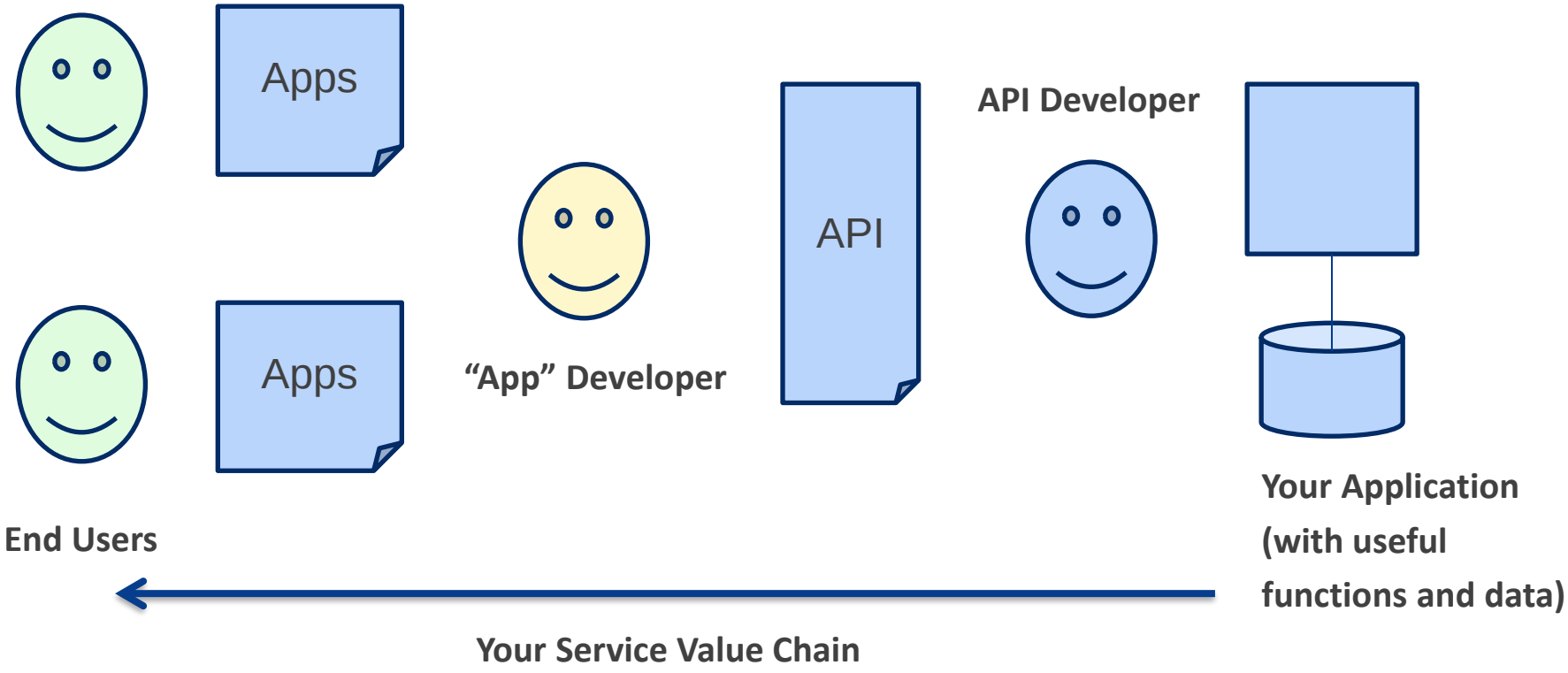
COMP9321:
Data services engineering

Week 5: Designing RESTful API and RESTful API Client

Term1, 2022

By Mortada Al-Banna

Designing an API – who should you target?



Designing RESTful APIs

A well-designed API should make it easy for the clients to understand your service without having to “study” the API documents in-depth.

self-describing, self-documenting as much as possible

the clients are developers like yourself, so probably they would like to have an API that is easy to pick up and go

The RESTful service principles actually give us a straightforward guideline for designing the Web API ...

