REST

About: me

- Marco Funaro software engineer @ GoCloud s.r.l.
- Twitter: @marcofunaro
- Linkedin: https://it.linkedin.com/in/marcofunaro
- email: marco.funaro@gocloud.it

“ The Representational State Transfer (REST) style is an abstraction of the architectural elements within a distributed hypermedia system.”

–Roy Fielding

> “ The REpresentational State Transfer (REST) style is an abstraction of the architectural elements within a distributed hypermedia system.”

–Roy Fielding

Where did it come from?

- It was not invented, it was deducted in 2000 by Roy Fielding in his PhD thesis

The original origin

- Tim Berners-Lee wants data sharing in research
- 1989: the first proposal of a solution
- 6 August 1991: the first web site is online (in France)
- WWW is born

Requirements

- Low-entry barriers:
  - Easy language, authoring always possible
- Extensible
- Distributed Hypermedia
- Internet scale:
  - Anarchic Scalability
  - Independent deployment

Bottom line

The system should be stupid

Bottom line

The system should be as stupid as possible

BUILDING BLOCKS

Resource

- Anything worth to be part of our model
- The nouns in the domain
- Close to the concept of object (not class) in OO

URIs

- Unique identifier for a resource
- It lasts forever
- Different URIs can denote the same resource
- No two different resources can have the same URI

Representations

- Resource are abstract concepts
- Some can be sent over the internet
- Some cannot (vending machines)

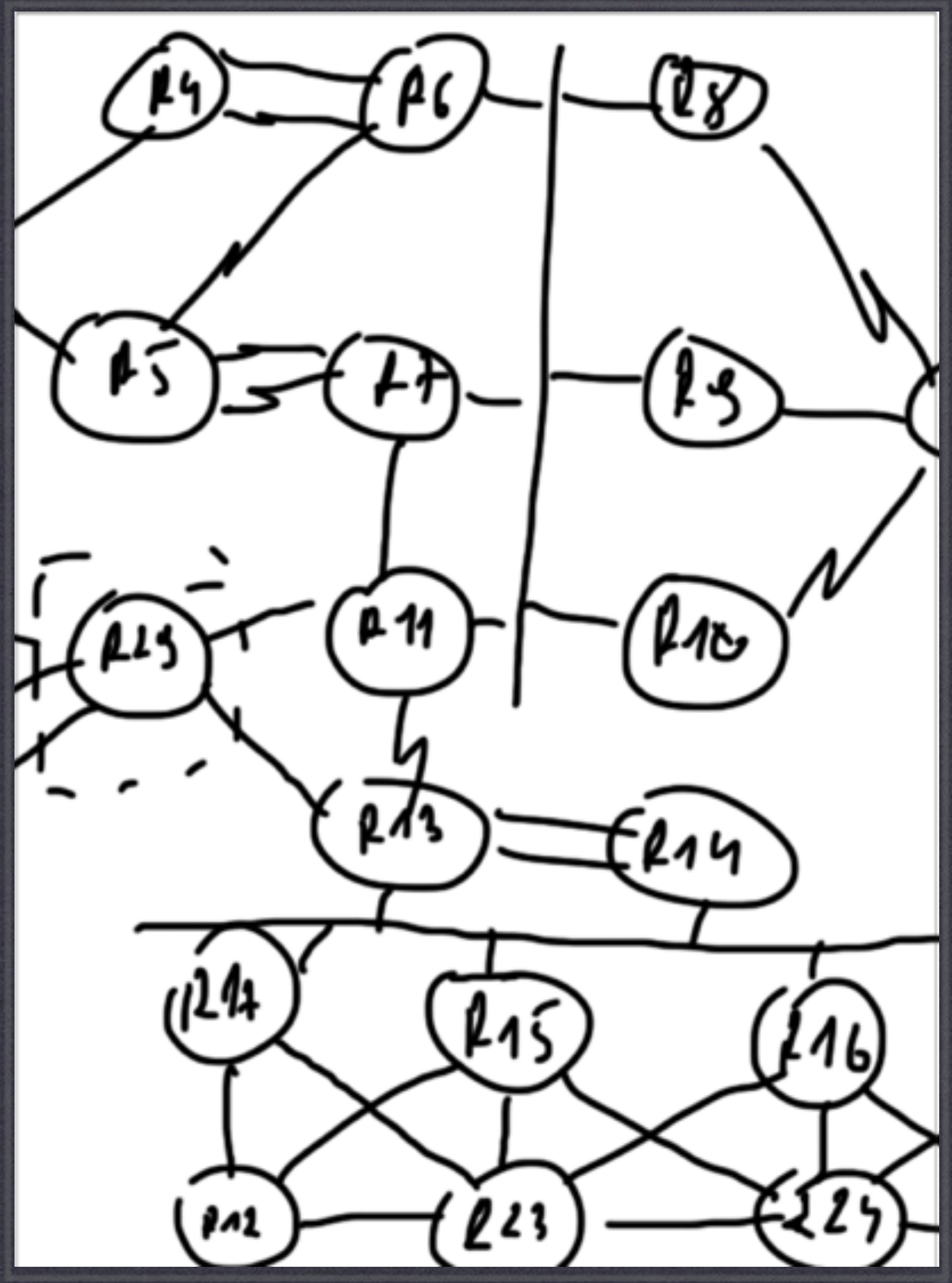
Representations

- Different users can see different representations
- Different representations can be explicitly asked
- Can be negotiated automatically

