Programmable Web Project
Part 2: Programmable Web

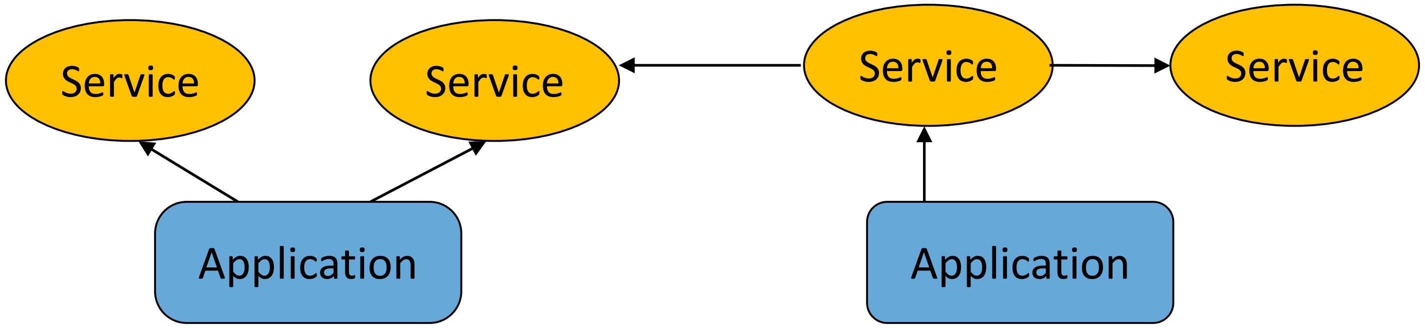
Spring 2020

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

SERVICES AND APIS

Web services

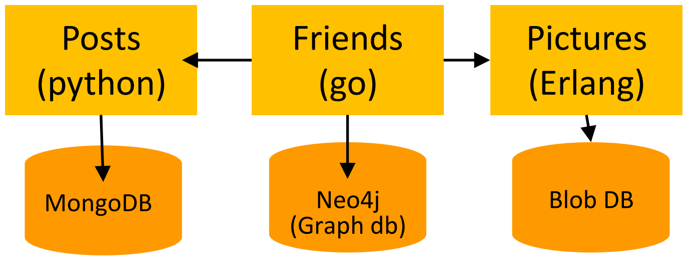
- Web services are logical units that provides certain functionality.
- They are **application independent**
  - services can be used by other services and applications.
  - services can incorporate the functionality of other services (**composite service**)

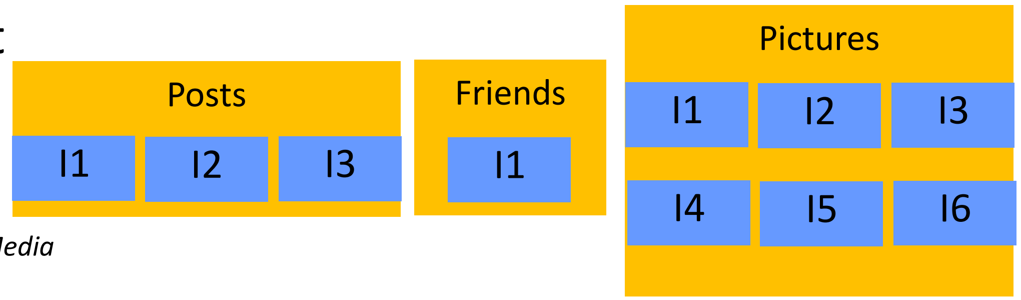


- Services need to communicate to the service consumer:
  - what **functionality** they provide
  - which **data formats** they accept and produce
  - what protocol they use

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

Web services

- Web services are not prepared to human consumption (in contrast to websites).
  - Web services require an architectural style to provide **clear and unambiguous interaction** (clearly defined interfaces), because there’s no smart human being on the client end to keep track.