“Clean, Clear, Consistent” are the key

URI design (= API endpoints)

Avoid using ‘www’, instead:

- https://api.twitter.com
- https://api.walmartlabs.com

Identify and “name” the resources. We want to move away from the RPC-style interface design where lots of ‘operation names’ are used in the URL

e.g., /getCoffeeOrders, /createOrder, /getOrder?id=123

Instead:

- Use nouns (preferably plurals) (e.g., orders)
  - Walmart /items /items/{id}
  - LInkedIn /people /people/{id}
  - BBC /programmes /programmes/{id}

URI Design ...

Use of Query Strings. Use it when appropriate …

Search or Select

- /orders?date=2015-04-15
- /customers?state=NSW&status=gold
- Twitter /friendships/lookup
- Walmart /search?query

Expression of the relationships between resources

- Facebook /me/photos
- Walmart /items/{id}/reviews
- /customers/123/orders vs. /orders?customer=123

Think: what is the expected resource as return in this URI?

URI design …

On the resources' URIs ... we add 'actions/verbs'

Completes the endpoints of your APIs

Should it?

Resource (URI)	GET	POST	PUT	DELETE
/coffeeOrders	get orders return a list, status code 200	new order return new order + new URI, status code 201	batch update status code (200, 204)	ERROR (?) status code (e.g., 400 - client error)
/coffeeOrders/123	get 123 return an item, status code 200	ERROR (?) return error status code (400 - client error)	update 123 updated item, status code (200, 204)	delete 123 status code (204, 200)

Note: PUT could also return 201 if the request resulted in a new resource

Decide How to Use the Status Codes

Using proper status codes, and using them consistently in your responses will help the client understand the interactions better.

The HTTP specification has a guideline for the codes ..., but at minimum:

Code	Description	When
200	OK	all good
400	Bad Request	you (client) did something wrong
500	Internal Error	we did something wrong here

And utilise more of these, but restrict the number of codes used by your API (clean/clear)

Code	Description	When
201	Created	your request created new resources
304	Not Modified	cached
404, 401, 403	Not Found, Unauthorised, Forbidden	for authentication & authorisation

RFC2616 (HTTP status codes)

https://www.w3c.org/Protocols/rfc2616/rfc2616.html

Decide your response format

Should support multiple formats and allow the client content negotiation (JSON only?)

Use simple objects.

A single result should return a single object.

Multiple results should return a collection - wrapped in a container.

/coffeeOrders/123

```
{
    “Id”: “123”,
    “type”: “latte”,
    “extra shot”: “no”,
    “payment”: {
        “date”: “2015-04-15”,
        “credit card”: “123457”
     },
    “served_by”: “mike”
}
```

/coffeeOrders

```
{
    “resultSize”: 25,
    “results”: [ {
                    “id”: “100”,
                    “type”: “latte”,
                    “extra shot”: “no”,
                    “payment”: {
                        “date”: “2015-04-15”,
                        “credit card”: “22223”
                     },
                     “served_by”: “sally”
                },
                { … },
              ]
}
```

To summarise so far:

GET /coffeeOrders/123

Success

Code = 200
OK

Send back the response (in the format requested by the client)

Failure

4xx	**Bad request**	**e.g., 404 not found**
5xx	**Processing error**	**e.g., 503 database unreachable**

Note: response body could contain detailed error messages

To summarise so far:

POST /coffeeOrders

Success

Code = 201
Created

May return:

- **Location: …/coffeeOrders/123, or**
- **the updated object in the body, or**
- **both.**

Failure

4xx	**Bad request**	**e.g., 400 missing required field**
5xx	**Processing error**	**e.g., 503 database unreachable**

Note: response body could contain detailed error messages

To summarise so far:

PUT /coffeeOrders/123

PATCH /coffeeOrders/123

Success

Code = 200,
Code=204,
Code =201

May return (optionally)

