

Programmable Web Project

Part 2: Programmable Web

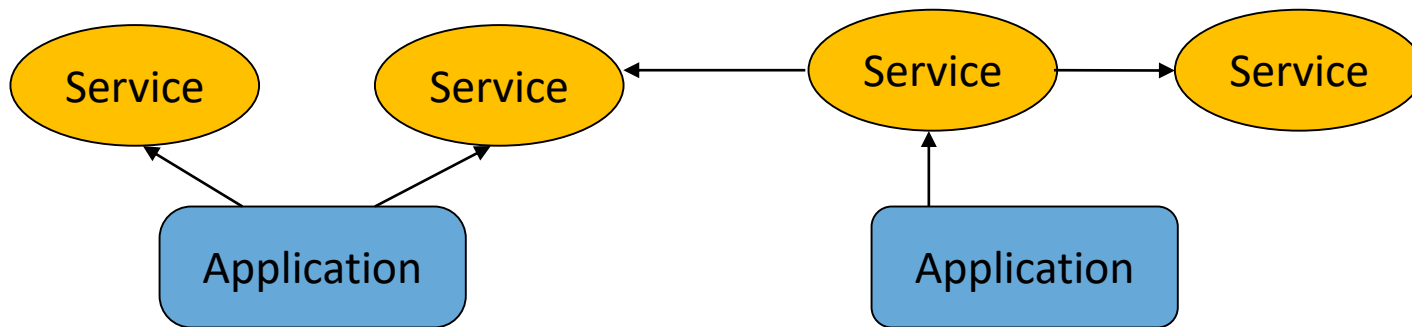
Spring 2020

- **Services and APIs**
- **RESTful Web APIs and HATEOAS**

SERVICES AND APIS

Web services

- Web services are logical units that provides certain functionality.
- They are **application independent**
 - services can be used by other services and applications.
 - services can incorporate the functionality of other services (**composite service**)



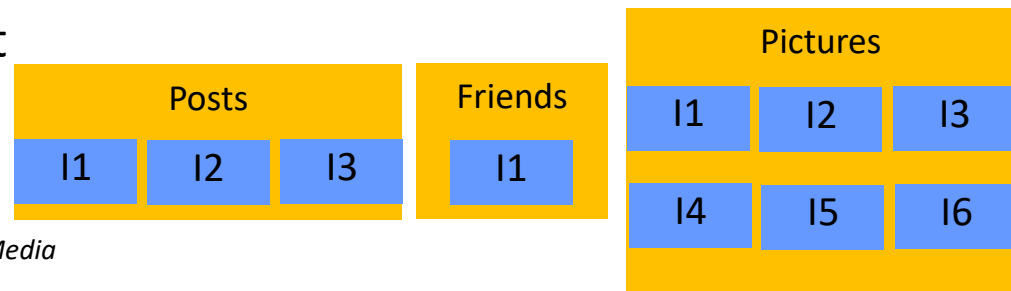
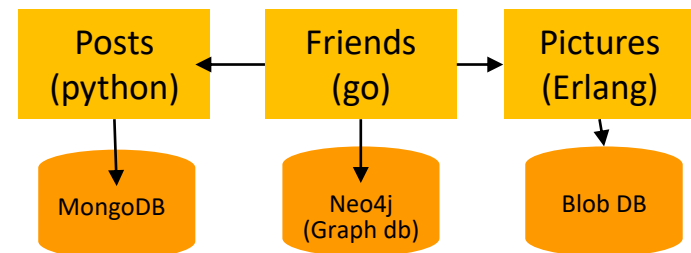
- Services need to communicate to the service consumer:
 - what **functionality** they provide
 - which **data formats** they accept and produce
 - what protocol they use

Microservices

- Set of small and autonomous services that work together.
 - Business boundaries clear defined -> just a piece of functionality
 - The smaller the better -> microservice should be maintained by small team
 - Each microservice runs in its own OS process.
 - Change independently of each other
 - Must be deploy without a change in the consumer.

- Benefits:

- Technology heterogeneity
- Resilience
- Scaling
- Easy of deployment
- Organizational alignment
- Composability