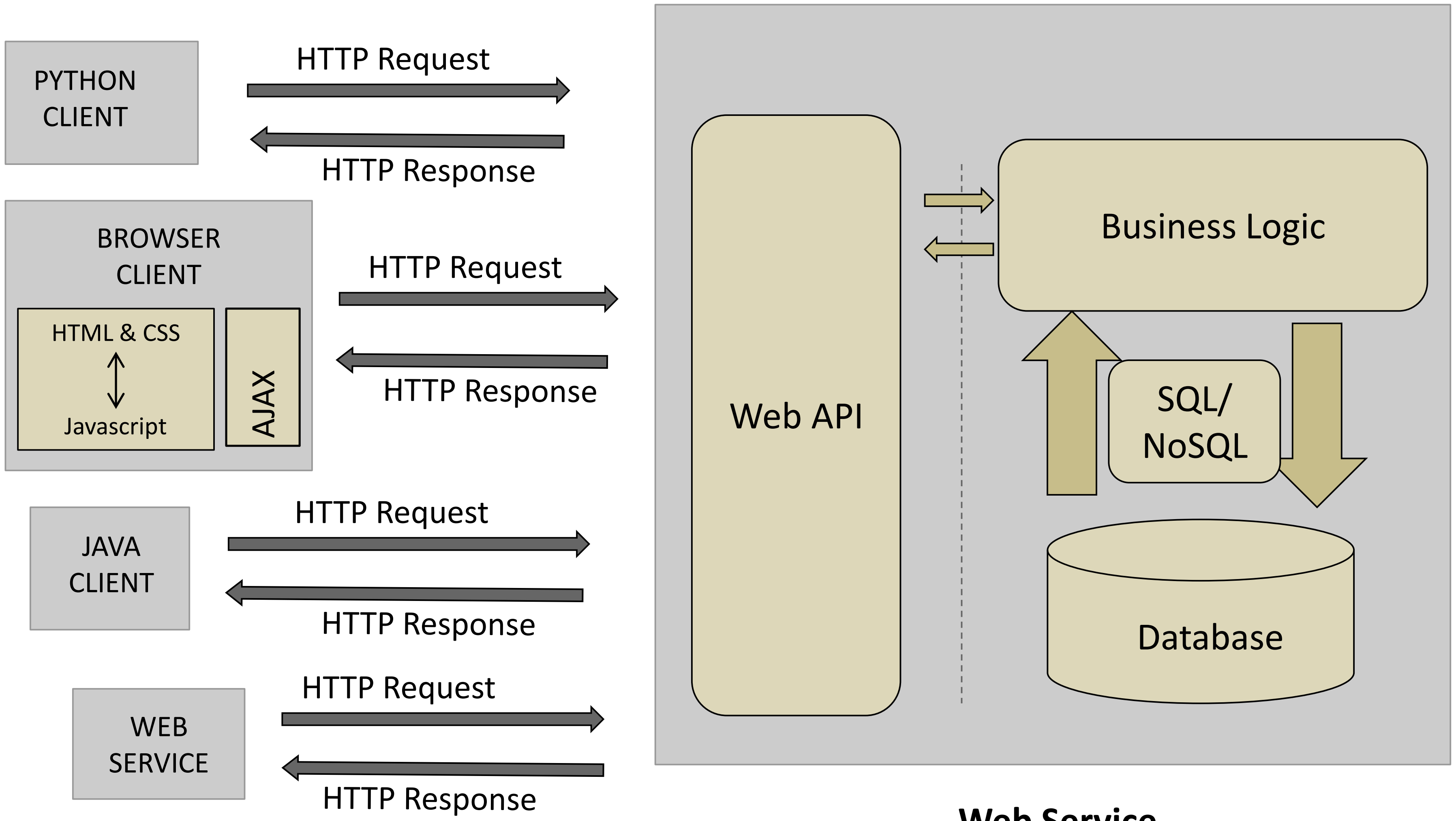
APIs and Web APIs

- **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



Web Service
Or
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

Web APIs Examples.

- **Flickr** Web API can be used to retrieve and upload photos from/to the Flickr sharing service. Pictures can be filtered using multiple criteria.
  https://www.flickr.com/
  https://www.flickr.com/services/api/

- **Blurb!** is a web application that makes easy design, publish, market and sell professional-quality books.
  http://www.blurb.com/flickr

- **Glimmr** is a Flickr viewer for Android. It uses Flickr API to collect data.
  https://play.google.com/store/apps/details?id=com.bourke.glimmr

- Much more in http://www.programmableweb.com

Architectural styles

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

RPC-style Web APIs

- **RPC: Remote procedure call**
  - A method or subroutine is executed in another address space, without the programmer explicitly encoding the details of the remote interaction.
- An RPC-style Web API accepts an envelope full of data from its client, and sends a similar envelope back.
  - The method and the scoping information are kept inside the envelope, or on stickers applied to the envelope.
- Every RPC-style Web API defines a brand new vocabulary: method name, method parameters
- Some examples:
  - XML-RPC
  - SOAP.

