Kiel University
Christian-Albrechts-Universität zu Kiel

Research Group
Distributed Systems

Distributed Systems Labs
REST API

Crash course – the bare minimum

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 /board
- DELETE /board/<entryID>

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 /board
- DELETE /board/<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

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

The App Garden

Create an App | API Documentation | Feeds | What is the App Garden?

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-seperated 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.

Example Response

When an upload is successful, the following xml is returned:

```
<photoid>1234</photoid>
```

`photoid` is the id of the new photo. **This response is formatted in the REST API response style.**

Error Codes

If the upload fails, a REST API error response is returned. The following error codes are possible:

2: No photo specified

Kiel University
Christian-Albrechts-Universität zu Kiel

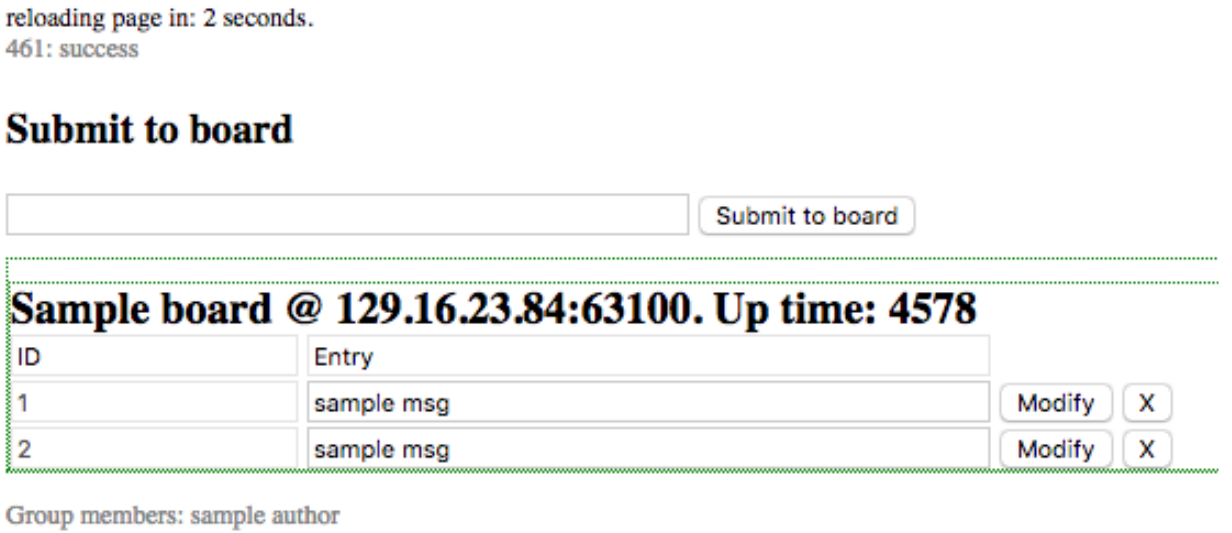
Research Group
Distributed Systems

REST: Using the Web Browser as a GUI

RESTful is cool (for industry)

The Web Browser as a GUI

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



The (distributed) board API (see lab 1 for required API)

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

Functions	Parameters	Returns
View the board's contents	None	The whole board start page : html
Retrieve entries only	None	List of available entries (not the full page) : html
Add a new entry	entry : text	Status
Retrieve one entry	None	The entry : html
Modify an entry	entry : text	Status
Delete an entry	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”>... **

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="noreload-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**

Functions	API (suggested)	Parameters	Returns (use HTTP code)
Add a new entry	POST /board	entry : text	Status
Modify an entry	PUT /board/<entryID>	entry : text	Status
Delete an entry	DELETE /board/<entryID>	None	Status
Modify or Delete an entry	*POST /board/<entryID>*	entry : text *delete: logical*	Status

Client / Server HTML

- 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="noreload-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