Links

- They actually build the web
- Make the anarchic system cohesive

PROPERTIES



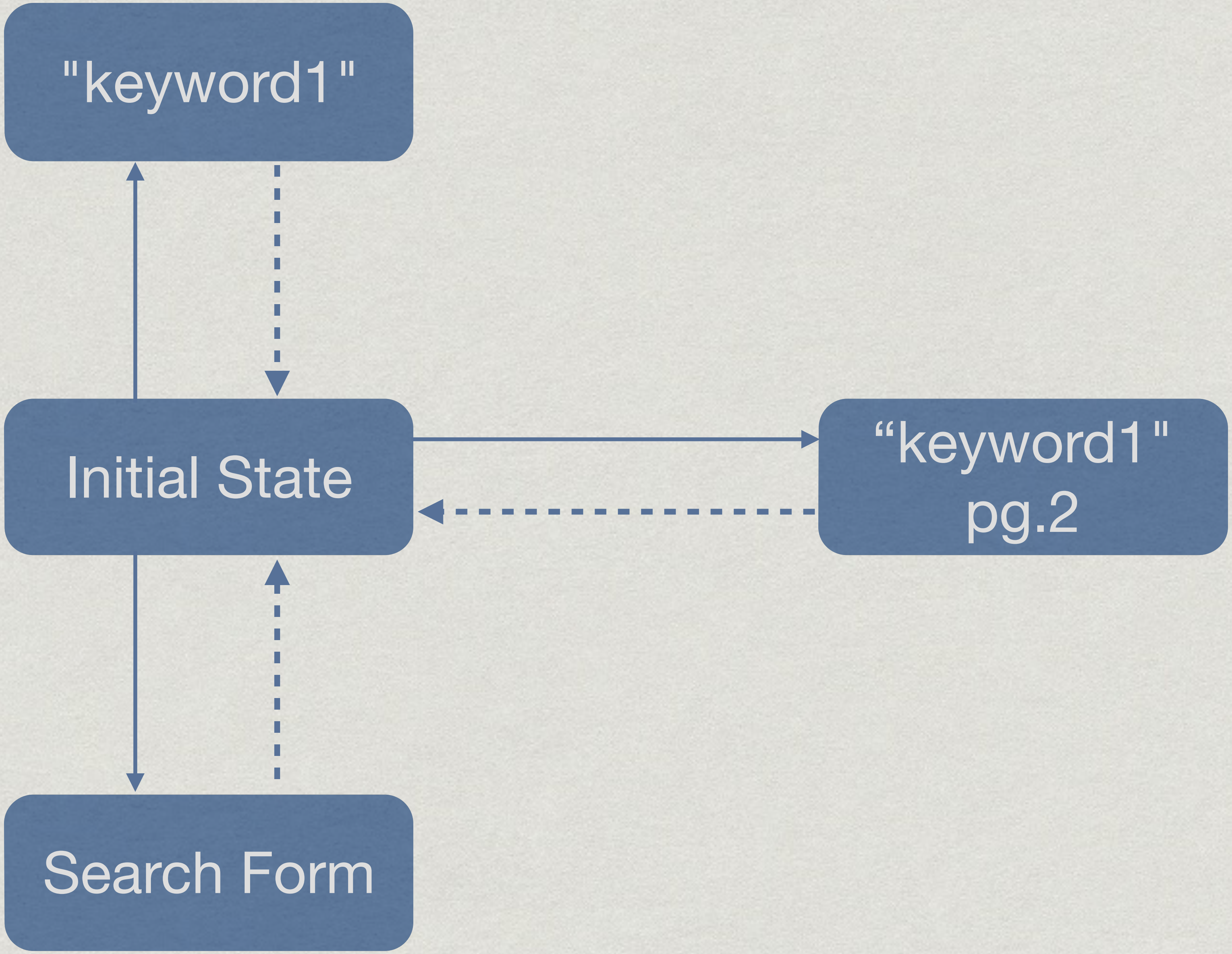
Addressability

- We have URIs and we are not afraid to use it:
  - For unforeseen usage
  - No single entry point, everyone can enter everywhere in the flow

Statelessness

- All the interactions are stateless
- Every request happens in isolation
- It’s self-contained
- No sessions