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

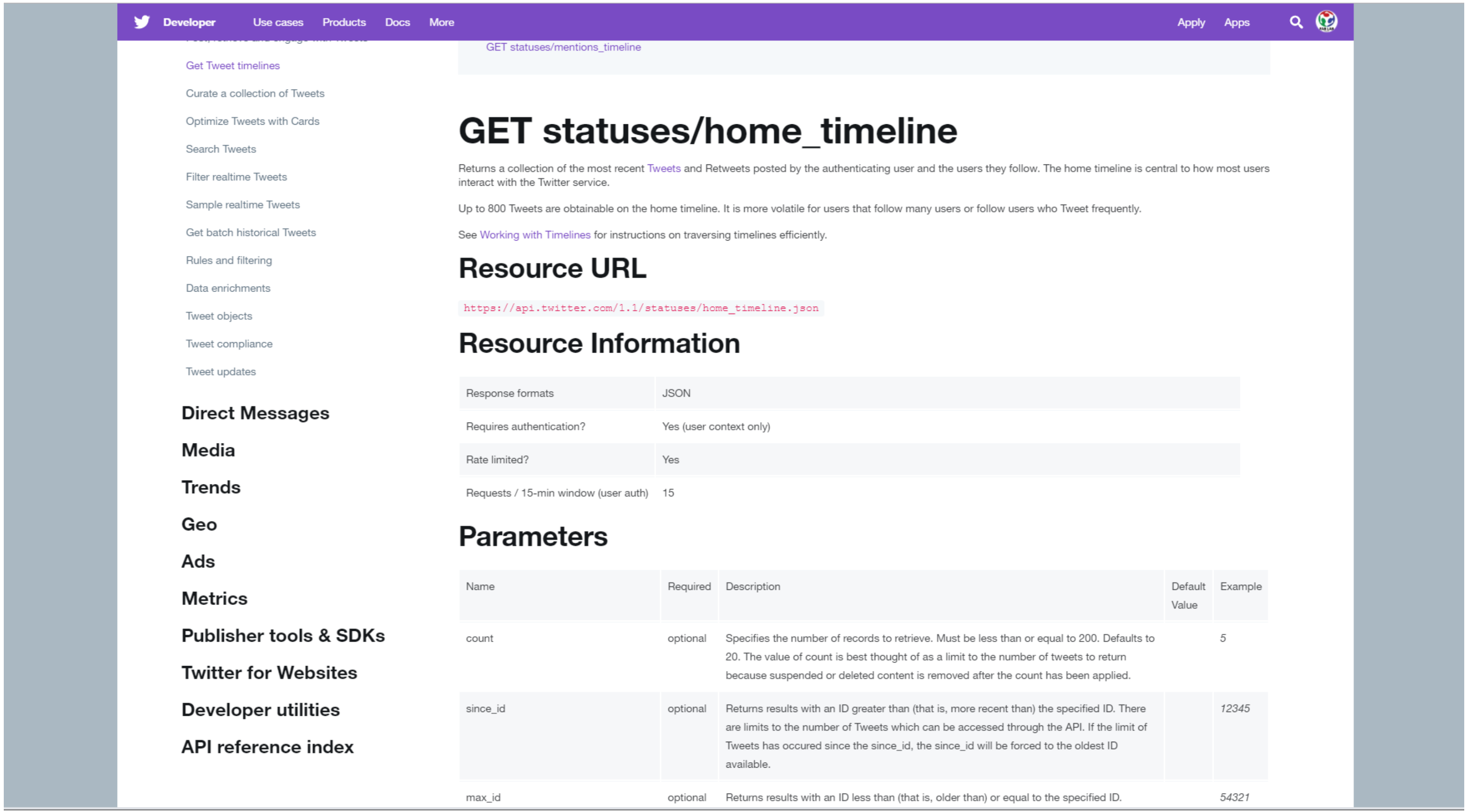
- Representation
  - Resource-oriented: operates with resources.
- **S**tate:
  - value of all properties of a resource at the certain moment.

- **T**ransfer: 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



Get Tweet timelines
Curate a collection of Tweets
Optimize Tweets with Cards
Search Tweets
Filter realtime Tweets
Sample realtime Tweets
Get batch historical Tweets
Rules and filtering
Data enrichments
Tweet objects
Tweet compliance
Tweet updates

Direct Messages
Media
Trends
Geo
Ads
Metrics
Publisher tools & SDKs
Twitter for Websites
Developer utilities
API reference index

GET statuses/mentions_timeline

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 occured 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