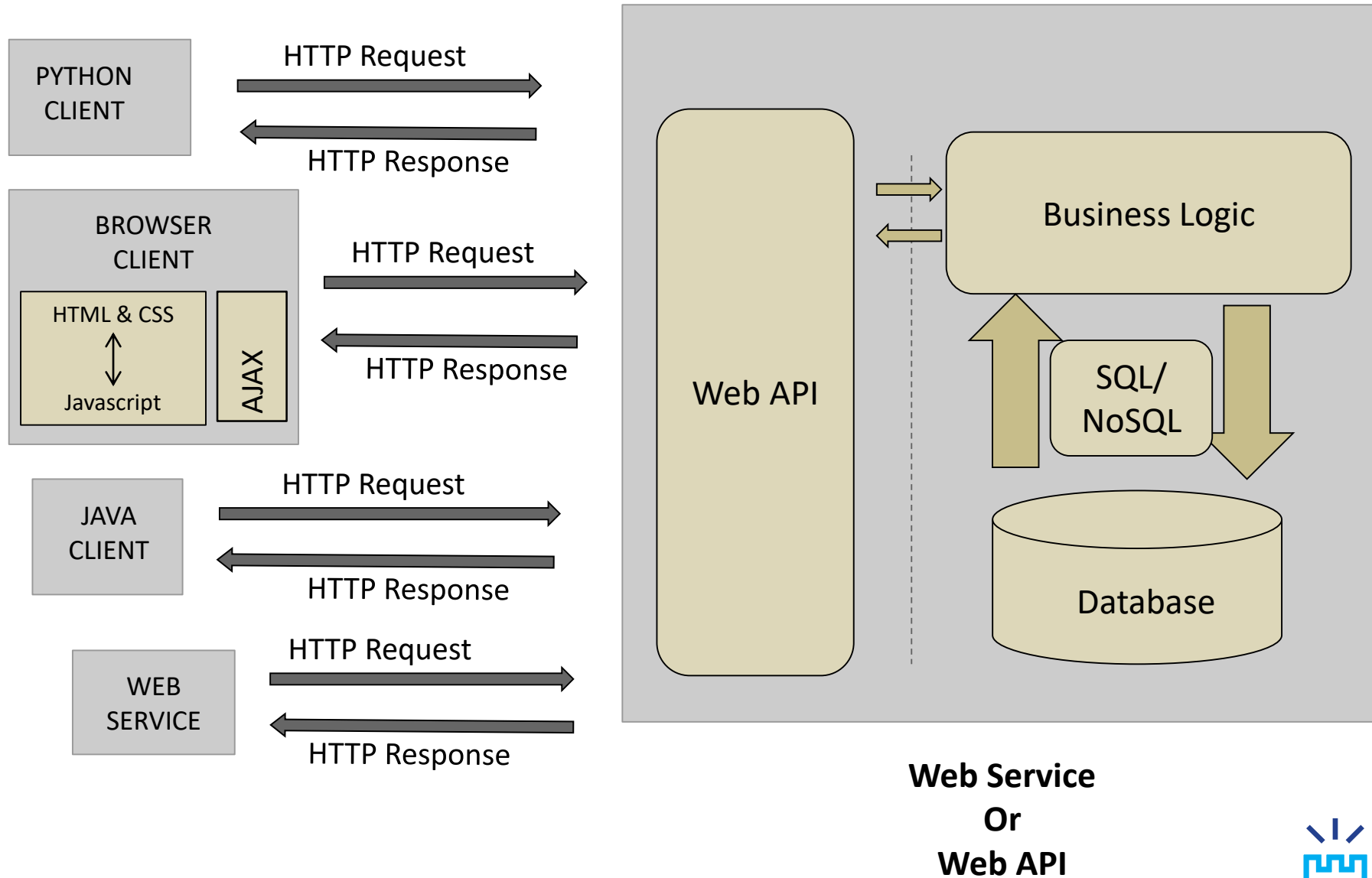


Building microservices. Sam Newman. O'Really Media

APIs and Web APIs

- **Application Programming Interfaces**
- Defines how the service functionality is exposed by means of one or more endpoints:
 - Protocol semantics
 - Application semantics
- **Nowadays, web service word is in disuse => We use Web API instead**

Web API



Website vs Web API

- Gist:

- Github tool that allows sharing code and applications
- Website at: <https://gist.github.com/>
- API at <https://developer.github.com/v3/gists/>
- Gist clients: <https://gist.github.com/defunkt/370230>
 - For instance, Sublime Text client: <https://github.com/condemil/Gist>

Architectural styles

- RPC
- REST
 - CRUD
 - Hypermedia (HATEOAS)
- Pub/Sub (Asynchronous Event-Based Collaboration)

RPC

```
<?xml version="1.0"?>
<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
    <param>
      <value><i4>40</i4></value>
    </param>
  </params>
</methodCall>
```

REST (Representational State Transfer)

- Architectural style proposed by Roy Thomas Fielding.
http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf
- Representation
 - Resource-oriented: operates with resources.
- State:
 - value of all properties of a resource at the certain moment.
- Transfer: State can be transferred
 - Clients can:
 - 1) retrieve the state of a resource and
 - 2) modify the state of the resource

REST APIs

- CRUD
 - Most extended approach. Majority of Web APIs nowadays
 - Not follow strictly REST principles
 - More on this next lecture
- Hypermedia
 - Follows strictly REST principles

Twitter API

The screenshot shows the Twitter Developer API documentation for the `GET statuses/home_timeline` endpoint. The page has a purple header with navigation links: Developer, Use cases, Products, Docs, More, Apply, Apps, and a search icon. A left sidebar lists various API categories like Get Tweet timelines, Curate a collection of Tweets, etc. The main content area includes the endpoint name, a description of what it returns, a note about the 800 tweet limit, a link to 'Working with Timelines', the Resource URL (`https://api.twitter.com/1.1/statuses/home_timeline.json`), Resource Information (JSON response, requires authentication, rate limited), and a table of parameters.

GET statuses/home_timeline

Returns a collection of the most recent [Tweets](#) and Retweets posted by the authenticating user and the users they follow. The home timeline is central to how most users interact with the Twitter service.

Up to 800 Tweets are obtainable on the home timeline. It is more volatile for users that follow many users or follow users who Tweet frequently.

See [Working with Timelines](#) for instructions on traversing timelines efficiently.

Resource URL

```
https://api.twitter.com/1.1/statuses/home_timeline.json
```

Resource Information

| | |
|--------------------------------------|-------------------------|
| Response formats | JSON |
| Requires authentication? | Yes (user context only) |
| Rate limited? | Yes |
| Requests / 15-min window (user auth) | 15 |

Parameters

| Name | Required | Description | Default Value | Example |
|----------|----------|---|---------------|---------|
| count | optional | Specifies the number of records to retrieve. Must be less than or equal to 200. Defaults to 20. The value of count is best thought of as a limit to the number of tweets to return because suspended or deleted content is removed after the count has been applied. | | 5 |
| since_id | optional | Returns results with an ID greater than (that is, more recent than) the specified ID. There are limits to the number of Tweets which can be accessed through the API. If the limit of Tweets has occurred since the since_id, the since_id will be forced to the oldest ID available. | | 12345 |
| max_id | optional | Returns results with an ID less than (that is, older than) or equal to the specified ID. | | 54321 |

<https://developer.twitter.com/en/docs.html>

Pub / Sub

- Some services emit events (user entered the room)
- Some services are subscribed to those events
 - When the publisher publish the events the subscriber receives the event
- Generally a **broker** is in charge of coordination:
 - Producers publish event to the broker
 - Broker handle subscriptions and inform when an event arrives
- Complex solution BUT creates effective loosely-couple solutions.

GraphQL

- Mixed of RPC and REST API concepts
 - Created by Facebook.
- GraphQL is a query language APIs, and a server-side runtime for executing queries by using a type system defined for the data.



<https://graphql.org/>
<https://graphql.org/learn/queries/>

What about current Web APIs (RPC or CRUD)?

- Need excessive documentation
 - Exhaustive description of required protocol: HTTP methods, URLs ...
- Integrating a new API inevitably requires writing custom software
 - Similar applications required totally different clients
- When an application API changes, clients break and have to be fixed
 - For instance a change in the object model in the server or the URL structure => change in the client.
- Clients need to store a lot of information
 - Protocol semantics
 - Application semantics

Web vs Programmable Web

- The **Programmable Web** use the same technologies and communication protocols as the WWW in order to cope with current problems.
- Current differences
 - The data is not delivered necessarily for human consumption (M2M)
 - Nowadays an **specific client** is needed per application at least until we solve the problems derivated from the **semantic challenge**
 - A client can be implemented using any programming language
 - Data is encapsulated and transmitted using any serialization languages such as **JSON, XML, HTML, YAML**

Programmable Web



Which are the resource properties? What can I do next?

How can I communicate with the resource?

Where is the resource? What is its id?