Stateless search engine



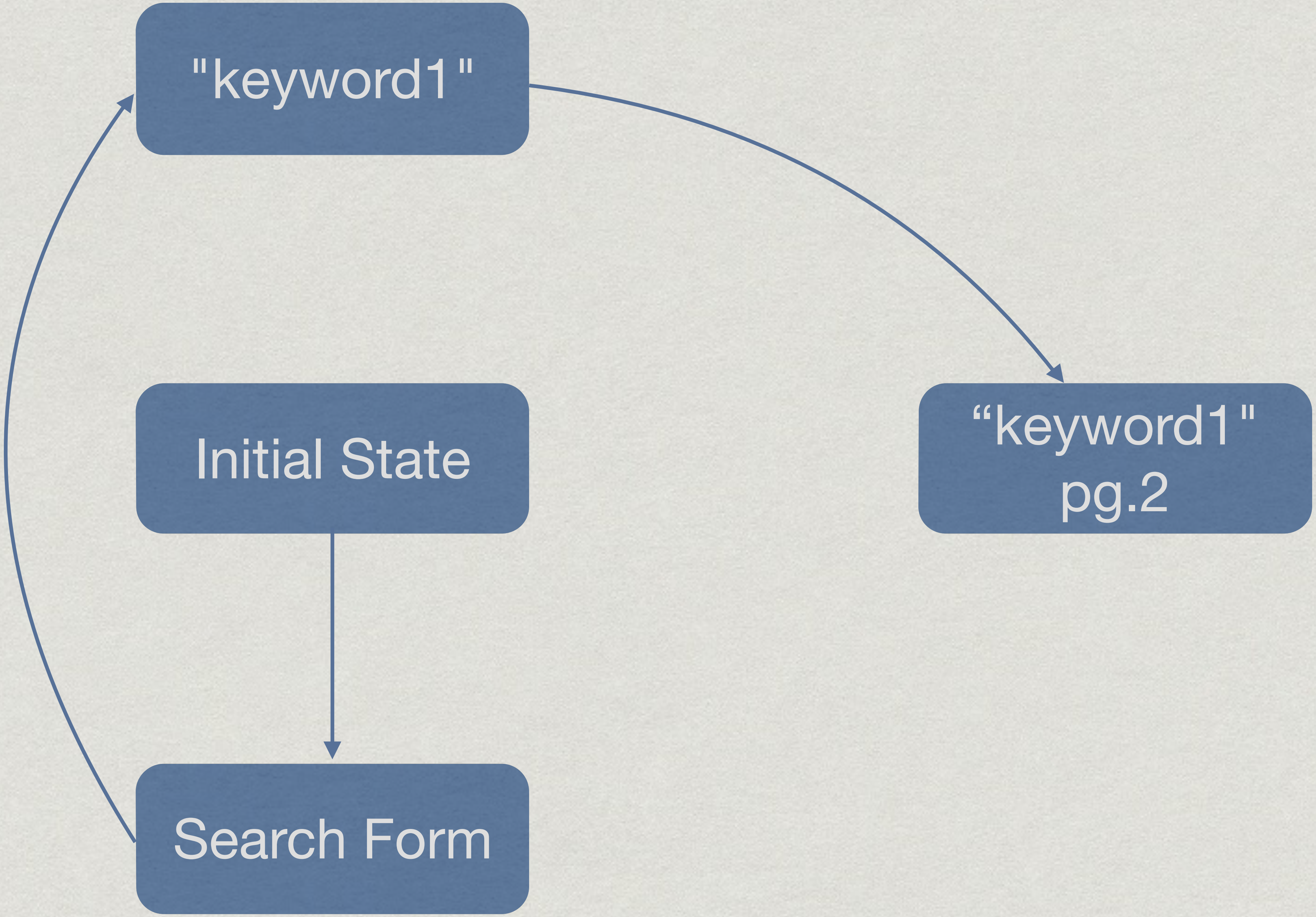
"keyword1"

Initial State

“keyword1" pg.2

Search Form

Stateful search engine



Statelessness — Effects

- With addressability, every data needed for the server can be referenced in the request
- It allows back and forward (beware of POSTs)
- It enables bookmarking
- It allows to scale (load balancer, caching)
- It might conflict with cookies and API keys

A Clarification

- Statelessness principle does not advocate for stateless applications
- Interactions state Vs Resource state
- The Flickr example (Pics always reside on the server)

Connectedness

- Aka HATEOAS (Hypermedia As The Engine Of Application State)
- Representations should carry links
- Such links are a loose guide for the users

Connectedness — Effects

- Improves on addressability: not only a broader interface but also unknown and unforeseen interactions
- Think of google search, no URL is typed in

Uniform Interface

- All resources have the interface
- REST does not mandate it
- HTTP does!

Uniform Interface

- GET: retrieves representation
- PUT: update the whole resource
- DELETE: delete resource
- POST: create new resource
- PATCH: partially update a resource

Uniform Interface

	Safe	Idempotent
GET	✓	✓
PUT		✓
DELETE		✓
PATCH		
POST		

Uniform Interface

- Safety and Idempotency are properties seen by the client
- The server can have side effects
- They should not influence the client
- E.g., hit count
- Unsafe operations translate interaction state in application state

Uniform Interface — POST

- POST can be overloaded
- The server can take different actions depending on the payload
- Uniformity broken
- The verb loses meaning

Uniform Interface — Effect

- Any client can work with any server
- They should not be aware of each other in advance
- They just need to be able to understand the interface (POST overloading!!)

Summary

- It’s just four concepts:
  - Resources
  - Their names (URIs)
  - Their representations
  - The links between them

- And four properties:
  - Addressability
  - Statelessness
  - Connectedness
  - Uniform Interface

TIPS & TRICKS

URI Design

- Not needed for pure REST
- Use / for hierarchies `/customers/details/first`
- Use comma for ordered sets: `/Earth/45.506544,9.228081`
- Use semicolons for unordered sets: `color-blends/red;blue`
- Use query variables for algorithm inputs
- Do not use verbs (controversial)

Asynchronous requests

- Long running computations might be served asynchronously
- The request immediately returns with `202 Accepted`
- and a uri the client can poll for the answer:

```
{
    “progress" : "20%",
    “response" : —
}
```

Ranges/pagination

- Some resource might have too many sub-resources
- Plain old pagination can be used

https://www.google.it/search?q=test&start=10

Notifications

- They clearly break REST
- Websocket has been proposed (and implemented)
- Still there is a workaround with long polling:
    - The client issues a request
    - The server will not generate a response
    - The client will wait for a timeout and then reconnect
    - Until the notification arrives

