

COMP9321

Data Services Engineering

Term 1, 2019

Week 4 Lecture 2

Designing RESTful API and Web/RESTful Client

COMP9321 2019T1

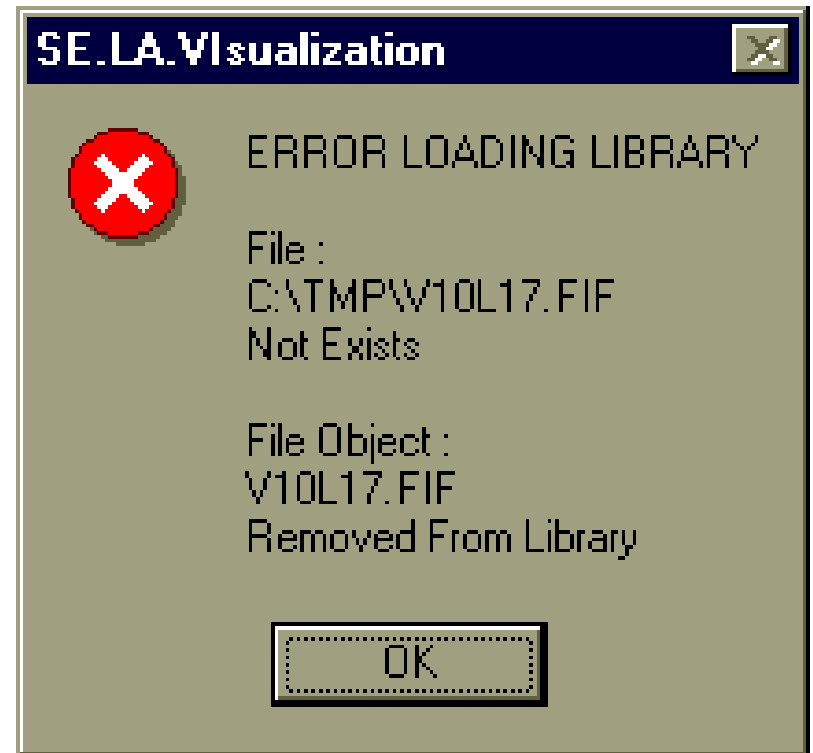
(Bad) Examples of User Interfaces



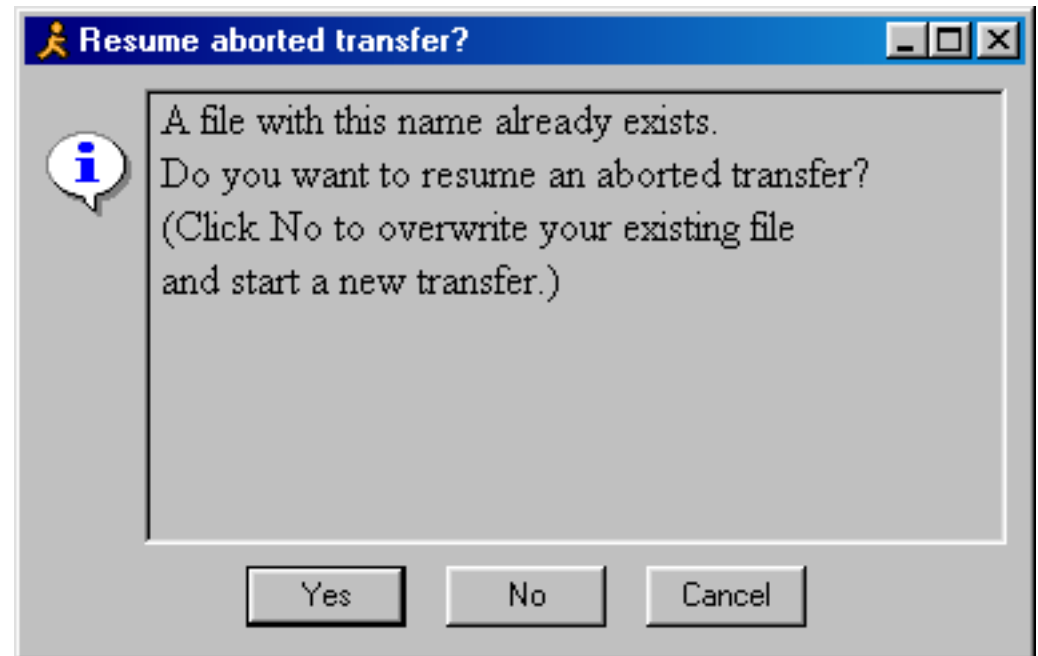
(Bad) Examples of User Interfaces



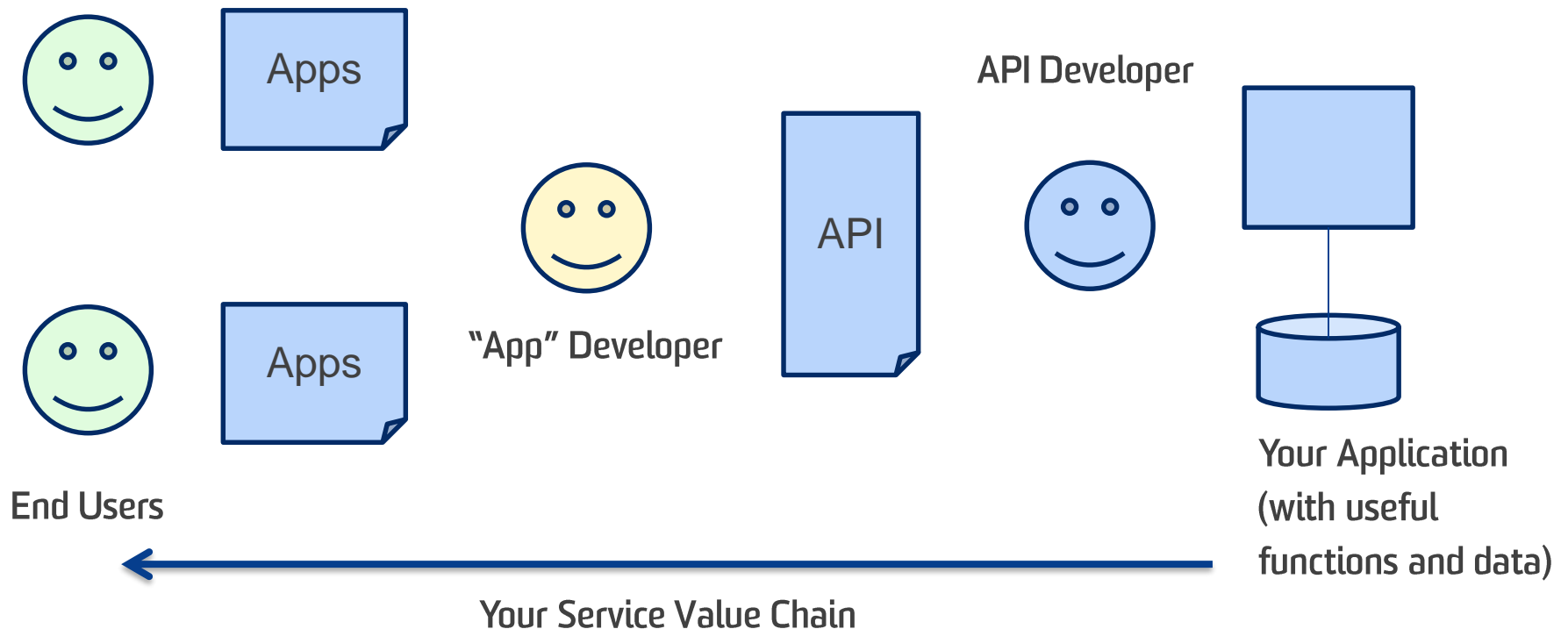
(Bad) Examples of User Interfaces



(Bad) Examples of User Interfaces



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 things) 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:

`/coffeeOrders`

```
{
  "resultSize": 25,
  "results": [ {
    "id": 100,
    "type": "latte",
  },
  { "id": "101",
    "type": "cap",
  },
  { ... },
]
}
```

GET `/coffeeOrders/100`

```
{
  "Id": "100",
  "type": "latte",
  "extra shot": "no",
  "payment": {
    "date": "2015-04-15",
    "credit card": "22223"
  },
  "served_by": "sally"
}
```

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": "100",
    "type": "latte",
    "links": [ {
      "href": "/coffeeOrders/100",
      "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.
- A version number in the URI, where the clients could indicate the version of a resource they need directly in the URL.

<http://service/v1/part/123>

<http://service/v2/part/123>

<http://service/part/123?version=v3>

API Versioning

- Some apps prefer using Accept and Content-Type with version identifiers;
- Content Type header is used to define a request and response body form (from both clients and servers)
- Accept header is used to define supported media type by clients.