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

Kiel University
Christian-Albrechts-Universität zu Kiel

Research Group
Distributed Systems

Bottle web framework

for Python 2.7

Making REST API – the easy way with Bottle

- Bottle is a lightweight web framework for python (v2.7 and 3.x)
  - https://bottlepy.org/docs/dev/index.html

- Create a function serving GET requests

```
@get('/hello')
def index():
    return template('<b>Hello !</b>!', name=name)
```

- Create a function serving GET requests, with a variable in the URI

```
@get('/hello/<name>')
def index(name):
        return template('<b>Hello {{name}}</b>!', name=name)
```

Serving POST request

- Create a function serving POST requests

```
@post('/input')
def receive_post_on_input ():
    parameter_value = request.forms.get('parameter_name')
    return template(,<b>We received value {{value}} for parameter 'parameter_name'</b>! ', value= parameter_value )
```

- It’s very simple!

HTML Template for use with python code

- Bottle uses its own template engine
- You can check how templates can be modified
  - https://bottlepy.org/docs/dev/stpl.html

Kiel University
Christian-Albrechts-Universität zu Kiel

Research Group
Distributed Systems

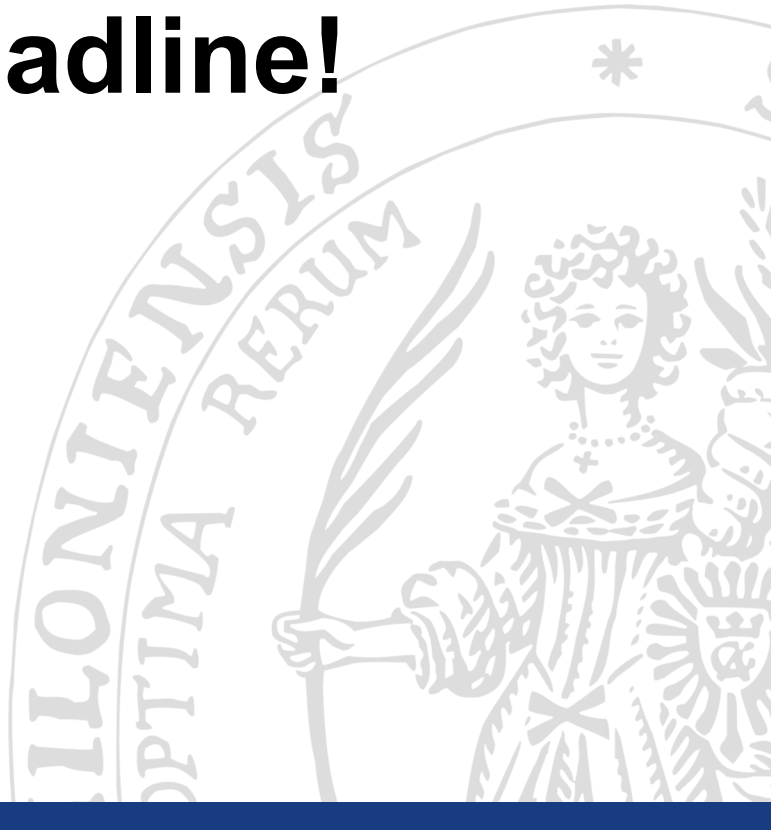
Python's Requests package

because making HTTP requests shouldn't take more than one line of code

Sending POST with Python

- We (highly) recommend you to use the Requests package
  - https://pypi.org/project/requests/
  - A nice tutorial: https://realpython.com/python-requests/
- How to send GET requests?
  - *response = requests.get(“http://uni-kiel.de”)*
  - *response.status_code == 200* means the request was successfully treated
  - *response.text* contains the HTML response
- What about POST requests?
  - *requests.post('https://httpbin.org/post', data={'key':'value’})*
  - *data* is a python dictionary!

Check iLearn for the lab deadline!



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

Credits – based on slides from

- Beshr Al Nahas
- And many others