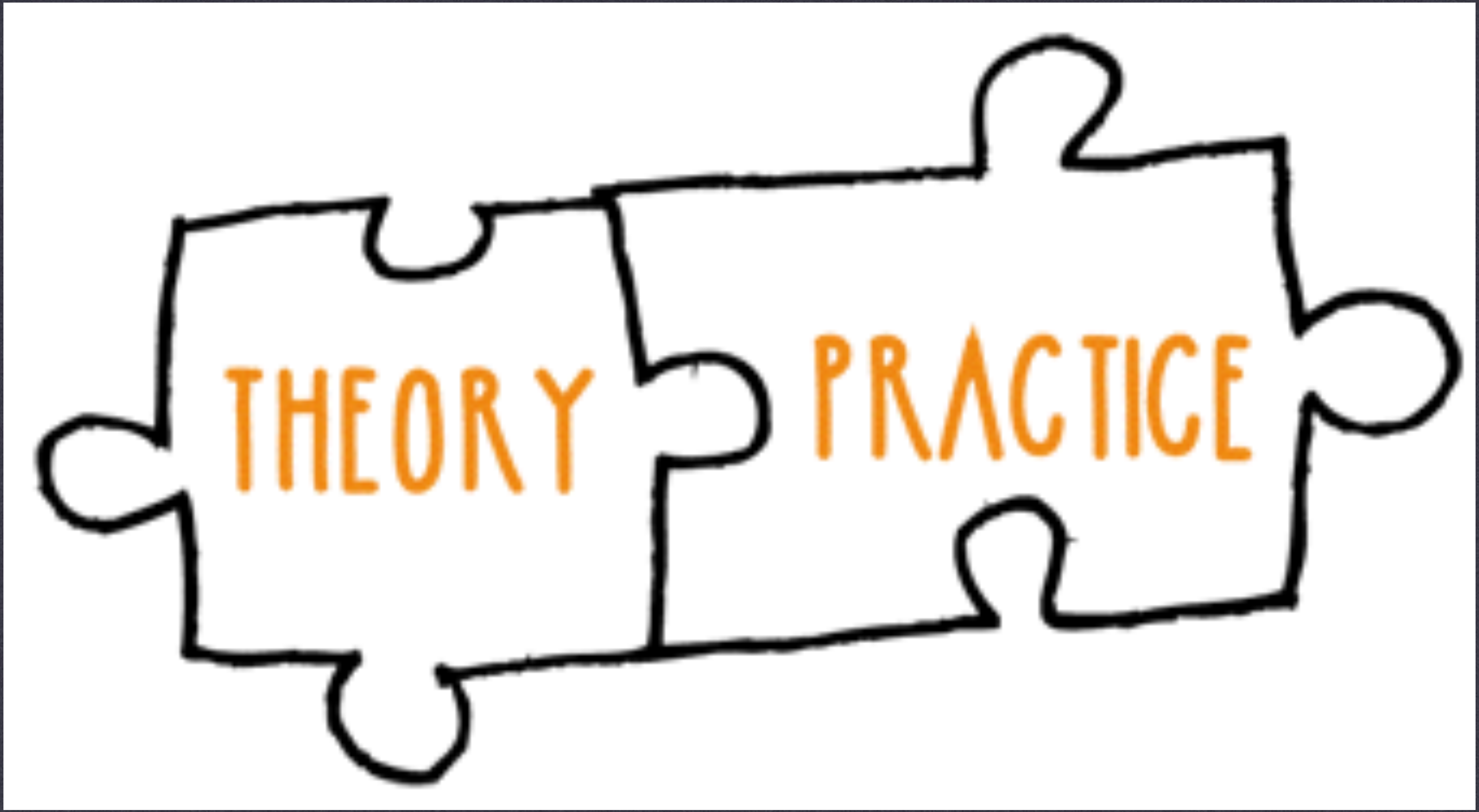
Be nice with each other

Client

- Don’t depend on URI structure
- Support unknown links
- Ignore unknown content

Server

- Don’t break URI structure unnecessarily
- Evolve via additional resources
- Support older formats