- **Location: …/coffeeOrders/123, or**
- **the new object in the body, or**
- **both.**

Failure

4xx	**Bad request**	**e.g., 400 missing required field**
5xx	**Processing error**	**e.g., 503 database unreachable**

Note: response body could contain detailed error messages

To summarise so far:

DELETE /coffeeOrders/123

Success

Code = 200,
Code = 204

May return (optionally)
Deleted resource in the body, or nothing …

Failure

4xx	**Bad request**	**e.g., 404 not found**
5xx	**Processing error**	**e.g., 503 database unreachable**

Note: response body could contain detailed error messages

Taking your API to the Next Level

HATEOAS = Hypermedia As The Engine Of Application State

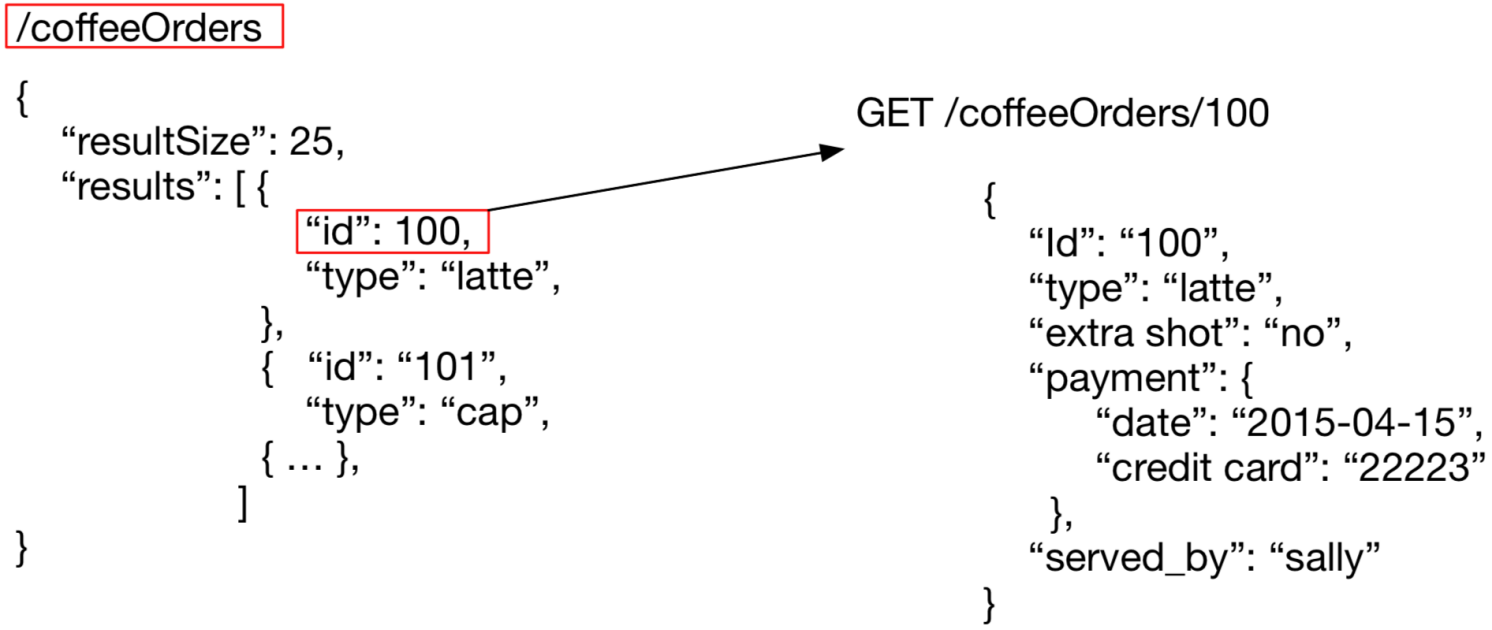
From Wikipedia: The principle is that a client interacts with a network application entirely through hypermedia provided dynamically by application servers. A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia.