API Versioning: Accept Header

Request

```
GET http://service/parts/123
Accept: application/json; version=1
```

Response

```
HTTP/1.1 200 OK
Content-Type:
application/json; version=1
{"partId":"123","name":"Engine"}
```

Request

```
GET http://service/parts/123
Accept: application/json; version=2
```

Response

```
HTTP/1.1 200 OK
Content-Type:
application/json; version=2
{"partId":"123",
 "name":"Engine","type":"Diesel"}
```

Request

```
GET http://service/parts/123
Accept: application/json; version=1,
application/xml; version=1
```

Response

```
HTTP/1.1 200 OK
Content-Type:
application/xml; version=1
<part>
  <partId>123</partId>
  <name> Engine </name>
</part>
```

Caching

Caching

- HTTP provides built-in caching framework. No need to change code while adding caching layer
- Caching can be established on the client/server/proxy

Header	Parameter Meaning
Last Modified	This parameter gives the Date and Time when the server last updated the representation.
Cache-Control	This is used for HTTP 1.1 header to control caching.
Date	Date and time when this representation was initially generated.
Expires	Date and time when representation will expire. (HTTP 1.0 clients)
Age	Total time in seconds since the representation was retrieved from the server.

Caching

- Setting expiration caching headers for responses of GET, and HEAD requests for all successful response codes.
- POST is considered to be non-cacheable
- Adding caching headers to the response codes starting with 3 and 4, to reduce the amount of error-triggering traffic from clients.

How to Handle Unsynchronized Tasks

How to Handle Unsynchronized Tasks

- HTTP is synchronous and stateless protocol.
 - The client and server get to know each other during the current request
 - Forget about the request after that.

An example for managing asynchronous tasks

1. Place a GET/POST request which takes too long to finish
2. Create a new task and return status code 202 with a representation of the new resource so the client can track the status of the asynchronous task
3. On completion of the request, return response code 303 and a location header containing a URI of resource that displayed the result set
4. On request failure, return response code 200 (OK) with a representation of the task resource informing that the process has failed. Clients will look at the body to find the reason for the failure.

How to Handle Unsynchronized Tasks

Multiple file uploading

Request

POST /files/ HTTP/1.1

Host: www.service.com

Response

HTTP/1.1 202 Accepted

Content-Type:

application/xml;charset=UTF-8

Content-Location:

http://www.example.org/files/1

<status>

<state>pending</state>

<message xml:lang="en">

File Upload process is started
and to get status refresh page
after sometime.

</message>

</status>

Request

GET /file/1 HTTP/1.1

Host: www.service.com

Response

HTTP/1.1 303

Location:

www.service.com/file/1

content-Location:

www.service.com/file/ process/1

<status

<state>completed</state>

<message> File Upload is
completed</message>

</status>

REST API Security...Does it matter?

OWASP REST API Security Cheat Sheet

- It matter enough that **Open Web Application Security Project (OWASP)** included many instances in their web security Top ten related to APIs and they have the REST Security cheat sheet.
- REST relies on the elements of the Web for security too (Check OWASP top 10)
- Things to remember (Input Validation, Methods restriction, logging)

REST APIs and Security

HTTPS (SSL)

- “Strong” server authentication, confidentiality and integrity protection
The only feasible way to secure against man-in-the-middle attacks
- Any security sensitive information in REST API should use SSL

See the OWASP Transport Layer Protection Cheat Sheet

REST APIs and Security (Cont'd)

API developers at least must deal with authentication and authorisation:

Authentication (401 Unauthorized) vs. Authorisation (403 Forbidden):

Common API authentication options:

- HTTP Basic (and Digest) Authentication: IETF RFC 2617
- Token-based Authentication
- API Key [+ Signature]
- OAuth (Open Authorisation) Protocol - strictly uses HTTP protocol elements only

Authentication

The basic idea revolves around: "login credentials" (for app, not human)

The questions: (i) what would the credentials look like and how would you pass them around "safely"? (ii) how to ensure stateless API interactions? (no 'session')

HTTP Basic Authentication Protocol (HTTP Specification)

Initial HTTP request to protected resource

```
GET /secret.html HTTP/1.1  
Host: example.org
```

Server responds with

```
HTTP/1.1 401 Unauthorized  
WWW-Authenticate: Basic realm="ProtectedArea"
```

Client resubmits request

```
GET /secret.html HTTP/1.1 Host: example.org  
Authorization: Basic Qm9iCnBhc3N3b3JkCg==
```

Further requests with same or deeper path can include the additional Authorization header pre-emptively