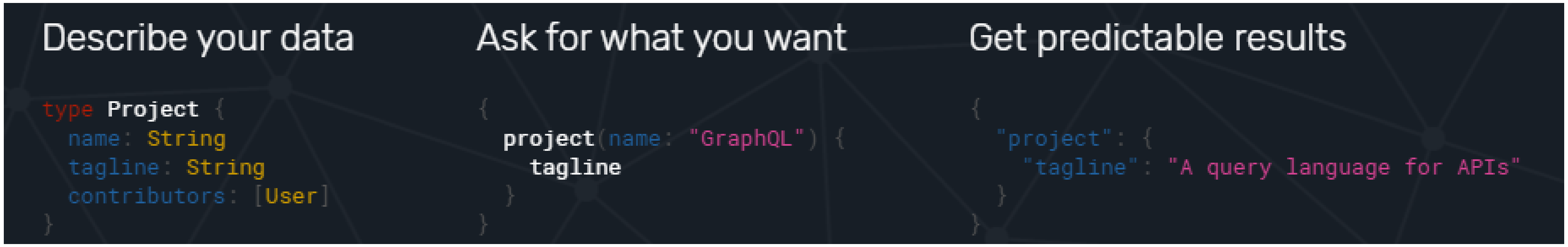
Pub / Sub

- 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.



Describe your data

```
type Project {
  name: String
  tagline: String
  contributors: [User]
}
```

Ask for what you want

```
{
  project(name: "GraphQL") {
    tagline
  }
}
```

Get predictable results

```
{
  "project": {
    "tagline": "A query language for APIs"
  }
}
```

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

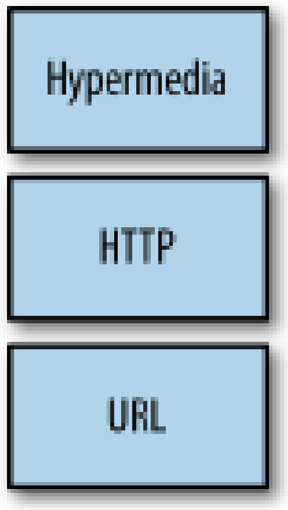
Web vs Programmable Web

- 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**

Hypermedia driven Web APIs

- Follows strictly Fielding dissertation principles.
  - REST APIs must be hypertext driven: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven
- Uses Hypermedia as the Engine of the Application State
  - 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.

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

Hypermedia driven Web APIs

- Follows strictly Fielding dissertation principles.
  - REST APIs must be hypertext driven: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven
- Uses Hypermedia as the Engine of the Application State
  - 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.

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

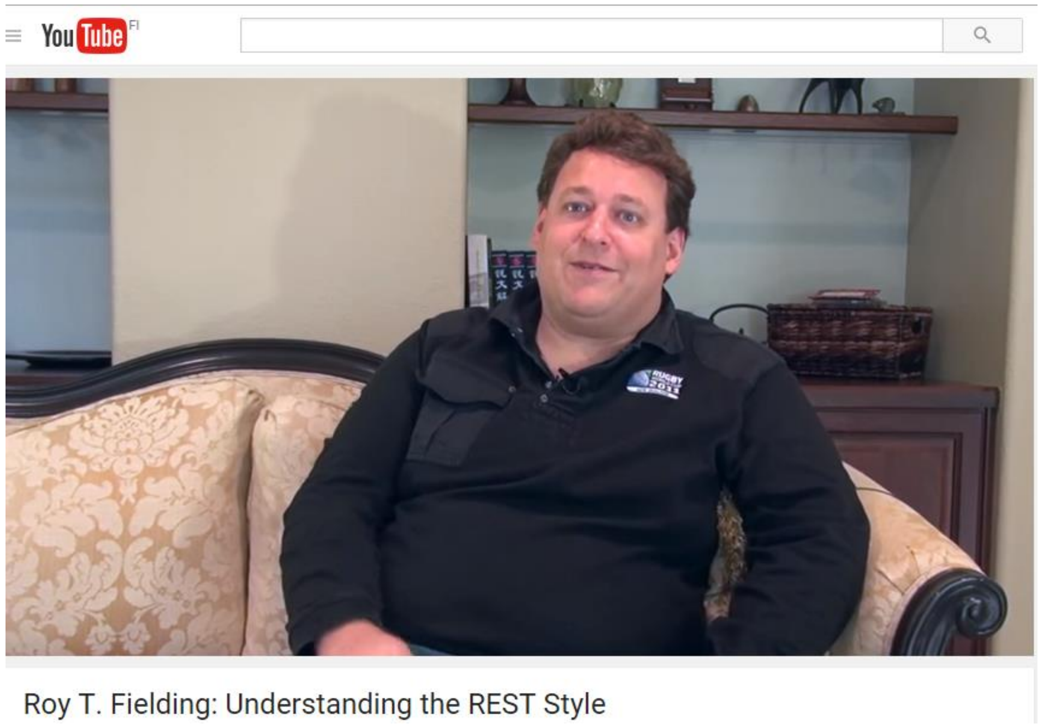
- Representation
  - Resource-oriented: operates with resources.
- **S**tate:
  - value of all properties of a resource at the certain moment.