PROJECT

EXAMPLES

DATE **DATE**

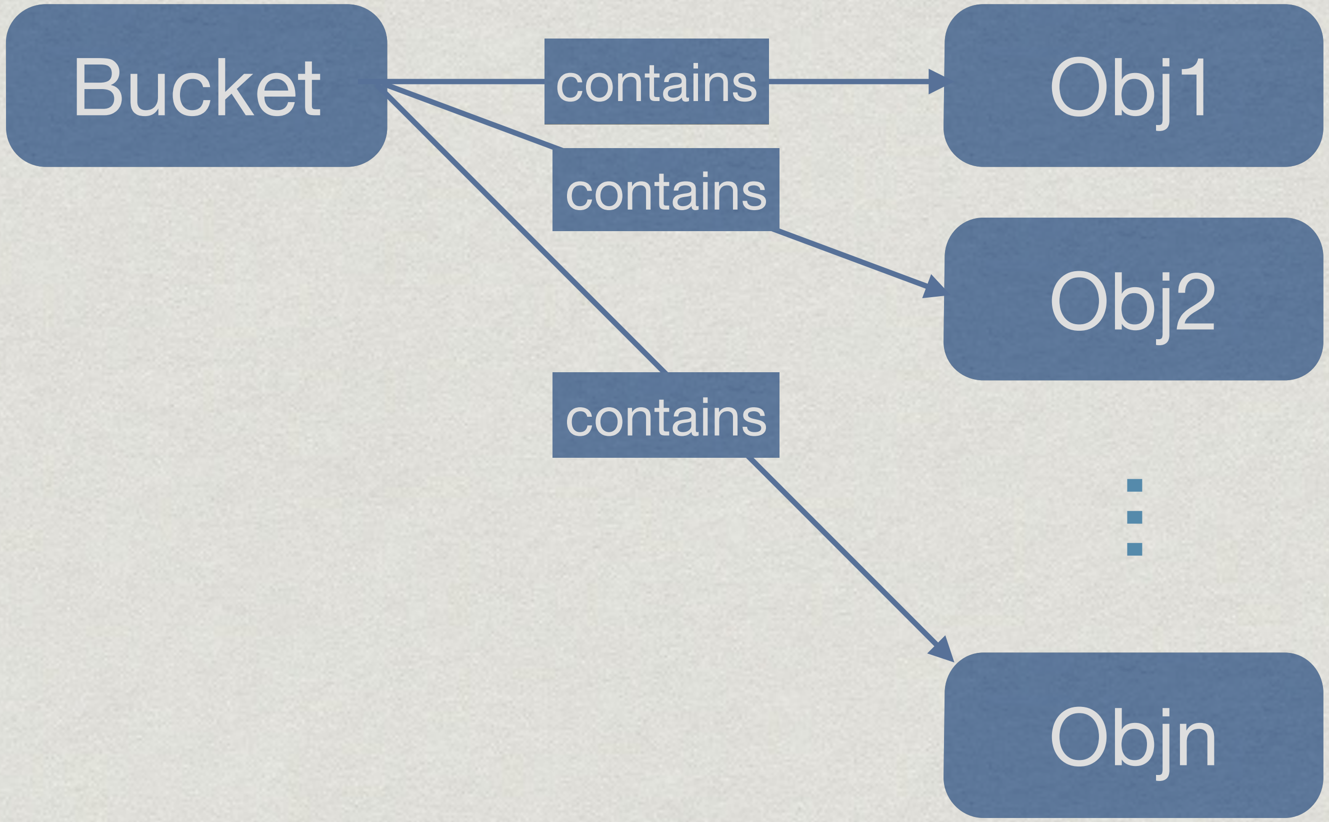
CLIENT **NAME**

BUCKET

Bucket

- A bucket can contains any uniquely identified object
- It’s a sort of key-value store

Resources



URIs

- /bucket
- /bucket/{id}

Representations

- Bucket:

```
{
 “items" : [
    “id1" : $value,
    “id2" : $value,
    “id3" : $value
  ]
}
```

Representations

- Bucket:

```
{
 “items" : [
    “id1" : $value,
    “id2" : $value,
    “id3" : $value
   ]
}
```

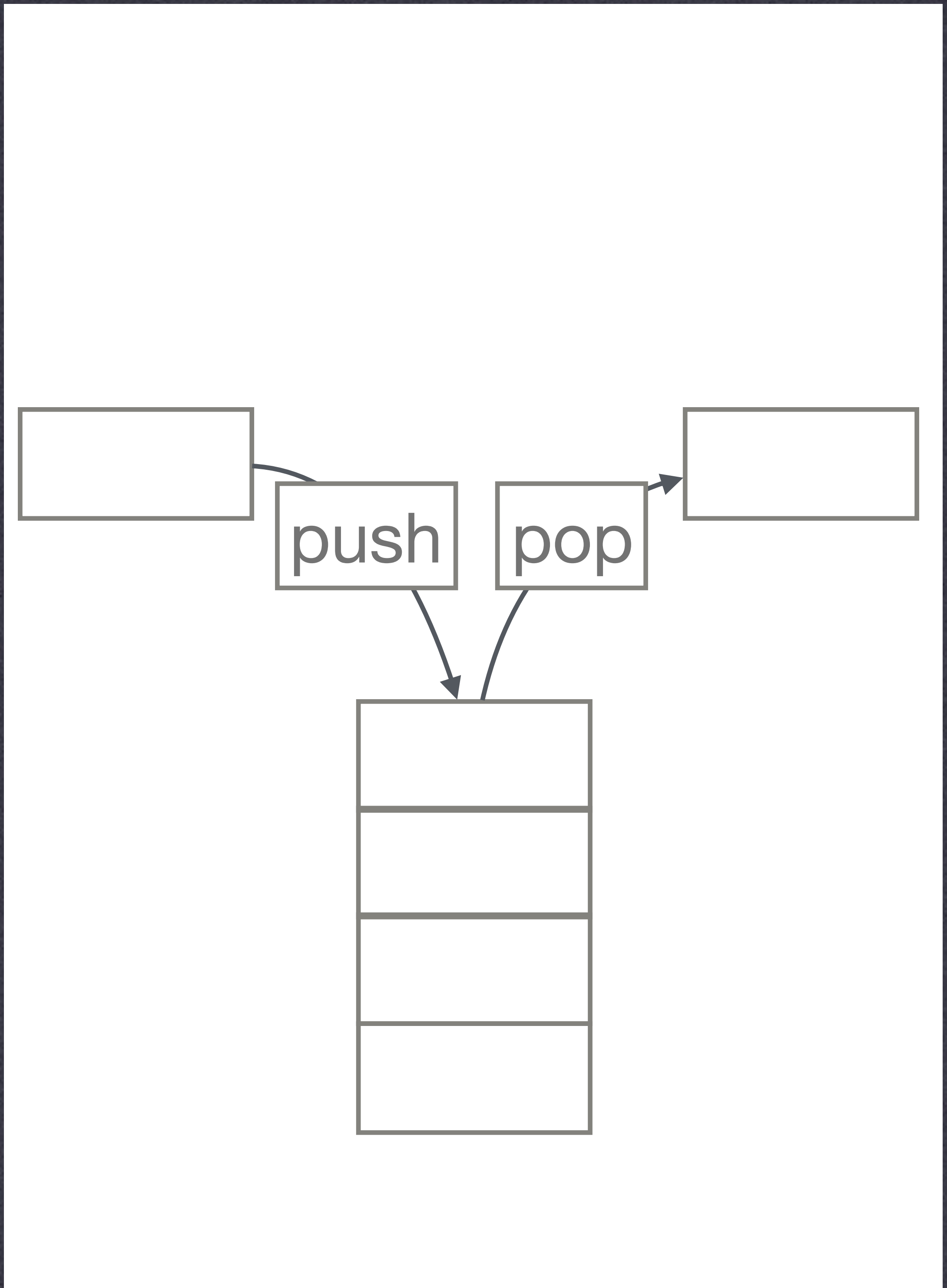
- Element:

```
{
 “id” : $id,
 “value”: $value
}
```