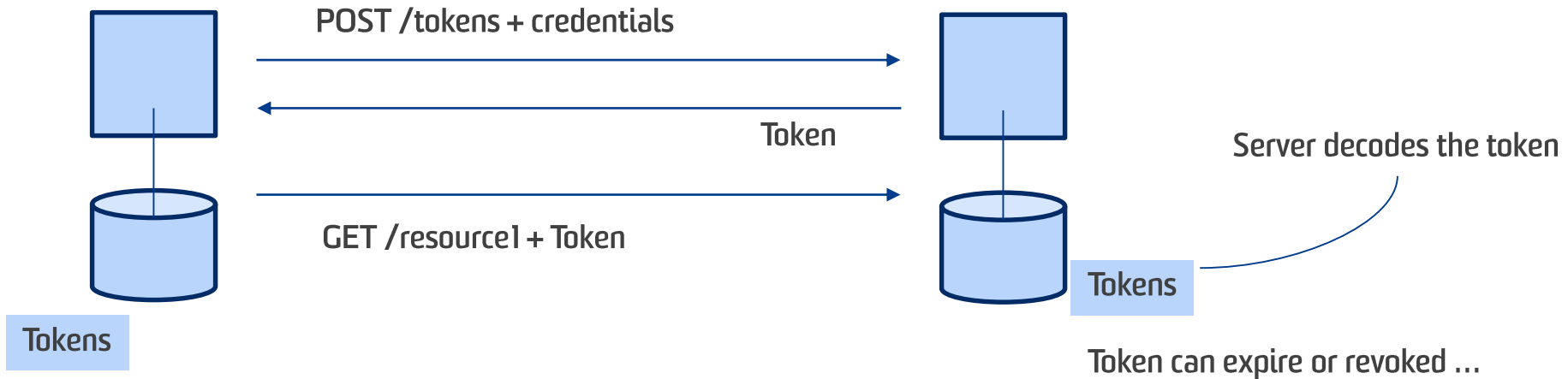
HTTP Basic Auth

Issues with HTTP Basic Auth as an API authentication scheme

- The password is sent over the network in base64 encoding - which can be converted back to plain text
- The password is sent repeatedly, for each request - larger attack window
- HTTP Basic Auth combined with SSL could work for some simple situations ... But normally this scheme is not recommended and considered not secure “enough”
- You may choose HTTPS protocol, which encrypts the HTTP pipe carrying the passwords

Token-based method

- User enters their login credentials. Server verifies the credentials are correct and returns a token
- This token is stored client-side (local storage). Subsequent requests to the server include this token
- The password is not sent around ...



JWT (JSON Web Tokens) – industry standard now (RFC 7519)
e.g., Facebook, Twitter, LinkedIn ...

Token-based method

- Question: How could you manage this scheme?
 - Maybe generate a random token/string and store it against each user - ?? User state creeping into the server ... Not good (e.g., think about load balancing)

JWT (JSON Web Tokens) – (almost) industry standard now (RFC 7519)

e.g., Facebook, Twitter, LinkedIn ...

- The message content consists of three parts JSON data ... (encoded and signed)
- A key idea is that the token is self contained. You can store the identity in the JSON, sign it and send the token to the client. The client will use the token in all subsequent requests to authenticate itself.
- Since it's "signed", the server can verify and validate the token, without having to do a database look up, session management, etc.
- That is, (i) a token can be effectively used to authenticate requests in a stateless fashion, (ii) login password of the client is not revealed

The signatures (cryptography technique)

Keyed-Hash Message Authentication Code (HMAC) is an algorithm that combines a certain payload with a secret using a cryptographic hash function.

The result is a code that can be used to verify a message only if both the generating and verifying parties know the secret. In other words, HMACs allow messages to be verified through shared secrets.