- **T**ransfer: State can be transferred
  - Clients can:
    1) retrieve the state of a resource and
    2) modify the state of the resource

REST Constraints

- Client-server architecture
- Stateless
- Cacheability
- Layered system
- Code on demand
- Uniform interface

REST



Roy T. Fielding: Understanding the REST Style

https://www.youtube.com/watch?v=w5j2KwzzB-0

ROA Introduction

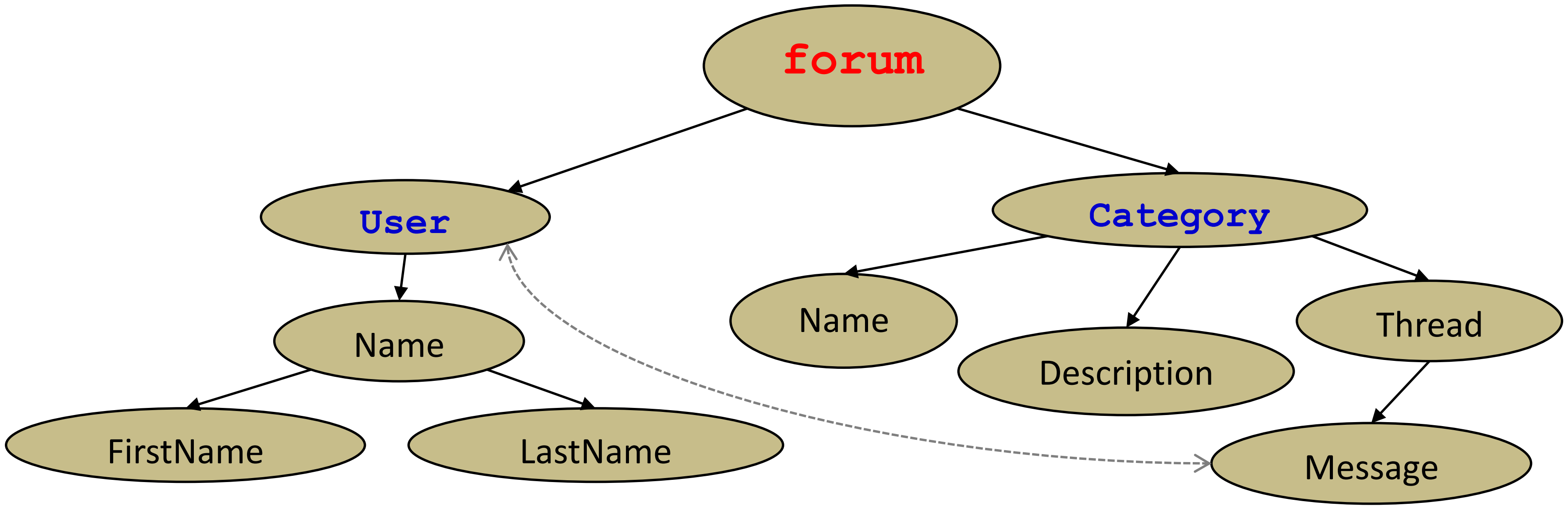
- **Resource Oriented Architecture (ROA)**
  - Architecture for creating Web APIs
  - It conforms the REST design principles
  - **Base technologies: URLs, HTTP and Hypermedia**
- **Resource** :
  - Anything important enough to be referenced as a thing itself
    - For example: List of the libraries 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) Statlessness
4) Connectedness