Think how people interact with a Web site. No one needs to look up a manual to know how to use a Web site ... Hypermedia (i.e., documents with links to other links) itself serves as a self-explanatory guide for the users.

The HATEOAS principle aims to realise this in API design.

Not Using HATEOAS

Not implementing the links in REST API would look like this:



You assume that the client knows how to construct the next request path (i.e., combine /coffeeOrders and id:100) - maybe by reading your API document?

Using HATEOAS

/coffeeOrders

```
{
   “resultSize”: 25,
   “links”: [{
            “href”: “/coffeeOrders”
            “rel”: “self”
        },
        {  “href”: “/coffeeOrders?page=1”,
            “rel”: “alternative”
        }
        {   “href”: “/coffeeOrders?page=2”,
             “rel”: “nextPage”
        }
    ],
   “results”: [ {
                    “Id”: “123”,
                    “type”: “latte”,
                    “links”: [ {
                            “href”: “/coffeeOrders/123”,
                            “rel”: “details”
                     }]
                },
                { … },
    ]
```

/coffeeOrders/123

```
{
   “Id”: “123”,
   “type”: “latte”,
   “extra shot”: “no”,
   “payment”: {
        “date”: “2015-04-15”,
        “credit card”: “123457”
    },
   “served_by”: “mike”
   “links”: [ {
            “href”: “/coffeeOrders/123”,
            “rel”: “self”
            },
            {
            “href”: “/payments/123”,
            “rel”: “next”
            }
    ]
}
```

Using HATEOAS

Use links to:

- help the clients use the API (self-describing as possible)
- navigate paging (prev, next)
- help create new/related items
- allow retrieving associations (i.e., relationships)
- hint at possible actions (update, delete)
- evolve your workflow (e.g., adding extra step in a workflow = adding a new link)

Standard link relations: http://www.iana.org/assignments/link-relations/link-relations.xhtml

Although the principle is well-understood, how HATEOAS links are implemented (i.e., how the links appear in the responses) is different from one implementation to another ...

API Versioning

- When your API is being consumed by the world, upgrading the APIs with some breaking changes would also lead to breaking the existing products or services using your API.
- Try to include the version of your API in the path to minimize confusion of what features in each version.

Example:

http://api.yourservice.com/v1/stuff/34/things

Standarised API specification

OpenAPI Specification (Swagger, swagger.io)

- a standard, language-agnostic interface definition to RESTful APIs
- both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection.
- “Auto-documentation” (from the written specification)
- “Auto inspection/testing interface”
- “Auto client code generation”

An idea that is increasingly becoming attractive … and this helps RESTful client application development (every aspect of the standard is to help API consumer understand and consume API quicker/less labour intensive way)

cf. SOAP/WSDL - the calculator service (http://www.dneonline.com/calculator.asmx)

