



Distributed Systems I

Lab Introduction - part 2

REST API crash course - The bare minimum

Beshr Al Nahas

API: What, Why?

- API = Application Programming Interface
- You use API's all the time
- Web API: A set of methods exposed over the web via HTTP to allow programmatic access to applications.
- Allows you to quickly add functionality/data that others have created.
- Allows frontend developers and backend developers to agree on a common interface

Functions

- View the board
- Add a new entry
- Delete an entry

An example API

- GET /board
- POST /entries
- DELETE /entries/entryID₂

REST - An Architectural Style, Not a Standard

- HTTP-based RESTful APIs
 - Base [URL](#), such as `http://api.example.com/resources/`
 - An [internet media type](#)
 - Standard [HTTP methods](#) (e.g., OPTIONS, GET, PUT, POST, and DELETE)
- While REST is not a standard, it does use standards:
 - HTTP,
 - URL,
 - XML/HTML/GIF/JPEG/etc (Resource Representations),
 - text/xml, text/html, image/gif, image/jpeg, etc (MIME Types)

RESTful Characteristics

- Client-Server
 - a pull-based interaction style
- Stateless
 - each request from client to server must contain all the information necessary to understand the request.
- Cache
 - to improve network efficiency responses must be capable of being labeled as cacheable or non-cacheable.
- Layered components
 - intermediaries, such as proxy servers, cache servers, gateways, ... etc., can be inserted between clients and resources to support performance, security, etc.

RESTful: Uniform Interface

- Organized around *resources*
- Uniform interface
 - Resources can be accessed using a generic interface;
 - e.g., HTTP GET, POST, PUT, DELETE
- Named resources
 - Each resource has a URI.
- Resource representation:
 - Generic formats; e.g., HTML, XML, JSON, TXT,...

Functions

- View the board
- Add a new entry
- Delete an entry

An example API

- GET **/board**
- POST **/entries**
- DELETE **/entries/entryID**

RESTful: Uniform Interface

- Self-explanatory answer
 - Metadata in the request and response;
 - i.e., [HTTP status code](#) (OK 200, Not Found 404, ... etc.), Content-Type etc.
 - Example Request
 - POST /entries/1 HTTP/1.1
Host: 129.16.23.84:63100
Content-Length: 25
Content-Type: application/x-www-form-urlencoded
Accept: text/html
entry=sample+msg&delete=1

How to consume an API -- for testing code

- Terminal: curl
- Firefox: RESTClient
- Chrome: Postman
- Online: <https://www.hurl.it/>
- Examples:
 - <http://nflarrest.com/api/v1/team>
 - <http://buscentral.herokuapp.com/suggestions> POST (Feedback, message) or GET
 - <https://www.youtube.com/watch?v=suHY8dLKzCU>

Uploading Photos

This is the specification for building photo uploader applications.

It works outside the normal Flickr API framework because it involves sending binary files over the wire.

Uploading apps can call the [flickr.people.getUploadStatus](#) method in the regular API to obtain file and bandwidth limits for the user.

Uploading

Photos should be POSTed to the following URL:

```
https://up.flickr.com/services/upload/
```

Authentication

This method requires authentication with 'write' permission.

For details of how to obtain authentication tokens and how to sign calls, see the [authentication api spec](#). Note that the 'photo' parameter **should not** be included in the signature. All other POST parameters should be included when generating the signature.

Arguments

photo

The file to upload.

title (optional)

The title of the photo.

description (optional)

A description of the photo. May contain some limited HTML.

tags (optional)

A space-separated list of tags to apply to the photo.

is_public, is_friend, is_family (optional)

Set to 0 for no, 1 for yes. Specifies who can view the photo.

safety_level (optional)

Set to 1 for Safe, 2 for Moderate, or 3 for Restricted.

content_type (optional)

Set to 1 for Photo, 2 for Screenshot, or 3 for Other.

hidden (optional)

Set to 1 to keep the photo in global search results, 2 to hide from public searches.

Real example:
<https://www.flickr.com/services/api/upload.api.html>

REST:

Using the Web Browser as a GUI

- We hope you learn something cool
 - RESTful is cool (for industry)

The Web Browser as a GUI

- Web applications need integration between client side (HTML/HTTP) and server side

loading page in: 2 seconds.
461: success

Submit to board

Sample board @ 129.16.23.84:63100. Up time: 4578

ID	Entry		
1	sample msg	<input type="button" value="Modify"/>	<input type="button" value="X"/>
2	sample msg	<input type="button" value="Modify"/>	<input type="button" value="X"/>

Group members: sample author

The (distributed) board API

- Each function has a name and parameters
- REST: HTTP method + URL

Functions
View the board's contents
Retrieve entries only
Add a new entry
Retrieve one entry
Modify an entry
Delete an entry

Parameters	Returns
None	The whole board start page : html
None	List of available entries (not the full page) : html
entry : text	Status
None	The entry : html
entry : text	Status
None	Status

Sending a GET Request

- Clicking a link generates a GET request
- Dynamic update – implemented for you
 - Use a JS timer to periodically refresh the page
 - Look for: `var page_reload_timeout = 5; //in seconds`
- The user does not feel good when interrupted by page reload
 - We want to refresh a specific section of the page, i.e., the board contents
 - emulate a static GUI with dynamic contents

Sending a GET Request - Dynamic

- Partial reload using JavaScript (JS) – implemented for you
 - Sends the same request as if you press the browser reload button
 - Extracts the relevant element from the response
 - Updates the display of the HTML element with the specified ID only
 - **The server needs to tag the board contents** with the same ID; e.g.,
`<div id="boardcontents_placeholder">...</div>`

Sending a POST request - HTML forms

- **POST /entries** send new entries to the board
- HTML form
 - *Method*: HTML supports only **GET** or **POST** requests (No **DELETE** or **PUT**)
 - *Action*: the URL

```
<form id="usrform" target="noload-form-target">  
  <input type="text" name="entry" size="50" autofocus required />  
  <input type="submit" formmethod="post" formaction="/entries" value="Submit to board" />  
</form>
```

- Submitting the form generates a **post** request with **entry** in the body
entry='.....'
- If we use **GET** instead, the parameter will be appended to the URL
GET /entries?entry='.....'