Forum Resource hierarchy

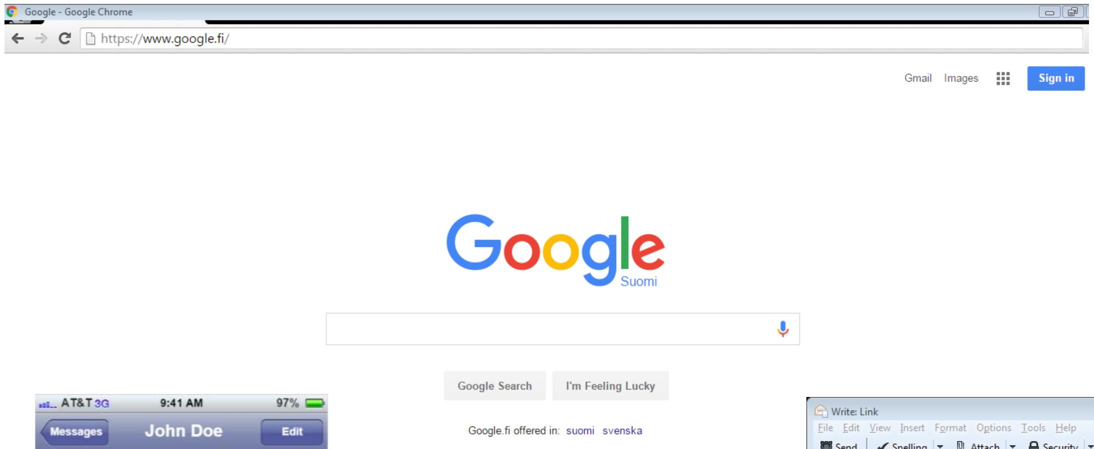


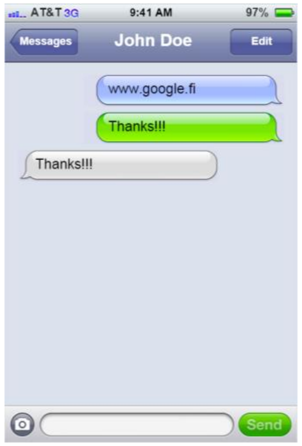
Addressability

- 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

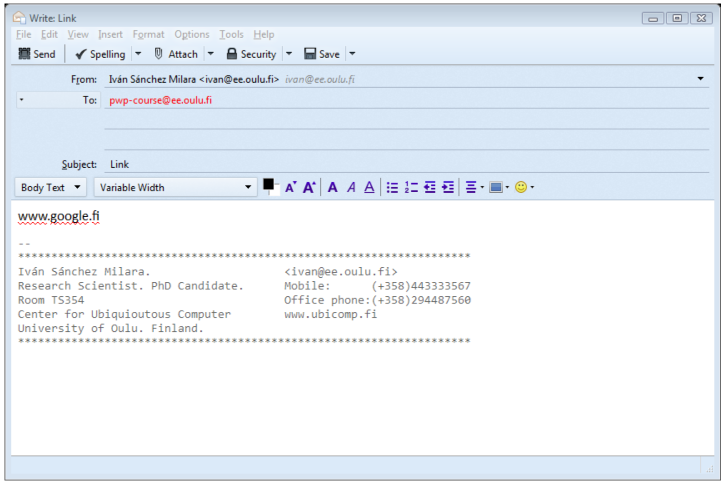




www.google.fi

Thanks!!!

Thanks!!!



From: Iván Sánchez Milara <ivan@ee.oulu.fi> ivan@ee.oulu.fi

To: pwp-course@ee.oulu.fi

Subject: Link

```
www.google.fi

--
************************************************************************
Iván Sánchez Milara.                      <ivan@ee.oulu.fi>
Research Scientist. PhD Candidate.        Mobile:       (+358)443333567
Room TS354                                Office phone:(+358)294487560
Center for Ubiquioutous Computer          www.ubicomp.fi
University of Oulu. Finland.
************************************************************************
```

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)

- **POST**
  - <u>Creates a subordinate resource</u>, that is, a resource existing as a children of another resource
    - Difference with PUT:
      - POST creates new resources when the client does not know their URI
    - Example: A client wants to create a new message in the forum
      - The forum backend generates itself APIs for new messages. Client does not know in advanced.
      - POST HTTP request to **`/forum/categories/categoryName`**
      - The server creates the message and assigns the URI, e.g., **`/forum/messages/message5`**
    - The server sends the URI of the new resource back to the client in the HTTP Response headers
  - <u>Appends information to the current resource state</u>
    - Example: Adding lines to a log entry
    - Difference with PUT:
      - POST modifies just part of the resource state

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 <u>in the server</u>
- **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 <u>in the client</u>

STATLESSNESS => 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**

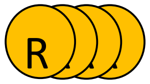
Statelessness in WWW

- 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

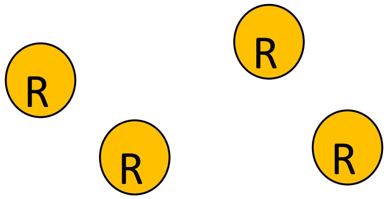
Connectedness (I)

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

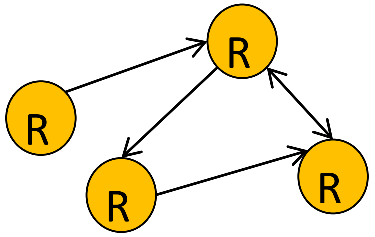
R=Resource



Service exposes everything under single URI
not addressable, not connected



Service is addressable, but not connected



Service is addressable and connected

Connectedness (II)

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*»

```
<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/Mistery"/>
      </msg:Registered>
   </msg:Message>
</msg:Thread>
```

