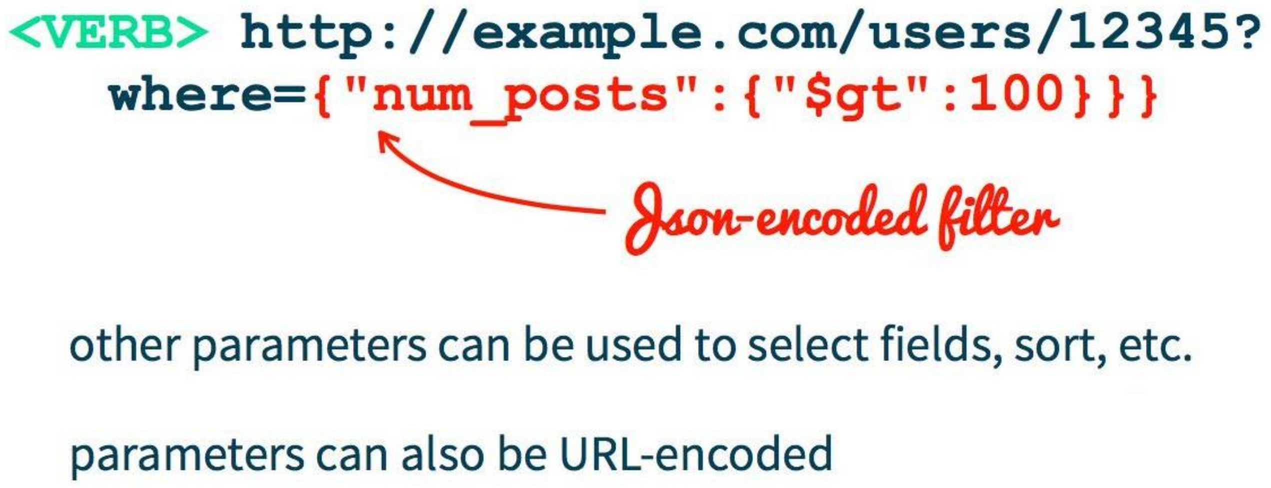
VERB> http://example.com/users/123

GET Return the specific object in collection

PUT Replace object with another one

DELETE Delete element

Using Parameters for Queries



```
<VERB> http://example.com/users/12345?
  where={"num_posts":{"$gt":100}}
```

Json-encoded filter

other parameters can be used to select fields, sort, etc.

parameters can also be URL-encoded

CheckList: Restful APIs

- Use nouns (but no verbs) as resources in URLs.
- Only expose relevant nouns
- GET method and query parameters should not alter the state (safe)
- PUT and DELETE methods should be idempotent (be applied only once on the server)

Class Activity

- Design a simple REST API to perform the following actions in a **Phonebook** application
  - Retrieve the list of all contacts in the phonebook
  - Retrieve a specific contact given their key
  - Retrieve the info of a specific contact given their first name and last name
  - Add a new contact to the phonebook
  - Modify the details of an existing contact
  - Remove a contact from the phonebook

Solution to the Activity - Retrieval

Use **nouns** rather than verbs

- To request all contacts, use
    - GET foo.com/contacts
- To request a specific contact given a key, use
    - GET foo.com/contacts/12345
- To find a contact (by first-name and last name),
    - GET foo.com/contacts?fname=“ABC”&lname=“XYZ”

Solution to the Activity – Add

Add a new contact to the phonebook

Add should be a **POST** request as it modifies the state of contacts, and is not idempotent

POST foo.com/contacts/

Send contact details in the body of the request, as JSON formatted object (say)

NOTE: We Post on the collection contacts

Solution to the Activity - Modify

Can use PUT if key is known (better than POST as it’s idempotent). Can also use PATCH for partial updates to save bandwidth, if needed.

PUT foo.com/contacts/12345

Send the new data (to be modified) in the body of the PUT request – assumes key is present

Solution to the Activity – Delete

Use Delete method in HTTP to remove the object given its key (idempotent). Should not do anything if contact is not present in server.

DELETE foo.com/contacts/12345

can also be used for multiple contacts as follows

DELETE foo.com/contacts?firstName="Jack"

OPEN API

REST API description language formerly known as "Swagger".

- Describe RESTful HTTP APIs in a machine-readable way
- Formally define a schema with endpoints of REST APIs and responses
- Communication vehicle between service providers and clients

```
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Petstore
  license:
    name: MIT
servers:
  - url: http://petstore.swagger.io/v1
paths:
  /pets:
    get:
      summary: List all pets
      operationId: listPets
      tags:
        - pets
      parameters:
        - name: limit
          in: query
          description: How many items to return at one time (max 100)
          required: false
          schema:
            type: integer
            format: int32
```

```
responses:
    200:
     description: A paged array of pets
     headers:
      x-next:
       description: A link to the next page of responses
       schema:
        type: string
     content:
      application/json:
       schema:
        $ref: "#/components/schemas/Pets"
```

```yaml
components:
  schemas:
    Pet:
      required:
        - id
        - name
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
    Pets:
      type: array
      items:
        $ref: "#/components/schemas/Pet"
```

```
components:
  schemas:
    Pet:
      required:
        - id
        - name
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
        tag:
          type: string
    Pets:
      type: array
      items:
        $ref: "#/components/schemas/Pet"
```

Resources

- https://www.openapis.org
- https://apievangelist.com
- https://speccy.io
- https://github.com/Rebilly/ReDoc
- https://openapi.tools
- https://github.com/openapitools/openapi-generator