Operations — Bucket

- GET /bucket —> returns its representation
- DELETE /bucket —> deletes everything (maybe disabled)
- PUT /bucket (bucketRep) —> complete override
- PATCH /bucket(bucketRep) —> overrides some elements
- POST /bucket(elementRep) —> creates new element

STACK



Stack

- A stack is a data structure with LIFO policy and three operations:
  - push: add an element on top of the stack
  - pop: removes and returns the element on top of the stack
  - peek: returns the element on top of the stack (without removing it)

Stack

- The only RESTful way to model a stack is like a bucket
- We would like to hide the internals and serve only the TOS
- But POP cannot be implemented without breaking REST
- We have to give up the control of LIFO semantics to the user

FACEBOOK

Facebook simplified

- We want to model the fb mechanisms for:
  - subscription
  - friends browsing
  - friends requests

Resources (and their URIs)

- fb/users
- fb/users/{id}
- fb/users/{id}/requests
- fb/users/{id}/requests/{reqId}
- fb/users/friendships/
- fb/users/friendships/{id}
- fb/users/friendships/{id};{id}

Operations — fb/users

	Semantics
GET	✗
PUT	✗
DELETE	✗
PATCH	✗
POST	Subscription

Operations — fb/users/{id}

	Semantics
GET	User rep
PUT*	Update profile
DELETE*	Exit to real life
PATCH*	Update profile
POST	✗

*** only for the owner**

Operations — fb/users/{id}/requests

	Semantics
GET*	Request list
PUT	✗
DELETE	✗
PATCH	✗
POST	Add new request

*** only for the owner**

Operations — fb/users/{id}/requests/{reqId}

	Semantics
GET*	Request rep
PUT	✗
DELETE*	Delete request
PATCH	✗
POST	✗

*** only for the owner**

Operations — fb/users/friendships/

	Semantics
GET*	All the friendships
PUT	✗
DELETE	✗
PATCH	✗
POST	New friendship

*** only for the owner**

Operations — fb/users/friendships/{id}

	Semantics
GET	id’s friendships
PUT	✗
DELETE	✗
PATCH	✗
POST	✗

*** only for the owner**

Operations — fb/users/friendships/{id};{id}

	Semantics
GET	1 friendship
PUT	✗
DELETE*	not friends anymore
PATCH	✗
POST	✗

*** only for the owner**

Representation — /users/{id}

```
{
 “id” : $id,
 “name”: “Marco”,
 “lastName” : “Funaro”,
 “birthday" : “22/12”,
 “friends" : /users/id/friends,
 “requests" : /users/id/requests

}
```

Representation — /users/{id}/requests

```
{
 “requests" : [
  {
   “reqUri" : /users/{id}/requests/{reqId1}
  },
  {
   “reqUri" : /users/{id}/requests/{reqId2}
  }
}
```

Representation — /users/{id}/requests

```
{
 “from”: /users/{id}
}
```

Representation — fb/users/friendships/

```
{
 “friendships" : [
   {
     “first”: /users/{friendId1},
     “second”: /users/{friendId2}
   },
   {
     “first”: /users/{friendId2},
     “second”: /users/{friendId5}
   }
}
```

Representation — fb/users/friendships/{id}

```
{
 “friends" : [
    /users/{friendId1},
    /users/{friendId3},
    /users/{friendId2}
  ]
}
```

Representation — fb/users/friendships/{id};{id}

```
{
  “first”: /users/{friendId1},
  “second”: /users/{friendId2}
}
```

Friend request approval

1. Alice submits a friendship request to Bob:

2. Bob inspects the requests

3. Bob browses Alice’s profile

4. Bob accepts request

Friend request approval

1. ALICE —> `POST: (fb/users/bob/request, {“from”: /users/{alice}})`

2. BOB —> `GET: fb/users/bob/requests/aliceReq`

3. BOB —> `GET: fb/users/bob/requests/{aliceReq.from}`

4. BOB —> `POST: fb/users/frienships{"first": fb/users/alice, “second”: fb/users/bob};`

4. BOB —> `DELETE: fb/users/bob/requests/aliceReq`

ATOM

The Atom Publishing Protocol (APP)