Connectedness in WWW

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

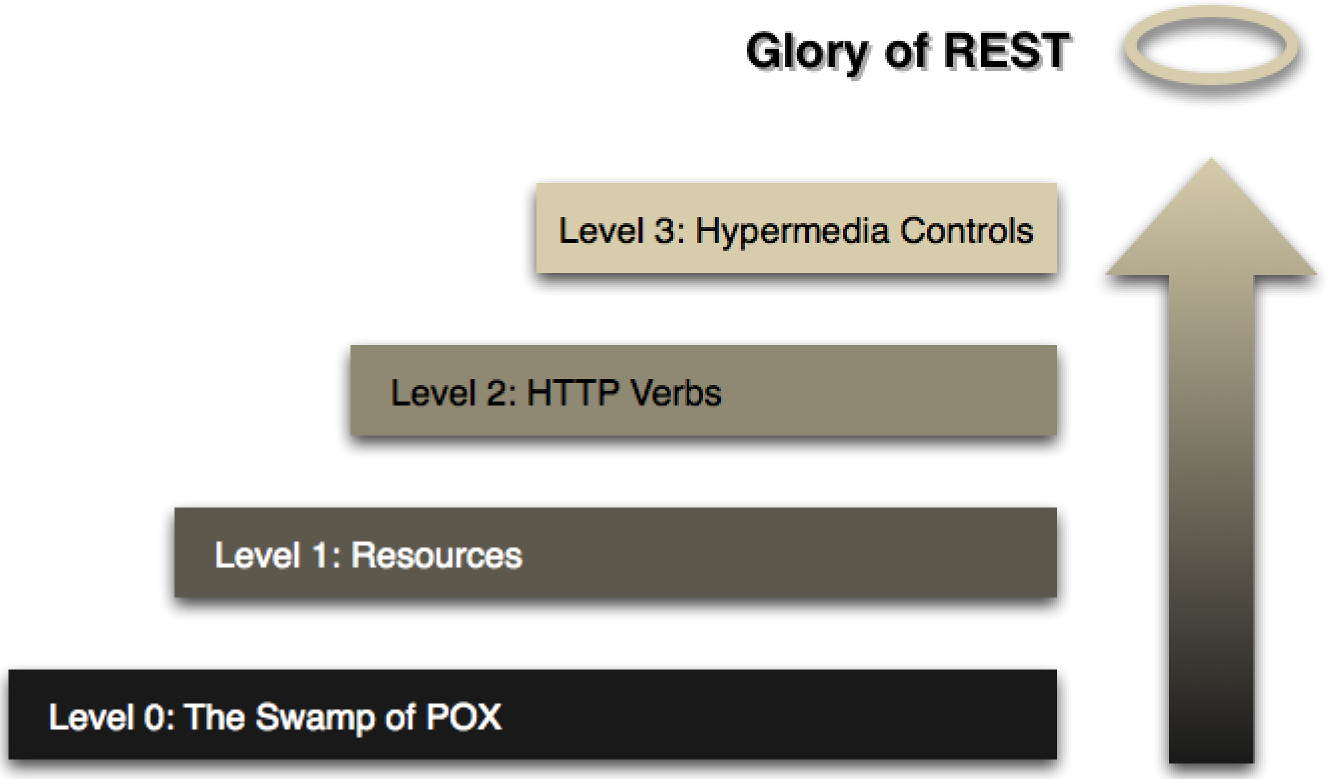
```
<a href="http://www.youtypeitwepostit.com/messages/">
          See the latest messages
</a>
```

```
<img rel="icon" src="http://www.example.com/logo.png" />
```

```
<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.

Richardson Maturity Model



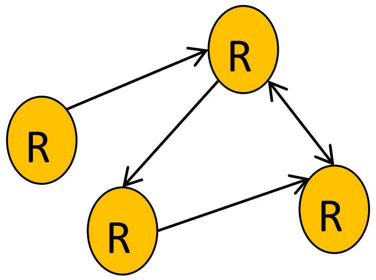
RESTful and Hypermedia

- PROGRAMMABLE WEB goal:
  - Achieve a machine to machine understanding similar the client-server understanding in the web.
    - E.g. Modifying the object model in the server does not affect the server
- RESTful designers forgot one of the principles of REST:
  - *What needs to be done to make the REST architectural style clear on the notion that hypertext is a constraint? In other words, if the engine of application state (and hence the API) is not being driven by hypertext, then it cannot be RESTful and cannot be a REST API.*

    Roy Fielding. REST APIs must be hypertext-driven
- Client does not need to know beforehand workflows or request formats. All that information comes on the server responses.

HATEOAS (I)