Web:

- Targeted to humans
- One client

Programmable Web:

- Targeted to machines
- Heterogeneous clients

Programmable Web Project

Part 3: RESTful Web APIs

Spring 2020

- **ROA Principles**
- **RESTful Web APIs**
- **Designing RESTful Web APIs**
- **Resource Oriented design vs hypermedia driven design**

INTRODUCTION TO ROA

REST (Representational State Transfer)

- Architectural style proposed by Roy Thomas Fielding.
http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf
- Representation
 - Resource-oriented: operates with resources.
- State:
 - value of all properties of a resource at the certain moment.
- Transfer: State can be transferred
 - Clients can:
 - 1) retrieve the state of a resource and
 - 2) modify the state of the resource

ROA Introduction

- **Resource Oriented Architecture (ROA)**

- Architecture for creating Web APIs
- It conforms the REST design principles
- **Base technologies: URLs, HTTP and Hypermedia**

- **Resource :**

- Anything important enough to be referenced as a thing itself
 - For example: List of the libraries of the city of Oulu, the last software version of Windows, the relation between two friends, the result of factorizing a number
 - Each resource is identified by a unique identifier

- We operate with resources **representations** by means of **HTTP Requests**

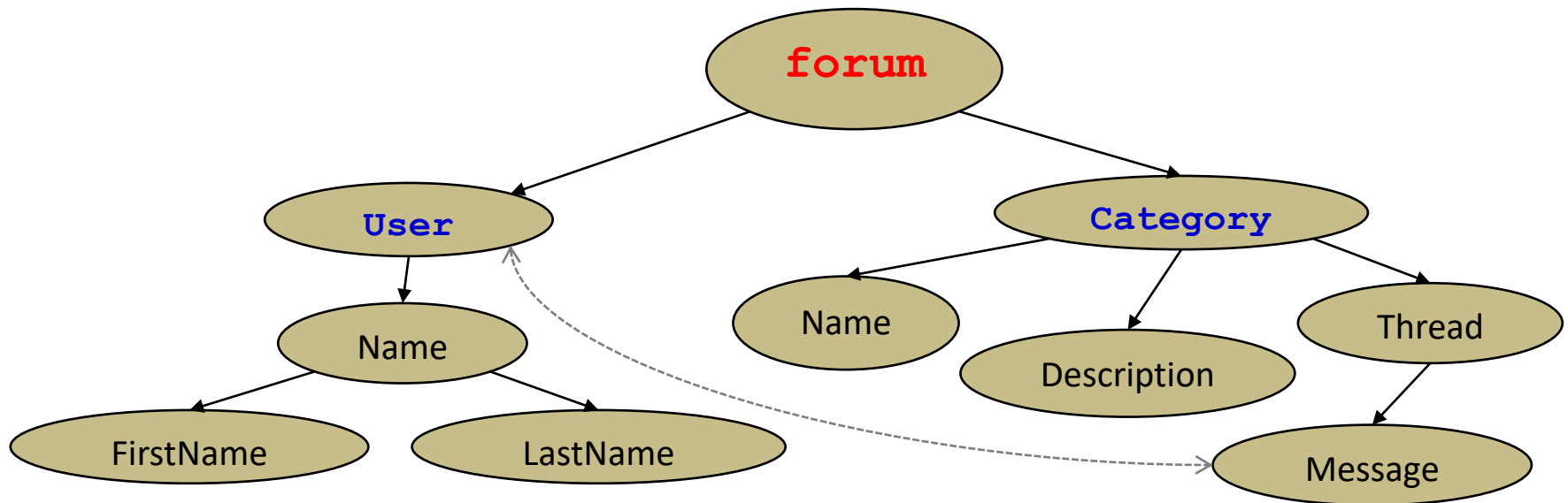
- Retrieve or manipulate the state of the resource

ROA pillars

Four properties:

- 1) Addressability
- 2) Uniform interface
- 3) Statelessness
- 4) Connectedness

Forum Resource hierarchy



Addressability

- Exposes the interesting aspects of its data set as resources
 - Each resource is exposed using its URI
 - The URI can be copied, pasted and distributed
 - Example:
 - **`http://forum.com/users/user1`** refers to the information of the user of the Forum
 - I can send this URI by email, and the receiver can access this information by copying this URI into his/her browser

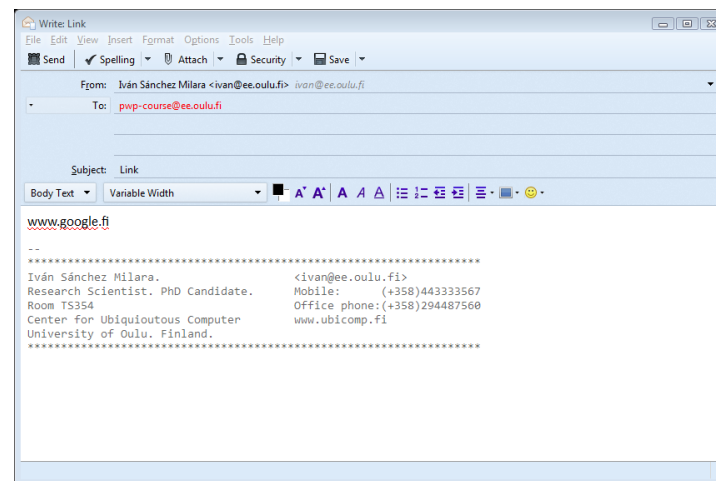
Addressability in WWW

- The WWW is addressable



Google Search I'm Feeling Lucky

Google.fi offered in: suomi svenska



Uniform interface (I)

- Every API uses the same methods with the same meanings
 - Without a uniform interface, clients have to learn how each API is expected to get and send information
- ROA uses uniform interface provided by HTTP to act over the resource provided in the URI

| Method | Description |
|--------|---|
| GET | Returns the resource representation |
| PUT | Changes the state of the resource Creates a new resource when the URL is known |
| POST | Create subordinate resources (no URL known beforehand) Appends information to the current resource state |
| DELETE | Removes a resource from the server |

Uniform interface (II)

- **PATCH** <http://tools.ietf.org/html/rfc5789>
 - Partial edition/modification of a resource
 - Client and server must agree on a new media type for patch documents
 - RFC 6902: proposed standard patch format for JSON.
 - Send a `diff` of the resource representation. Changes to be done to the resource.
 - **Content-Type:** `application/json-patch+json`
 - ```
[{ "op": "remove", "path": "/a/b/c" }, { "op": "add", "path": "/a/b/c",
 "value": ["foo", "bar"] }, { "op": "replace", "path": "/a/b/c",
 "value": 42 }]
```

## Uniform interface (III)

- **URI:** <http://forum.com/messages/msg-3>

```
<msg:Message messageId="msg-3">
 <msg:Title>Edmonton's goalie</msg:Title>
 <msg:Body>Does anyone know where Jussi Markkanen used to play before
 he came to Edmonton Oilers? He was excellent in the Stanley Cup finals
 last season! Too bad they lost...</msg:Body>
 <msg:Sent>2005-09-04T19:22:39+02:00</msg:Sent>
 <msg:SenderIP>217.119.25.162</msg:SenderIP>
 <msg:Registered userID="user-7">
 <user:Nickname>HockeyFan</user:Nickname>
 <user:Avatar file="avatar_7.jpg"/>
 <atom:link rel="self" href="http://forum/users/HockeyFan"/>
 </msg:Registered>
</msg:Message>
```