Provides authentication as well as integrity properties in security

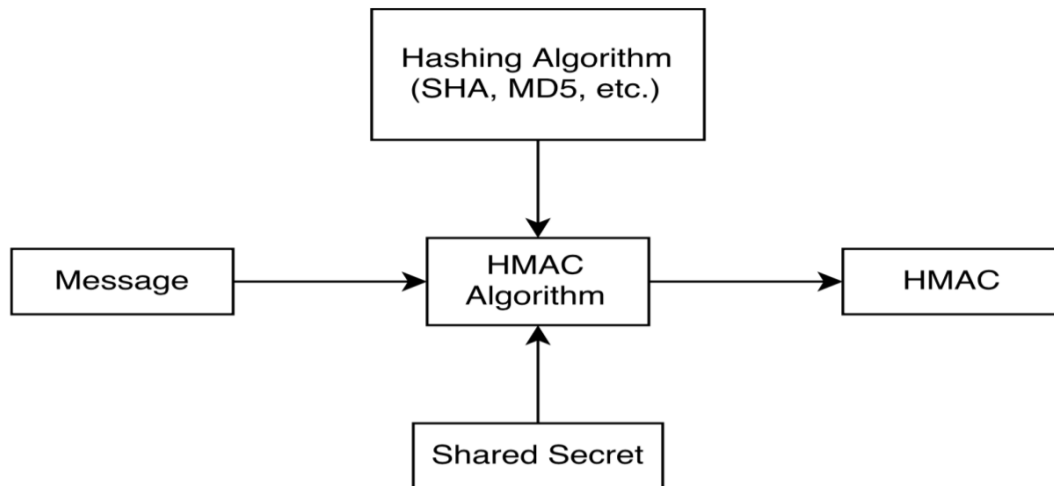
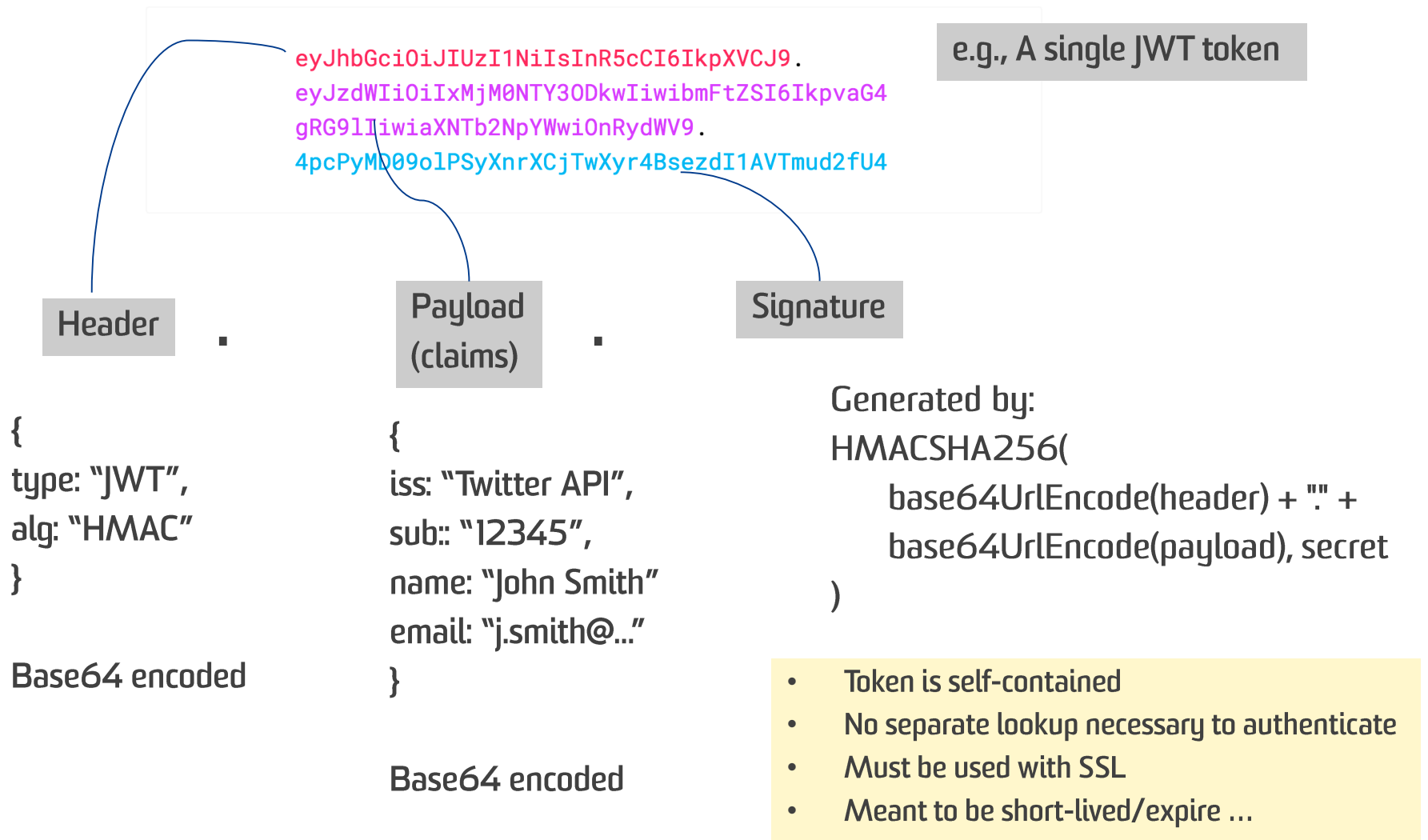