Standarised API specification

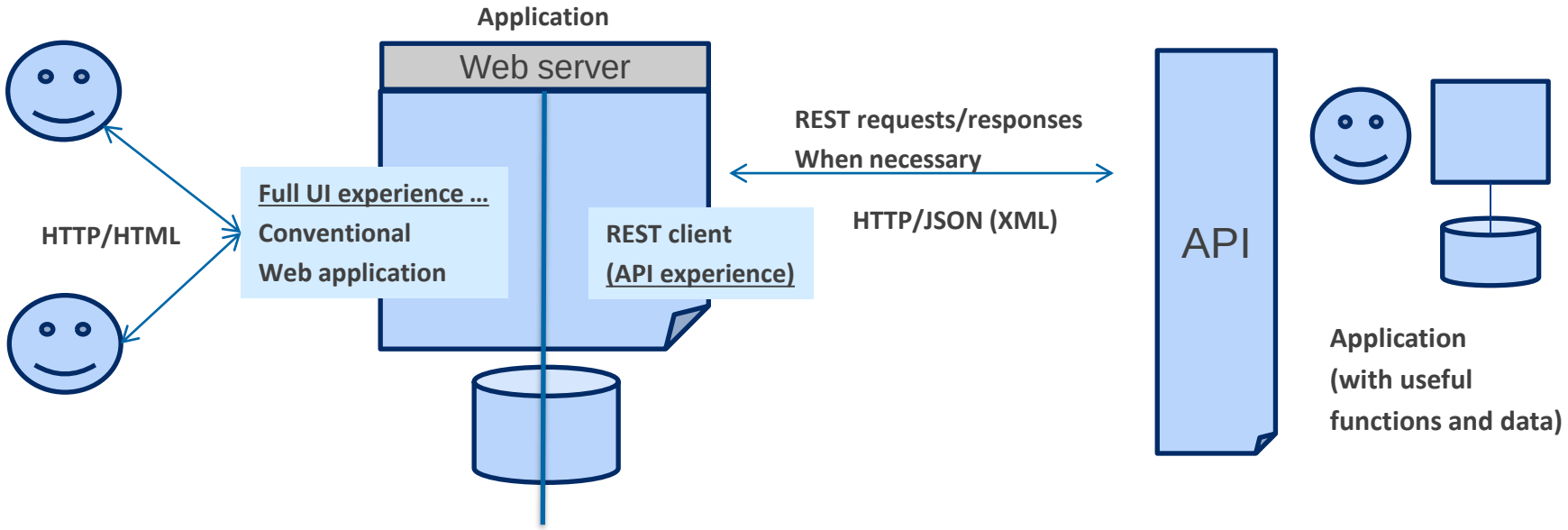
Basic structure (what is in it …)

- A quick run through: https://swagger.io/docs/specification/basic-structure/
- Petstore.yaml (https://editor.swagger.io/)

Consuming a REST API

RESTful Service Client

- The client in this situation is pre-written software program



Web and RESTful client to Services

So … an application may have ‘RESTful client’ embedded so that it can incorporate external APIs

- Understanding, getting to know an API quickly is also a “skill”
- Something like OpenAPI Specification could help here …
- Pure HTTP request/response (stateless) interactions with the API expected

Two types:

Simple programmatic interactions with the service

- Using so-called HTTP Client libraries
- Maybe less ”human-based interaction” with the response content