- **GET:** Retrieves this representation
- **DELETE:** Removes the message with id «msg-3» from the server
- **PUT:** Edits the message with id «msg-3». Title, Body, Sent, SenderIP, and Registered could be modified and **MUST** be included in the request body (The complete representation is sent and it replaces the old one)
- **POST:** Add a response to the message with id «msg-3» (subordinate resource). The body of the request should include the new message

# Uniform interface in WWW

- Only GET and POST supported in HTML
- Rest of HTTP methods supported through Javascript

# Statelessness (I). State concept.

- **Resource state:**

- A resource representation that is exchanged between server and client
- Same for all the clients making simultaneous requests
- Lives in the server

- **Application state:**

- Snapshot of the entire system at a particular instant, including past actions and possible future state transitions
- Future possible application states are informed in the resource representation sent by the server.
- Lives in the client

**STATELESSNESS => REFERS TO APPLICATION STATE**

## Statelessness (II)

- **Every HTTP request happens in complete isolation (STATELESS)**
  - Server never operates based on information from previous requests, **SERVER DOES NOT STORE APPLICATION STATE**
    - *Eg: In a photo album application if I am in “picture 3” I cannot request the “next picture” but “picture 4”*
  - Server considers each client request in isolation and in terms of the current resource state. However it provides information on which are the future states.
  - **Client handles** the application **workflow**

# Statelessness in WWW

- Originally the WWW is statless
  - GET an URL always should return same website
- Multiple applications needs state information (login, last accessed, visited pages)
  - Cookies
  - Session id in URL

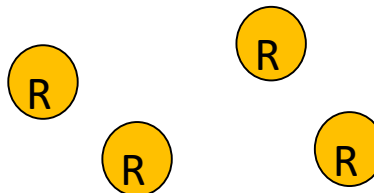
# Connectedness (I)

- Resource representation **MUST** contain links to other resources
- Links must include
  - The relation among resources
  - Optionally, information on how to access linked resources

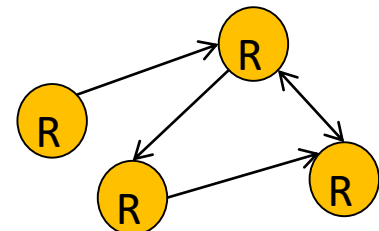
R=Resource



Service exposes everything  
under single URI  
not addressable, not connected



Service is addressable, but not  
connected



Service is addressable  
and connected

# Connectedness (II)

```

<msg:Thread>
 <msg:Message messageId="msg-3">
 <atom:link rel="self" href="http://forum/messages/msg-3"></atom:link>
 <msg:Title>Edmonton's goalie</msg:Title>
 <msg:Registered userID="user-7">
 <user:Nickname>HockeyFan</user:Nickname>
 <user:Avatar file="avatar_7.jpg"/>
 <atom:link rel="self" href="http://forum/users/HockeyFan"/>
 </msg:Registered>
 </msg:Message>
 <msg:Message messageId="msg-7" replyTo="msg-3">
 <atom:link rel="self" href="http://forum/messages/msg-7"/>
 <msg:Title>History</msg:Title>
 <atom:link rel="http://forum/rels/parent-message"
href="http://forum/messages/msg-3"/>
 <msg:Registered userID="user-1">
 <user:Nickname>Mystery</user:Nickname>
 <user:Avatar file="avatar_1.png"/>
 <atom:link rel="self" href="http://forum/users/Mystery"/>
 </msg:Registered>
 </msg:Message>
</msg:Thread>

```

A representation of the message with id «*msg-3*»

A representation of user with nickname «*HockeyFan*»

A representation of the parent message of «*msg-7*»

# Connectedness in WWW

- WWW is connected

- Access and modification of any resource state: following links or filling forms

```

 See the latest messages

```

```

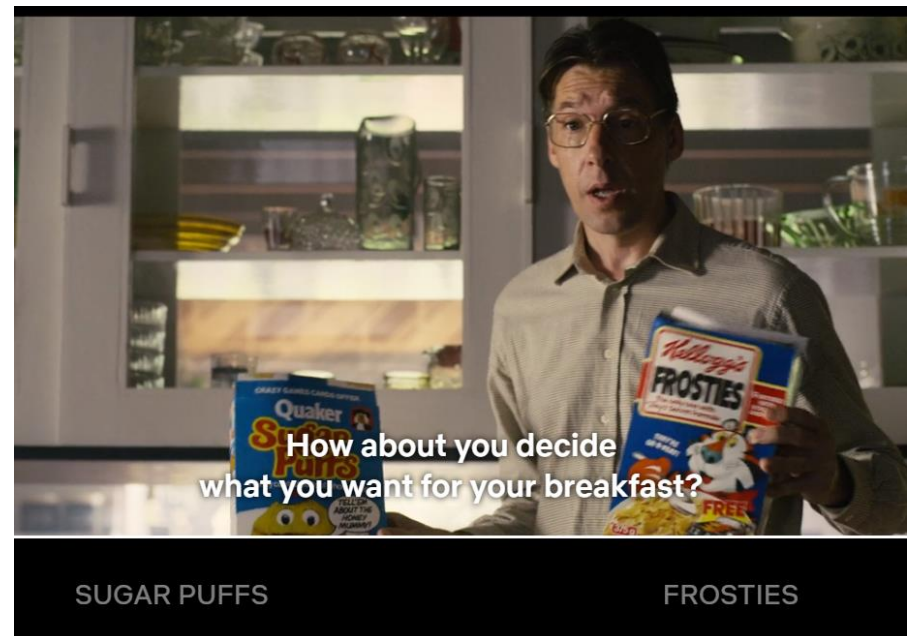
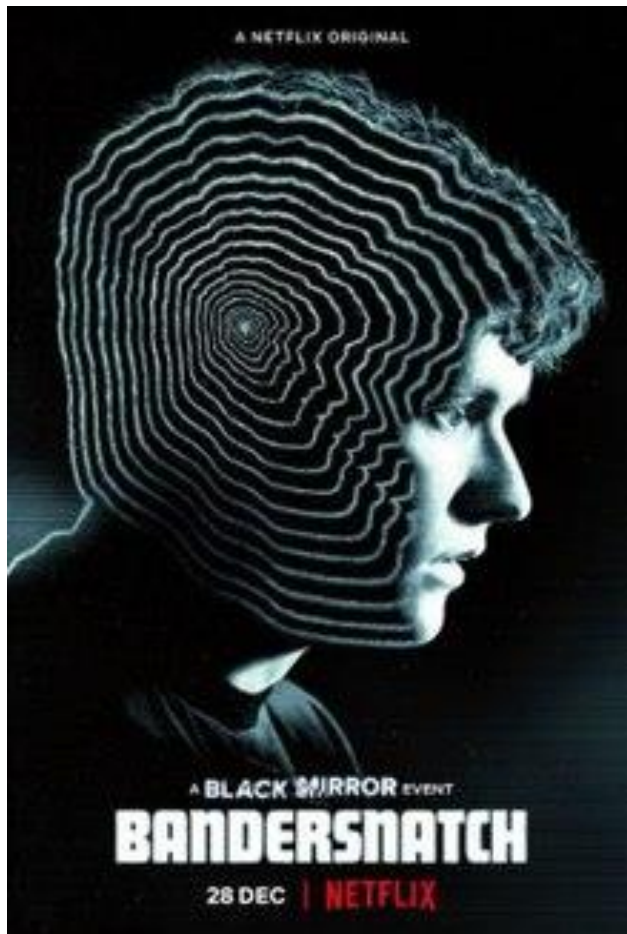
```

```
<form action="http://www.youtypeitwepostit.com/messages"
method="post">
 <input type="text" name="message" value=""
required="true" />
 <input type="submit" value="Post" />
</form>
```