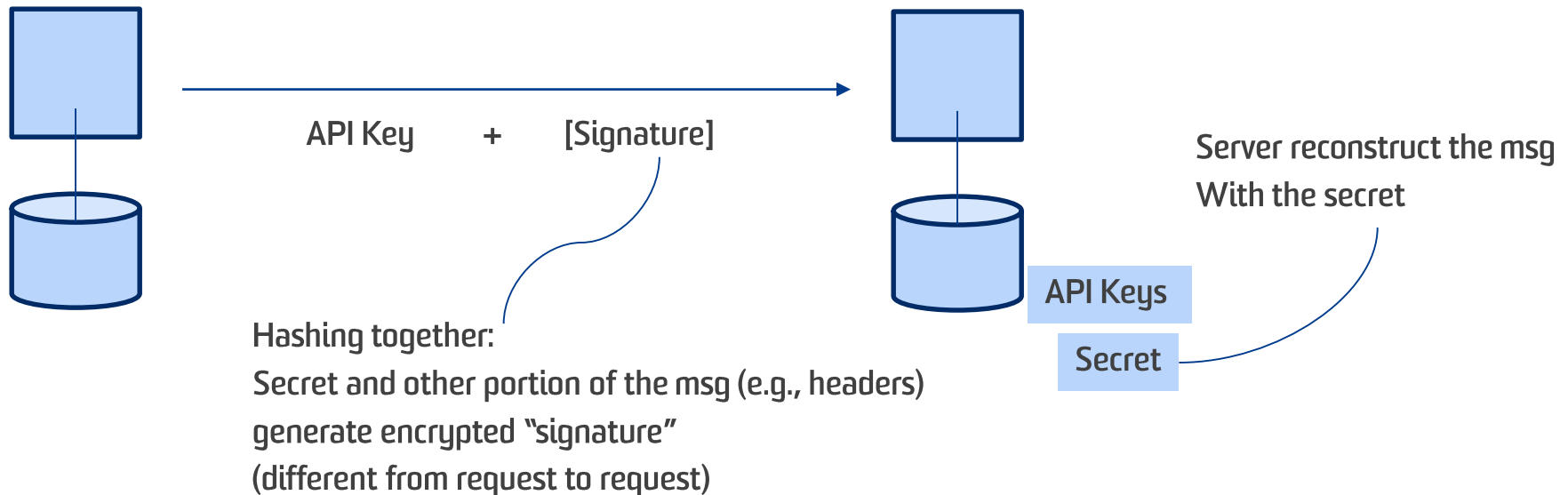
Figure 7.3: HMAC

Token-based method: what is in a token?



API Key method

- From User (API consumer) point of view:
 - Sign up for the service, API key for the user is issued by the server
 - Copy the issued API key [and secret] in all requests
 - \approx user id and password, except it is meant to be authenticate the 'client application'



API Key Method


- API key (in combination with Secret) -> an authentication scheme
- Usually, an API key gives you access to a wide range of services from the same provider
- API provider also use it
 - to get usage analytics
 - to limit the usage rate (e.g., 5 calls per second | 429 “Too Many Requests”)
 - to issue further access tokens
- Most API providers would use API management platform (Apigee, Mulesoft, etc.) to handle questions like:
 - Where should the client send the API key and secret?
 - HTTP Header | Query parameters | Request body (custom section)
 - API key & secret management
 - Running an appropriate security scheme
 - Rate limiting and usage analytics

OAuth

- OAuth tries to solve this problem ...

Find people you know on Facebook

Your friends on Facebook are the same friends, acquaintances and family members that you communicate with in the real world. You can use any of the tools on this page to find more friends.

 **Find People You Email**

[Upload Contact File](#)

Searching your email address book is the fastest and most effective way to find your friends on Facebook.

Your Email:

Password:

Find Friends

We won't store your password or contact anyone without your permission.

- Essentially, an authorisation scheme for data access ...

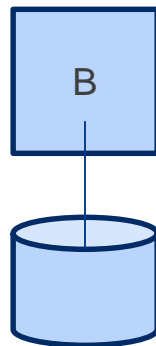
OAuth ([RFC-6749](#))

How does the user allow Company B to access the data in Company A without revealing the login credential for Company A to Company B?