Increasing interactivity/responsiveness for the UI layer

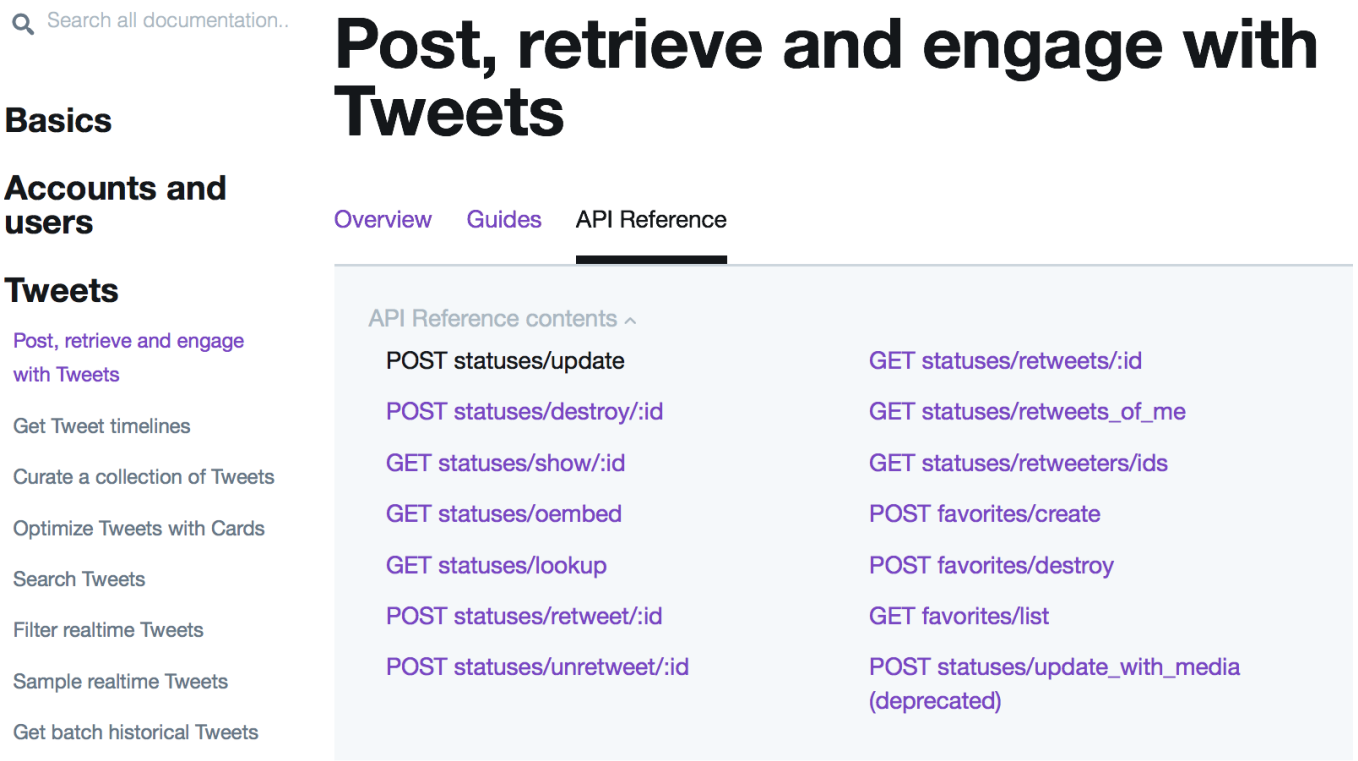
- Increased interactivity with the ‘User’, not with the ‘API’
  - Not drawing the whole page again
- Numerous JavaScript based frameworks (tool)
  - Single Page Applications (SPA)

How to write Web REST API Client

What are the steps involved in writing client code for a RESTful Web API? (e.g., JSON)

First, read and understand the “contract” – API documentation.

e.g., Twitter API Doc.



How to Write Web REST API Client

- The purpose of understanding the contract is for you to understand the following basic tasks that are common in all Web API client
- Common Tasks:
  - Recognising the **Objects** in HTTP responses
  - Constructing **Addresses** (URLs) for interacting with the service
  - Handling **Actions** such as filtering, editing or deleting data

- Take the example from the book:
  - http://rwcbook03.herokuapp.com/task/
  - http://rwcbook03.herokuapp.com/files/json-client.html
  - https://rwcbook.github.io/json-crud-docs/

Use a client-API interaction pattern

Very basic “Single Page Application” structure …

A REST Client App could be designed to act in a simple, repeating loop that roughly goes like this:

1. Execute an HTTP request
2. Store the response (JSON) in memory;
3. Inspect the response for the current context;
4. Walk through the response and render the context-related information on the screen

JSON-client code (from the Amundsen example)

- A processing pattern for a typical listing item handling:
- A processing pattern for adding appropriate “actions” to each list item:

(All done via the basic interaction pattern mentioned above)

Simple programmatic interactions

In Python: What can **Requests** do?

Requests will allow you to send HTTP/1.1 requests using Python. With it, you can add content like headers, form data, multipart files, and parameters via simple Python libraries. It also allows you to access the response data of Python in the same way.

```
@app.route('/')
def hello_world():
    return render_template('showMe.html')


@app.route('/process', methods=['post'])
def processTasks():
    """Get a list of all tasks."""
    url = 'http://rwcbook04.herokuapp.com/task/'
    headers = {'Accept': 'application/json'}
    response = requests.get(url, headers=headers)
    item_dict = response.json()
    return render_template('responseView.html', numTask=len(item_dict['task']))


if __name__ == '__main__':
    app.run()
```

This type interaction covers wider range of application scenarios where the client application does not necessary have human users (i.e., a complete automation)

Response content types of the API may matter (more choice favourable?)