# RESTFUL WEB APIS. HYPERMEDIA.

# HATEOAS

## Hypermedia As The Engine Of Application State

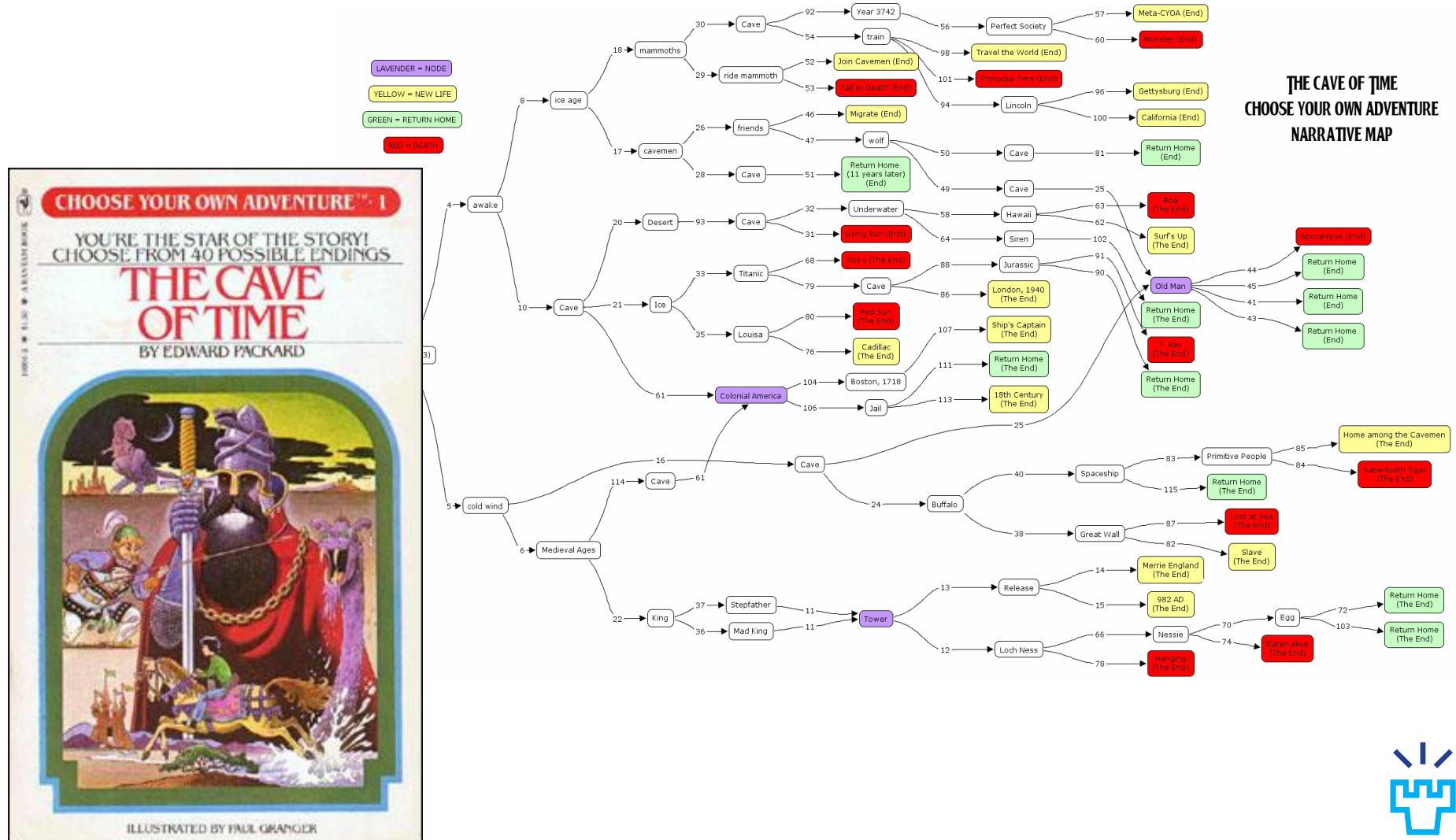


# HATEOAS

## Hypermedia As The Engine Of Application State



# Hypermedia As The Engine Of Application State



# HATEOAS

Hypermedia formats contain:

- Data
- **Hypermedia controls**
  - The URI of the associated resource (link)
  - The relation between both resources
  - Usually, protocol information:
    - Which method I need to execute to access / modify the target resource?
    - What is the format of the request body?
    - ...

```

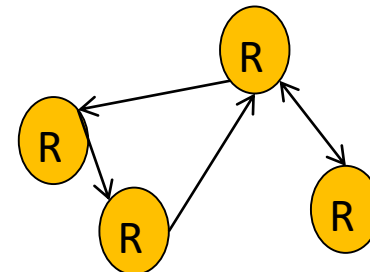
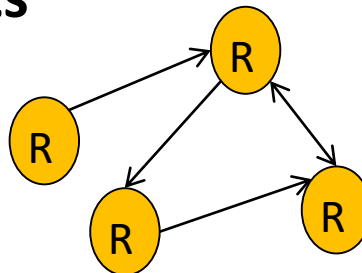
entities" : [
 { "class" : ["switch"],
 "href" : "/switches/4",
 "rel" : ["item"],
 "properties" : { "position" : ["up"] },
 "actions" : [
 { "name" : "flip",
 "href" : "/switches/4",
 "title" : "Flip the mysterious
switch.",
 "method": "POST"
 }
]
 }
]