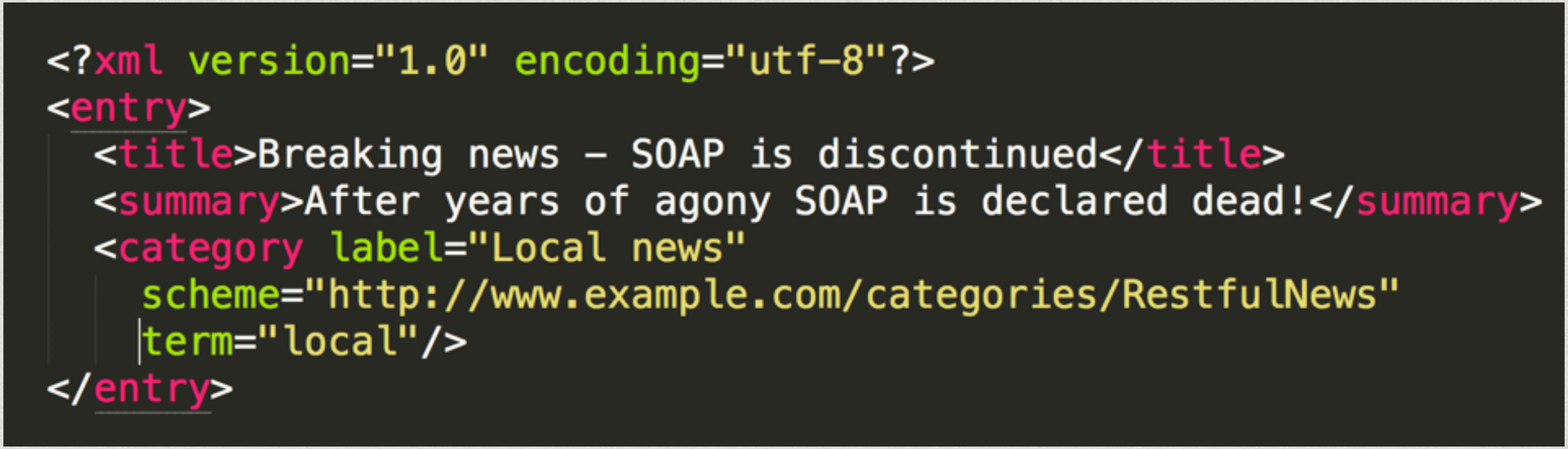
- It’s a protocol built on top of REST
- Defines an XML vocabulary for publishing:
  - authors, summaries categories
- It’s the protocol for RSS feeds
- Is very simple and extensions are responsibility of implementors

Collection

- A list of published items (the RSS feed)
- GET is used to list all the items
- POST to create new item
- PUT & DELETE are not specified they can implemented or not

Member

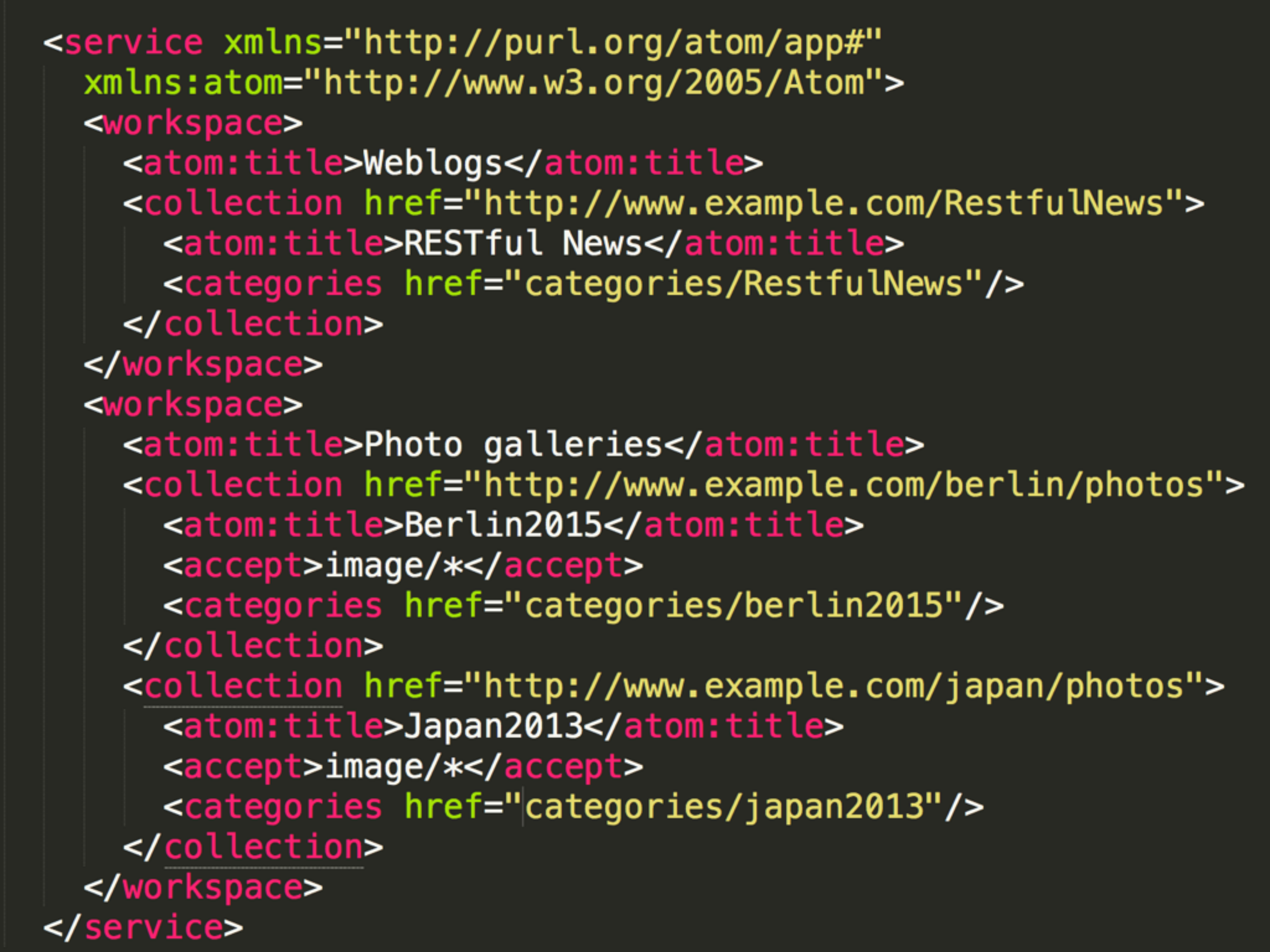
- It’s an entry in the feed
- It’s created through POST



```
<?xml version="1.0" encoding="utf-8"?>
<entry>
  <title>Breaking news - SOAP is discontinued</title>
  <summary>After years of agony SOAP is declared dead!</summary>
  <category label="Local news"
    scheme="http://www.example.com/categories/RestfulNews"
    term="local"/>
</entry>
```