Simple programmatic interactions

In Python: What can **Requests** do?

Requests will allow you to send HTTP/1.1 requests using Python. With it, you can add content like headers, form data, multipart files, and parameters via simple Python libraries. It also allows you to access the response data of Python in the same way.

```
import requests

resp = requests.get('https://todolist.example.com/tasks/')
if resp.status_code != 200:
    # This means something went wrong.
    raise ApiError('GET /tasks/ {}'.format(resp.status_code))
for todo_item in resp.json():
    print('{} {}'.format(todo_item['id'], todo_item['summary']))
```

This type interaction covers wider range of application scenarios where the client application does not necessary have human users (i.e., a complete automation)

Response content types of the API may matter (more choice favourable?)

https://realpython.com/api-integration-in-python/

Simple programmatic interactions

```
task = {"summary": "Take out trash", "description": "" }
resp = requests.post('https://todolist.example.com/tasks/', json=task)
if resp.status_code != 201:
    raise ApiError('POST /tasks/ {}'.format(resp.status_code))
print('Created task. ID: {}'.format(resp.json()["id"]))
```

the json argument to post. If we use that, requests will do the following for us:

- Convert that into a JSON representation string, json.dumps()
- Set the requests' content type to "application/json" (by adding an HTTP header).

https://realpython.com/api-integration-in-python/

Requests (check for status code)

- It is not always the case that things will go smoothly “Status code check”

Example:

```
resp = requests.get('https://api.example.com/stuff/')
if resp.status_code != 200:
    # This means something went wrong.
    raise ApiError('GET /stuff/ {}'.format(resp.status_code))
```

- Pay attention that the status code changes according to operation

```
thing = {"summary": “someThing", "description": ""}
resp = requests.post('https://api.example.com/stuff/', json=thing)
if resp.status_code != 201:
    raise ApiError('POST /stuff/ {}'.format(resp.status_code))
```

Caching API Requests

- To implement caching, we can use a simple package called Requests-cache, which is a “transparent persistent cache for requests”.
- How does it work(Simple Approach):

```
import requests_cache

requests_cache.install_cache(cache_name='mystuff_cache',
backend='sqlite', expire_after=180)
```

- By default the cache is saved in a sqlite database. We could also use a python dict, redis, and mongodb

https://requests-cache.readthedocs.io/en/latest/index.html

Build your own API Library

- If you are developing a sophisticated application you need to move away from simple calls into constructing your own API library.
- This Also applies if you are the owner of the API so having an API library can make your API more usable.

https://requests-cache.readthedocs.io/en/latest/index.html

Build your own API Library (Example)

```
import requests

def get_stuff():
        return requests.get(_url('/stuff/'))

def describe_stuff(stuff_id):
        return requests.get(_url('/stuff/{:d}/'.format(stuff_id)))

def add_stuff(summary, description="")
        return requests.post(_url('/stuff/'), json={
                'summary': summary,
                'description': description,
                })
```

Coping with changes …

Software systems evolve over time … The APIs you rely on will too.

There is a principle that has been used by HTTP itself to evolve its versions without causing failure …

Must Ignore

HTTP directive “MUST IGNORE” = any element of a response that the receiver do not understand must be ignored without halting the further processing of the response.

HTML evolved over time using the similar approach … (“forgiving” browsers)

Take V4. of the Amundsen’s API.

MUST IGNORE …

V1.

{

“links”: […],

“data”: […].

“actions”: […]

}

V2.

{

“links”: […],

“data”: […].

“actions”: […],

“extensions”: […]

}

WITH response Do

PROCESS response.links

PROCESS response.data

PROCESS response.actions

END

(proceeds without breaking/haltering)

If Client wants to take advantage of the new feature, the client code will be updated, of course.

Useful Resources

- https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/
- https://realpython.com/api-integration-in-python/
- Book: RESTful Web Clients: Enabling Reuse Through Hypermedia, By Mike Amundsen

Q&A