```

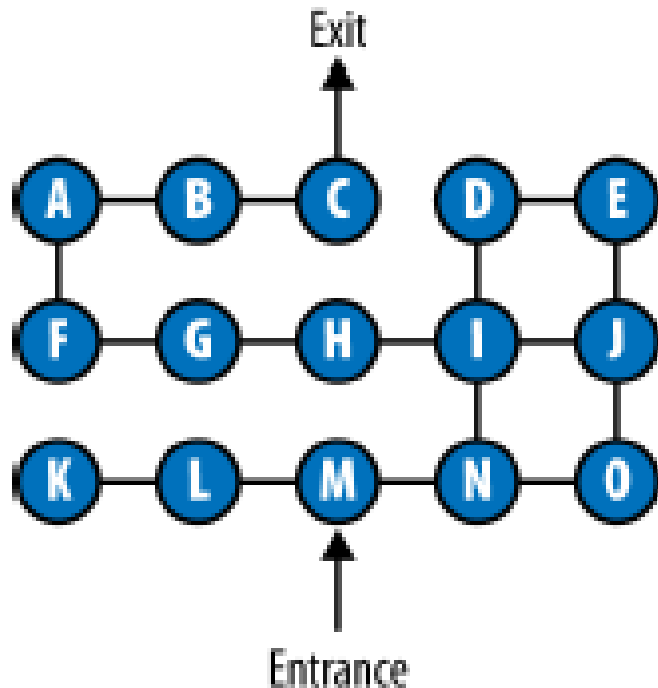
# HATEOAS

## Hypermedia As The Engine Of Application State

- Ideally, client just need the entry point to a service
  - The rest of the URIs (resources) are discovered through the **hypermedia controls**
    - Workflow always informed from the server using the hypermedia controls
  - **RESOURCES AS STATE IN A MACHINE DIAGRAM**
- Well designed RESTful APIs permit **modifying the server architecture (e.g. URL structure) and data model without breaking the clients**



# HATEOAS



RESTful Web APIs. Richardson, Amundsen and Ruby

```
<maze version="1.0">
 <cell href="/cells/M" rel="current">
 <title>The Entrance Hallway</title>
 <link rel="east" href="/cells/N"/>
 <link rel="west" href="/cells/L"/>
 </cell>
</maze>
```

Which are the hypermedia controls?

**Server job is to describe mazes so clients can engage with them  
without dictating any goals**

# Semantic challenge (I)

- In WWW browser does not understand problems domain.
  - Humans process information coming from the server and decide on future actions
- In M2M this is not possible:
  - Machines NEED to understand the problem domain
  - How can we program a computer to make the decisions about which links to follow?
- **This is the biggest challenge in web API design using hypermedia: bridging the semantic gap between understanding a document's structure and understanding its semantics.**

# Semantic Challenge (II)

## Semantic gap

- The gap between the structure of a document and its real-world meaning

### Protocol semantics

- What kind of actions a client can perform?
- Usually solved using **hypermedia control**

### Application semantics

- How the representation is explained in terms of real world concepts.
- Same word might have different meanings in different contexts.
  - E.g. **time**:
    - Preparation time if we are using a recipe book
    - Workout duration if we are building a gym agenda
    - Time of the day if we are using a calendar

# Semantic challenge (III)

Two ways of communicating semantics to the client

**Media Types**

**Profiles**

# Media types

- Defines the format of the message
  - Sometimes include protocol and application semantics
- There are some general purpose media types with hypermedia support:
  - Allows defining the protocol semantics and application semantics in the API
  - **HAL, HTML, SIREN, MASON**



**PLAIN JSON OR PLAIN XML DOES NOT SUPPORT HYPERMEDIA**

**LIST OF HYPERMEDIA FORMATS IN APPENDIX A: Hypermedia formats**

# Media types



## PLAIN JSON OR PLAIN XML DOES NOT SUPPORT HYPERMEDIA

```
<users>
 <user>
 <nickname>Axel</nickname>
 </user>
 <user>
 <nickname>Bob</nickname>
 </user>
</users>
```

```
{users:[
 user:{nickname:"Axel"},
 user:{nickname:"Bob"}
]}
```

```

 "Axel"
 "Bob"

```

## LIST OF HYPERMEDIA FORMATS IN APPENDIX A: Hypermedia formats

# Media Types: Collection+JSON

Mime type: [application/vnd.collection+json](http://amundsen.com/media-types/collection/)

Link: <http://amundsen.com/media-types/collection/>

```
{ "collection":
 {
 "version" : "1.0",
 "href" : "http://www.youtypeitwepostit.com/api/",
 "items" : [
 { "href" : "http://www.youtypeitwepostit.com/api/messages/21818525390699506",
 "data" : [
 { "name" : "text", "value" : "Test." },
 { "name" : "date_posted", "value" : "2013-04-22T05:33:58.930Z" }
],
 "links" : []
 },
 { "href" : "http://www.youtypeitwepostit.com/api/messages/3689331521745771",
 "data" : [
 { "name" : "text", "value" : "Hello." },
 { "name" : "date_posted", "value" : "2013-04-20T12:55:59.685Z" }
],
 "links" : []
 },
 "template" : {
 "data" : [
 { "prompt" : "Text of message", "name" : "text", "value" : "" }
]
 }
]
 }
}
```

## LIST OF HYPERMEDIA FORMATS IN APPENDIX A: Hypermedia formats

# Mason

- Mime-type: application/vnd.mason+json
- Link: <https://github.com/JornWildt/Mason>

```
{
 "name": "eeyore",
 "color": "grey",
 "@controls": {
 "self": {
 "href": "http://api.example.org/donkey/eeyore"
 },
 "dk:mood": {
 "title": "Change mood",
 "href": "http://api.example.org/donkey/eeyore/mood",
 "method": "PUT",
 "encoding": "json",
 "schema": {
 "type": "object",
 "properties": {
 "Mood": {"type": "string"},
 "Reason": {"type": "string"}
 }
 }
 }
 }
}
```

## LIST OF HYPERMEDIA FORMATS IN APPENDIX A: Hypermedia formats

# Profile

- Explains the document semantics that are not covered by its media type.