Your valuable customer
(human, here ...)



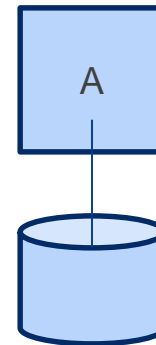
Some other service
Wanting your
customer's data
(e.g., email, address)



API requests

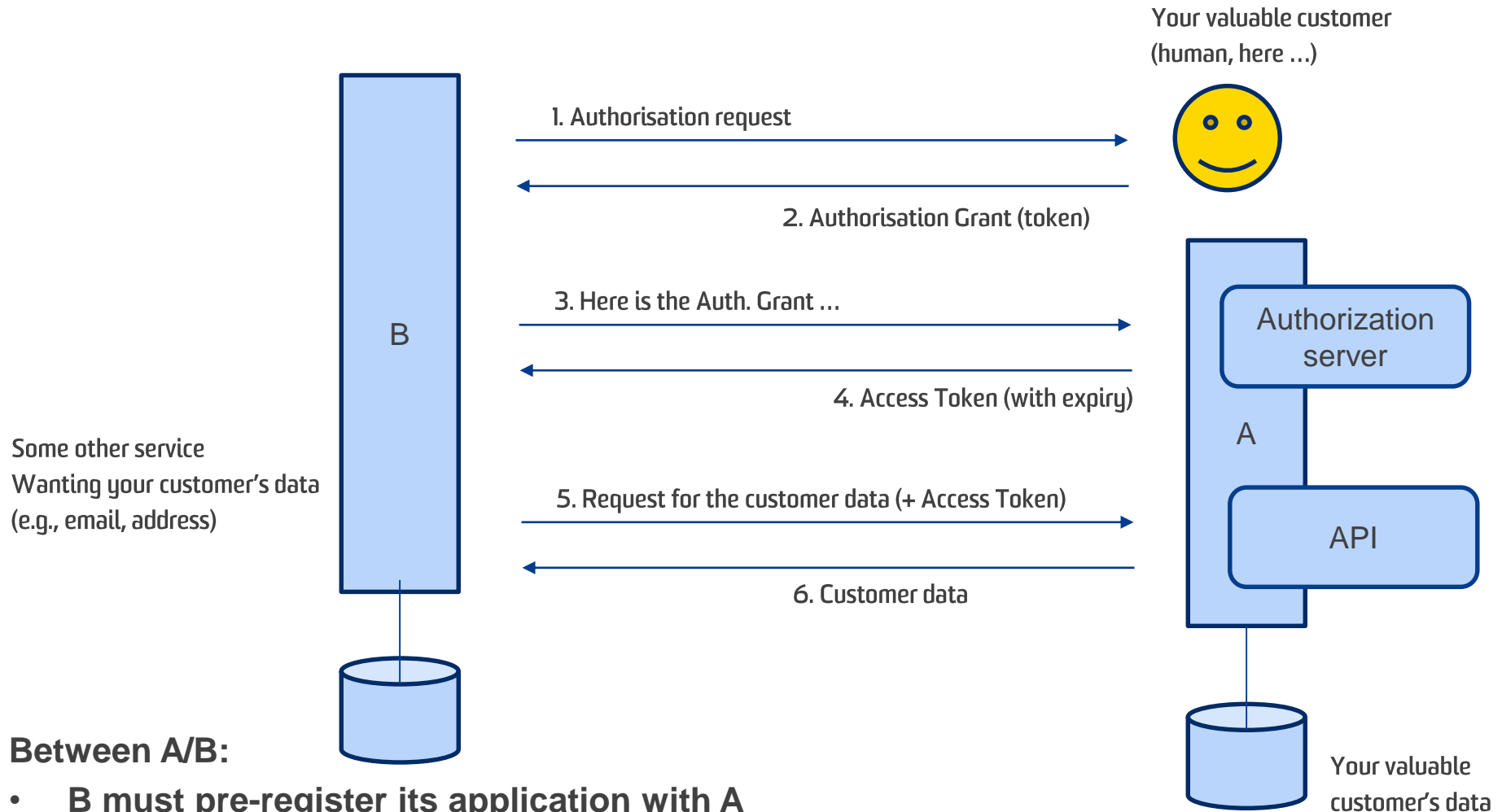


Customer data



Your valuable
customer's data

OAuth workflow (e.g., social login scheme)



OAuth has different scope

Facebook:

<https://developers.facebook.com/docs/facebook-login/access-tokens>

Spotify:

<https://beta.developer.spotify.com/documentation/general/guides/scopes/>

Should consider the scope during the design of OAuth scheme for your API.