Service Document

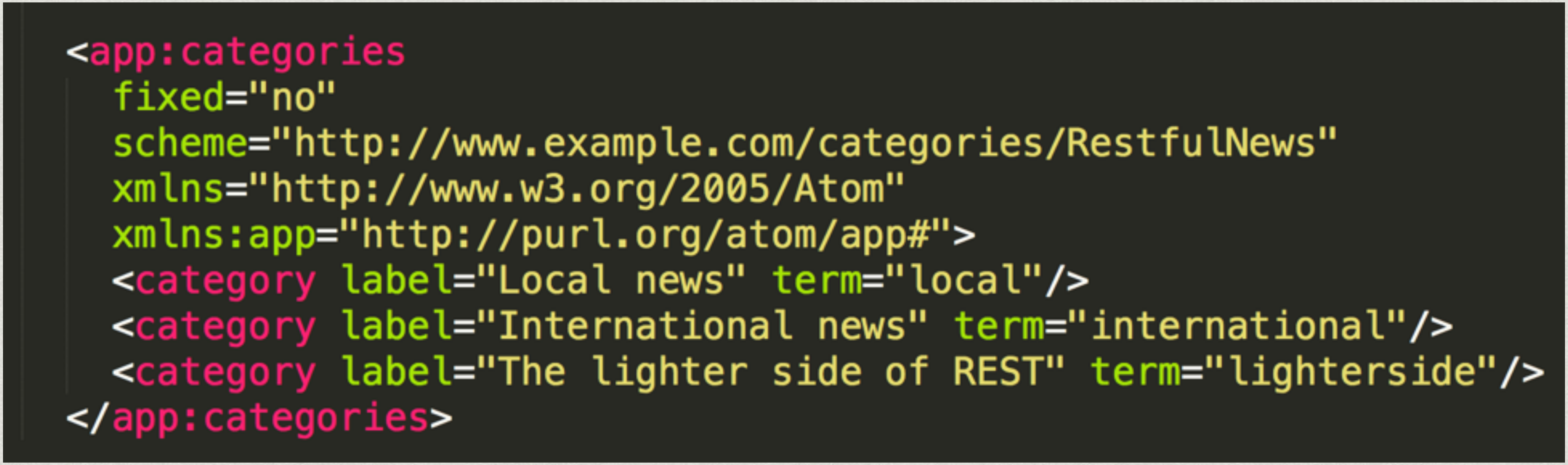
- Gathers several collections
- It’s the home page of an aggregator
- Or a registry
- GET for collection list, POST for new collections



```xml
<service xmlns="http://purl.org/atom/app#"
  xmlns:atom="http://www.w3.org/2005/Atom">
  <workspace>
    <atom:title>Weblogs</atom:title>
    <collection href="http://www.example.com/RestfulNews">
      <atom:title>RESTful News</atom:title>
      <categories href="categories/RestfulNews"/>
    </collection>
  </workspace>
  <workspace>
    <atom:title>Photo galleries</atom:title>
    <collection href="http://www.example.com/berlin/photos">
      <atom:title>Berlin2015</atom:title>
      <accept>image/*</accept>
      <categories href="categories/berlin2015"/>
    </collection>
    <collection href="http://www.example.com/japan/photos">
      <atom:title>Japan2013</atom:title>
      <accept>image/*</accept>
      <categories href="categories/japan2013"/>
    </collection>
  </workspace>
</service>
```

Category Document

- Not al tags defined in service documents must be present (as per spec)
- Only GET is defined on this resource (so it should be defined offline)



```
<app:categories
  fixed="no"
  scheme="http://www.example.com/categories/RestfulNews"
  xmlns="http://www.w3.org/2005/Atom"
  xmlns:app="http://purl.org/atom/app#">
  <category label="Local news" term="local"/>
  <category label="International news" term="international"/>
  <category label="The lighter side of REST" term="lighterside"/>
</app:categories>
```

Summary

- Everything is well thought out, if your problem fits, use it

	GET	POST	PUT	DELETE
Service doc	XML rep	✗	✗	✗
Category doc	XML rep	✗	✗	✗
Collection	Atom feed	new member	✗	✗
Member	Resolve URI	✗	update rep or URI	delete member

TRANSACTIONS

Money transfer

- Should occur in a transaction
- We have 200$ in both accounts/1 & accounts/2
- How do we (safely) move 50$ from 1 to 2?

Resources!!

- `POST transactions/transfer -> 201 created, txId`
- `PUT transactions/transfer/txId/1, balance=150`
- `PUT transactions/transfer/txId/2, balance=250`
- `PUT transactions/transfer/txId, committed=true`

Bibliography

- C2 Architectural Style
- Roy Fielding’s PhD thesis
- REST: I don't Think it Means What You Think it Does — Stefan Tilkov
- RESTful web services — Leonard Richardson & Sam Ruby
- The Atom Publishing Protocol