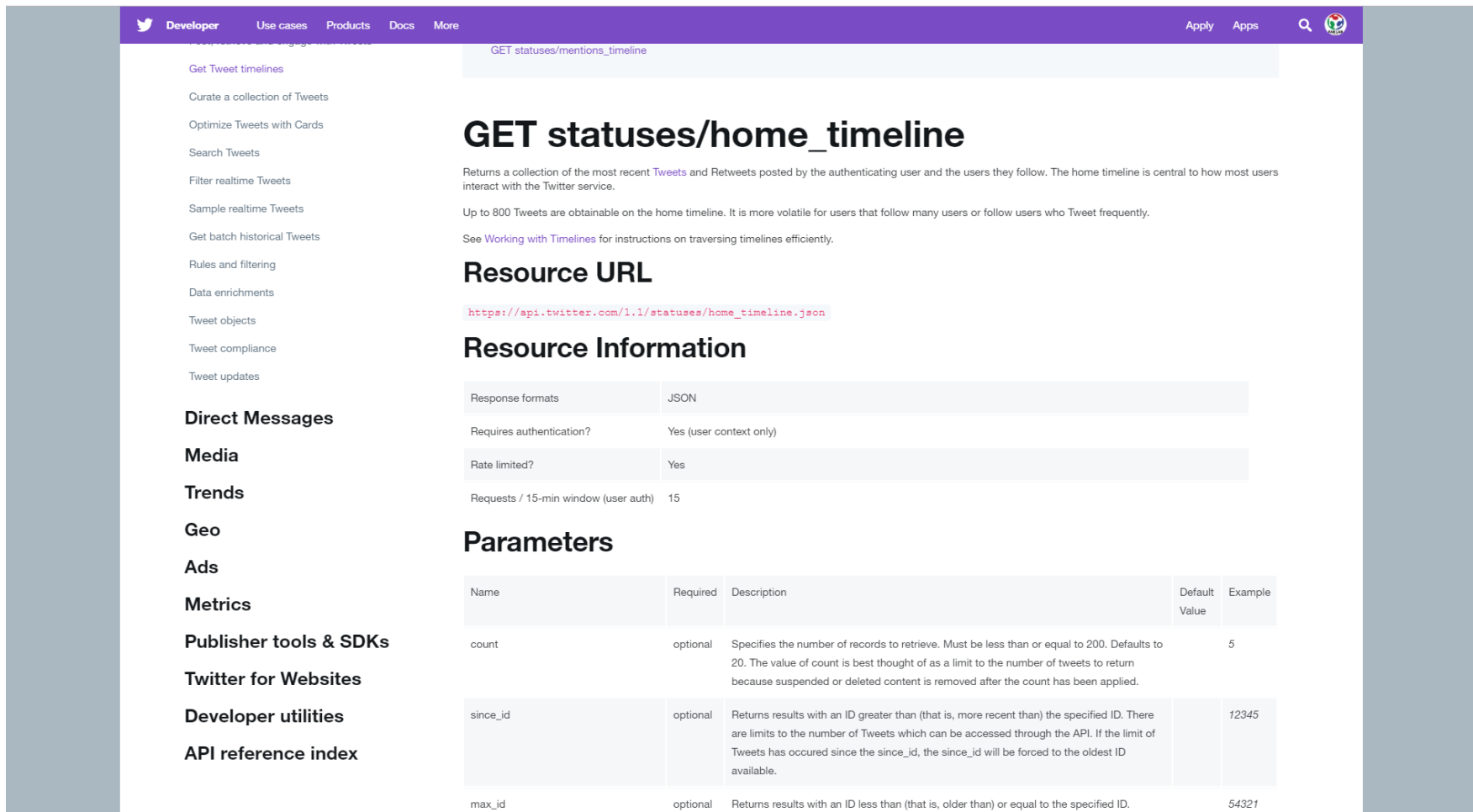
- A profile describes the exact meaning of each **semantic descriptor**

`<span class="fn">Jenny Gallegos</span>`

– *“A profile is defined to not alter the semantics of the resource representation itself, but to allow clients to learn about additional semantics[...] associated with the resource representation, in addition to those defined by the media type”*  
[RFC 6906]

- It is provided to the cliente either defined **in a text document** or using a specific description language: ALPS, JSON-LD, RDF-Schema, XMDP

# Twitter API



The screenshot shows the Twitter Developer API documentation for the `GET statuses/home_timeline` endpoint. The page has a purple header with navigation links: Developer, Use cases, Products, Docs, More, Apply, Apps, and a search icon. A left sidebar lists various API categories like Get Tweet timelines, Curate a collection of Tweets, etc. The main content area includes the endpoint name, a description of what it returns, a note about the 800 tweet limit, a resource URL, and a resource information table. Below that is a parameters table.

## GET statuses/home\_timeline

Returns a collection of the most recent Tweets and Retweets posted by the authenticating user and the users they follow. The home timeline is central to how most users interact with the Twitter service.

Up to 800 Tweets are obtainable on the home timeline. It is more volatile for users that follow many users or follow users who Tweet frequently.

See [Working with Timelines](#) for instructions on traversing timelines efficiently.

### Resource URL

`https://api.twitter.com/1.1/statuses/home_timeline.json`

### Resource Information

|                                      |                         |
|--------------------------------------|-------------------------|
| Response formats                     | JSON                    |
| Requires authentication?             | Yes (user context only) |
| Rate limited?                        | Yes                     |
| Requests / 15-min window (user auth) | 15                      |

### Parameters

| Name     | Required | Description                                                                                                                                                                                                                                                                           | Default Value | Example |
|----------|----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|---------|
| count    | optional | Specifies the number of records to retrieve. Must be less than or equal to 200. Defaults to 20. The value of count is best thought of as a limit to the number of tweets to return because suspended or deleted content is removed after the count has been applied.                  |               | 5       |
| since_id | optional | Returns results with an ID greater than (that is, more recent than) the specified ID. There are limits to the number of Tweets which can be accessed through the API. If the limit of Tweets has occurred since the since_id, the since_id will be forced to the oldest ID available. |               | 12345   |
| max_id   | optional | Returns results with an ID less than (that is, older than) or equal to the specified ID.                                                                                                                                                                                              |               | 54321   |

<https://developer.twitter.com/en/docs.html>

## Richardson Maturity Model

Phil Sturgeon | May 20, 2019

*In the world of HTTP APIs, a REST is made from layers of abstraction on top of RPC to solve certain problems. Each layer builds on the one before it.*

### RPC

Hitting the same endpoint with GET or POST or maybe a combination of both, usually firing around a method and a bunch of arguments. Very few shared conventions from one RPC implementation to another.



### #1: Resources

Every conceptual thing on the Internet now has its own Uniform Resource Identifier, like a nickname for a resource or piece of functionality, which can be referred to later.



### #2: HTTP Methods

Resources can declare their own cacheability now. Resources made things unique, methods make semantics clear. GET can be cached, POST cannot be, etc.



Automatic retries now possible, retrying on a GET = fine, POST = bad, PUT = fine, etc.

No need to invent naming conventions for partial and full updates, can simply use PATCH & PUT

The gRPC "HTTP Bridge" gets here.



### #3: Hypermedia Controls

Instead of an API being just a data store, you turn it into a state machine, providing next available actions via "links", which are relevant for that resource at that moment, instead of forcing clients to interpret state from raw data.

This makes clients thinner, and less prone to inconsistencies from state inference mismatches.

### REST

Just kidding you had a REST API the second you implemented Step 3.

Now go make your SDK better so clients can leverage it, and keep improving and evolving your API.