Using the API

Standardised 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>)

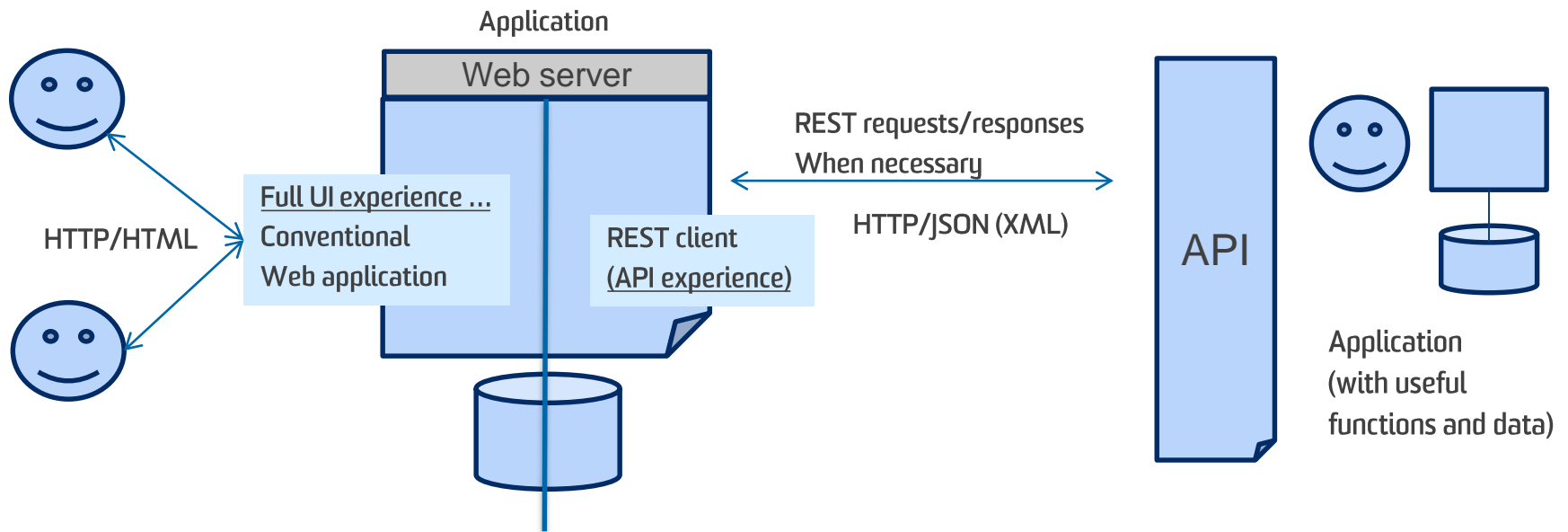
Standardised API specification

Basic structure (what is in it ...)

- A quick run through: <https://swagger.io/docs/specification/basic-structure/>
- Petstore.yaml

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.

🔍 Search all documentation..

Basics

Accounts and users

Tweets

[Post, retrieve and engage with Tweets](#)

[Get Tweet timelines](#)

[Curate a collection of Tweets](#)

[Optimize Tweets with Cards](#)

[Search Tweets](#)

[Filter realtime Tweets](#)

[Sample realtime Tweets](#)

[Get batch historical Tweets](#)

Post, retrieve and engage with Tweets

[Overview](#) [Guides](#) [API Reference](#)

API Reference contents ^

[POST statuses/update](#)

[POST statuses/destroy/:id](#)

[GET statuses/show/:id](#)

[GET statuses/oembed](#)

[GET statuses/lookup](#)

[POST statuses/retweet/:id](#)

[POST statuses/unretweet/:id](#)

[GET statuses/retweets/:id](#)

[GET statuses/retweets_of_me](#)

[GET statuses/retweeters/ids](#)

[POST favorites/create](#)

[POST favorites/destroy](#)

[GET favorites/list](#)

[POST statuses/update_with_media \(deprecated\)](#)

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
 - OAA challenges – named by Mike Amundsen
- 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?)

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

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.

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": [...]  
}
```

WITH response Do
PROCESS response.links
PROCESS response.data
PROCESS response.actions
END
(proceeds without breaking/haltering)

V2.

```
{  
  "links": [ ...],  
  "data": [ ...].  
  "actions": [...],  
  "extensions": [...]  
}
```

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