After GET or POST

- The browser navigates to the new address
- and expects a response from the server
 - Good for GET most of the time since you want to see the new result
 - POST from a form results in loading a new page
 - Interrupts the user - Not nice
 - Work around:
 - Change the default behavior using JS
 - Use form target – defines where the response of the post goes to
 - Instead of loading the whole page

Sending DELETE and PUT requests

- HTML forms supports only **GET** or **POST** requests (No **DELETE** or **PUT**)
 - Use JS to send the request
 - Or for the sake of this course, change the API to use **GET** or **POST**
 - Use extra **parameters**

<u>Functions</u>	API	Parameters	Returns
Add a new entry	POST /entries	entry : text	Status
Modify an entry	PUT /entries/entryID	entry : text	Status
Delete an entry	DELETE /entries/entryID	None	Status
<u>Modify</u> or <u>Delete</u> an entry	POST /entries/entryID	entry : text <i>delete: logical</i>	Status

Client / Server HTML



1 | sample msg | Modify X

- The server sends each entry as an HTML form
 - The text of the entry itself is put into a textbox so it can be edited
 - This form contains all the parameters necessary to identify the entry

```
<form class="entryform" target="noload-form-target" method="post" action="entries/1">
  <input type="text" name="id" value="1" readonly disabled> <!-- disabled field won't be sent -->
  <input type="text" name="entry" value="msg" size="50">
  <button type="submit" name="delete" value="0">Modify</button>
  <button type="submit" name="delete" value="1">X</button>
</form>
```

- When you press the button *Modify*:
 - The HTTP header: **POST /entries/1**
 - The body of the post: **entry='msg'&delete=0**
 - Note that the parameters are separated by &

Code Skeleton

- Code Skeleton
 - Python (server code) + HTML templates (GUI)
 - The files are full of comments. Read them.
 - It is optional to use this skeleton
 - I strongly recommend it for making your (labs) life easy 😊
- Separate the core from GUI
 - For code readability: Avoid mixing core code with HTML markup
 - Use HTML templates
 - Don't waste time trying to make it look beautiful
- Use exception handling

HTML Template for use with python code

- The template is saved in external file
 - Use python format (e.g.; %d for integers, %s for strings, ... etc.)
 - Escape special characters (like %)
- Create a new entry
 - Read the template file to a variable
 - Fill the parameters in the template
 - newEntry = template % ("entries/1", 1, "First message")
- Template example

```
<form class="entryform" target="noload-form-target" method="post" action="%s">
<input type="text" name="id" value="%d" readonly disabled>
<input type="text" name="entry" value="%s" size="70%%"> <button type="submit"
name="delete" value="0">Modify</button>
<button type="submit" name="delete" value="1">X</button>
</form>
```

Tips for efficient development cycle

- Automate repetitive tasks:
 - Show boards from different servers in one window
e.g.; use frames
 - [test_multiple_instances.html](#)– update the addresses
 - To make it automatically, fill [neighborlist.txt](#) and run
`sh make_frames.sh >test_multiple_instances.html`
 - Automate sending requests to quickly find bugs
 - Use *curl* for example. See slide [7. How to consume an API](#)

HTTPServer

Python 2.7

How to make a basic server

- HTTPServer is the class running the server
 - You can extend it to manage variables
 - Single-threaded and event-driven
- BaseHTTPRequestHandler
 - Handle the requests received by the server through do_GET and do_POST
 - the server is an instance variable of the handler

Parsing POST requests

- POST requests data is usually encoded as an [application/x-www-form-urlencoded](#)
- Looks like that: *field1=this+is+the+value&field2=is+encoded*
- Parse the request to retrieve information
 - Either implement it yourself,
 - or use `parse_qs`

Connect to a HTTP Server

- HTTPConnection
- an instance of the class represents a HTTP connection with a server
- You can choose the method (GET,POST,...), and plenty of other parameters
- You should check the HTTP response code to be sure the server understood your request
- And catch exceptions!!

Demo

- Demo the server in action
- Tour of the code
- Show the source of [test_multiple_instances.html](#)

References

- *API Crash Course*, Patrick Murphy at CWU Startup Club, <http://cwustartup.com/APICrashCourse.pptx>
- *Building Web Services the REST Way*, Roger L. Costello, <http://www.xfront.com/REST-Web-Services.html>
- *REST Architecture Model: Definition, Constraints and Benefits*, Ricardo Plansky, <http://imasters.expert/rest-architecture-model-definition-constraints-benefits/>
- *API Integration in Python - Part 1*, Aaron Maxwell, <https://realpython.com/blog/python/api-integration-in-python/>
- https://en.wikipedia.org/wiki/Representational_state_transfer
- <http://www.w3schools.com/html>