<https://apisyouwonthate.com/blog/rest-and-hypermedia-in-2019>

Icons by @webalys from streamlineicons.com  
Robot icon made by photo3idea\_studio from www.flaticon.com

# CLIENTS

|    | A                     | B           | C          | D         | E             | F            | G          | H         | I             | J            | K                   | L              | M             |
|----|-----------------------|-------------|------------|-----------|---------------|--------------|------------|-----------|---------------|--------------|---------------------|----------------|---------------|
| 1  |                       |             | Visiting   |           |               |              | Remote     |           |               |              | Visiting > remote ? |                |               |
| 2  | Place                 | Total sales | #sales     | #weeks    | %             | %/week       | #sales     | #weeks    | %             | %/week       | #                   | %              | %/week        |
| 3  | France                | 350         | 220        | 9         | 62.86%        | 6.98%        | 130        | 1         | 37.14%        | 37.14%       | 90                  | 25.71%         | -30.16%       |
| 4  | Poland                | 131         | 51         | 8         | 38.93%        | 4.87%        | 80         | 9         | 61.07%        | 6.79%        | -29                 | -22.14%        | -1.92%        |
| 5  | Belarus               | 19          | 9          | 4         | 47.37%        | 11.84%       | 10         | 2         | 52.63%        | 26.32%       | -1                  | -5.26%         | -14.47%       |
| 6  | <b>TOTAL EUROPE</b>   | <b>350</b>  | <b>220</b> | <b>21</b> | <b>62.86%</b> | <b>2.99%</b> | <b>130</b> | <b>12</b> | <b>37.14%</b> | <b>3.10%</b> | <b>90</b>           | <b>25.71%</b>  | <b>-0.10%</b> |
| 7  | U.S.A.                | 17          | 13         | 2         | 76.47%        | 38.24%       | 4          | 6         | 23.53%        | 3.92%        | 9                   | 52.94%         | 34.31%        |
| 8  | Argentina             | 112         | 42         | 9         | 37.50%        | 4.17%        | 70         | 8         | 62.50%        | 7.81%        | -28                 | -25.00%        | -3.65%        |
| 9  | Mexico                | 114         | 47         | 1         | 41.23%        | 41.23%       | 67         | 3         | 58.77%        | 19.59%       | -20                 | -17.54%        | 21.64%        |
| 10 | <b>TOTAL AMERICAS</b> | <b>725</b>  | <b>247</b> | <b>12</b> | <b>34.07%</b> | <b>2.84%</b> | <b>478</b> | <b>17</b> | <b>65.93%</b> | <b>3.88%</b> | <b>-231</b>         | <b>-31.86%</b> | <b>-1.04%</b> |
| 11 | Egypt                 | 44          | 14         | 1         | 31.82%        | 31.82%       | 30         | 4         | 68.18%        | 17.05%       | -16                 | -36.36%        | 14.77%        |
| 12 | South Africa          | 114         | 27         | 7         | 23.68%        | 3.38%        | 87         | 8         | 76.32%        | 9.54%        | -60                 | -52.63%        | -6.16%        |
| 13 | Nigeria               | 149         | 38         | 4         | 25.50%        | 6.38%        | 111        | 6         | 74.50%        | 12.42%       | -73                 | -48.99%        | -6.04%        |
| 14 | <b>TOTAL AFRICA</b>   | <b>382</b>  | <b>321</b> | <b>12</b> | <b>84.03%</b> | <b>7.00%</b> | <b>61</b>  | <b>18</b> | <b>15.97%</b> | <b>0.89%</b> | <b>260</b>          | <b>68.06%</b>  | <b>6.12%</b>  |
| 15 | China                 | 111         | 102        | 4         | 91.89%        | 22.97%       | 9          | 4         | 8.11%         | 2.03%        | 93                  | 83.78%         | 20.95%        |
| 16 | Indonesia             | 271         | 178        | 7         | 65.68%        | 9.38%        | 93         | 8         | 34.32%        | 4.29%        | 85                  | 31.37%         | 5.09%         |
| 17 | Thailand              | 16          | 5          | 1         | 31.25%        | 31.25%       | 11         | 7         | 68.75%        | 9.82%        | -6                  | -37.50%        | 21.43%        |
| 18 | <b>TOTAL ASIA</b>     | <b>272</b>  | <b>156</b> | <b>12</b> | <b>57.35%</b> | <b>4.78%</b> | <b>116</b> | <b>19</b> | <b>42.65%</b> | <b>2.24%</b> | <b>40</b>           | <b>14.71%</b>  | <b>2.53%</b>  |
| 19 | <b>TOTAL</b>          | <b>1729</b> | <b>944</b> | <b>57</b> | <b>54.60%</b> | <b>0.96%</b> | <b>785</b> | <b>66</b> | <b>45.40%</b> | <b>0.69%</b> | <b>159</b>          | <b>9.20%</b>   | <b>0.27%</b>  |

Spreadsheets are general purposes clients, 'canvas' for creating all sort of solutions

Our goal is to build clients, in which the workflow is not fully hardcoded but build upon the information send by the server

- Clients that do not memorize solution ahead of time
- Are able to adapt to new possible actions as the service presents them
- Are able to adapt to changes in the URLs

# Summary

- REST APIs must be hypertext driven according to Fielding:  
<http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>
- HATEOAS
  - Hypermedia describes the actions that you can perform with the resources.
    - Client does not memorize operations nor workflow. Everything is in the messages
- Documentation reduced drastically: messages are documented by themselves
  - A REST API should spend almost all of its descriptive effort in defining the media type used for representing resources and driving application states
- Easier to create general clients
  - Example: RSS and Atom PUB. Multiple clients can read the same RSS feed.

# DESIGN OF RESTFUL WEB APIS USING ROA

# RESTful Web services design steps

1. Figure out the data set
2. Split the data set into resources
  - Create Hierachy
3. Name the resources with URIs
4. Establish the relations and possible actions among resources
5. Expose a subset of the uniform interface
6. Design the resource representations using hypermedia formats
  1. Define the media types
  2. Define the profiles
7. Define protocol specific attributes
  - E.g. Headers, response code
8. Consider error conditions: What might go wrong?

# Hypermedia driven APIs examples

- **Skype for business:**

- <https://msdn.microsoft.com/en-us/skype/ucwa/hypermedia>

- **Paypal** is promoting the use of Hypermedia in their REST API:

- <https://developer.paypal.com/docs/api/overview/>

- <https://developer.paypal.com/docs/integration/direct/paypal-rest-payment-hateoas-links/>

- **Amazon AppStream:**

- <http://docs.aws.amazon.com/appstream/latest/developerguide/api-reference.html>

- **Foxycart:**

- <https://api.foxycart.com/docs#>

- **Zalando:**

- <http://zalando.github.io/restful-api-guidelines/index.html>