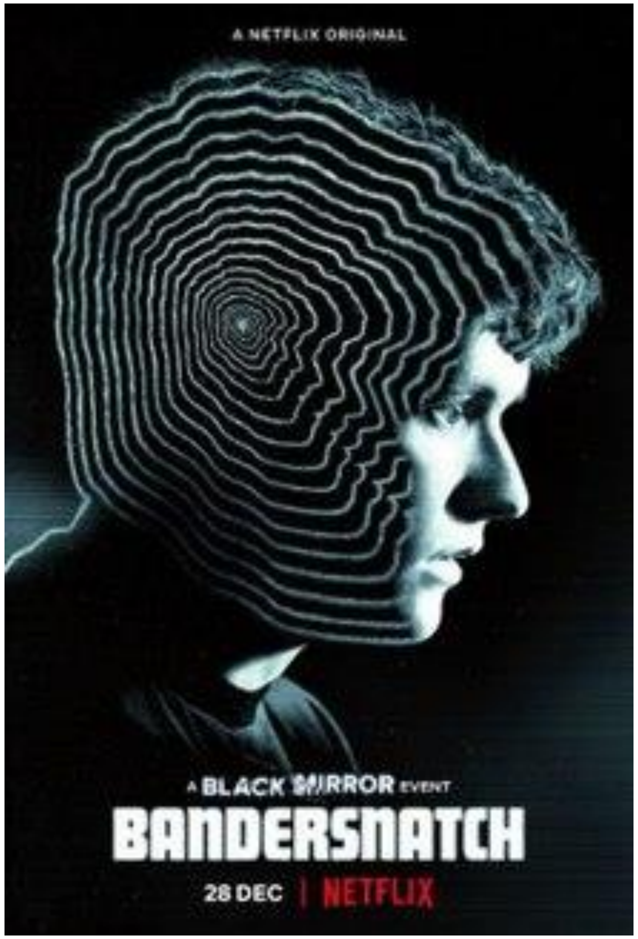
- **Hypermedia As The Engine Of Application State**
  - Hypermedia
    - Techniques to integrate content in multiple formats (text, image, audio, video…) in a way that all content is connected and accessible to the user
  - Engine of Application State
    - **Hypermedia:** Core and driving force of the transformation of the application state
    - The server manipulates the client's state by sending a hypermedia "menu" containing options from which the client is free to choose.
    - Hypermedia contains:
      - Data
      - **Hypermedia controls:**
        - » Enables the state transitions, guiding clients future requests.
        - » Provides protocol semantics: which URL, method, request body is required to perform an application state transition.
        - » Server warrantees workflow control. The hypermedia control:
          - » Describe relationship among resources
          - » Explain who the client should integrate the response into the workflow
        - » In HTML `<a>`, `<img>`, `<script>` and Link header are hypermedia control

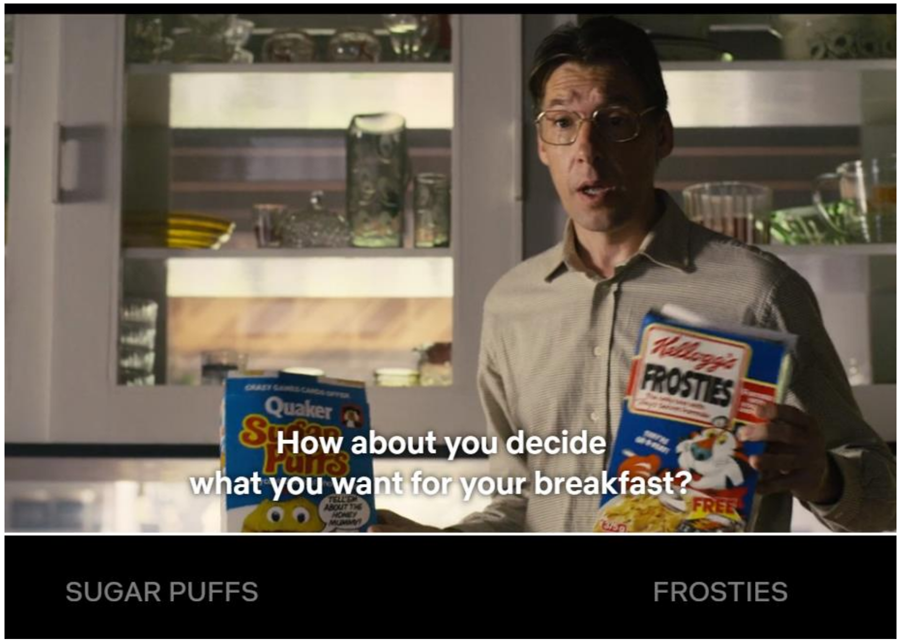


- **Hypermedia drives the application state**

HATEOAS

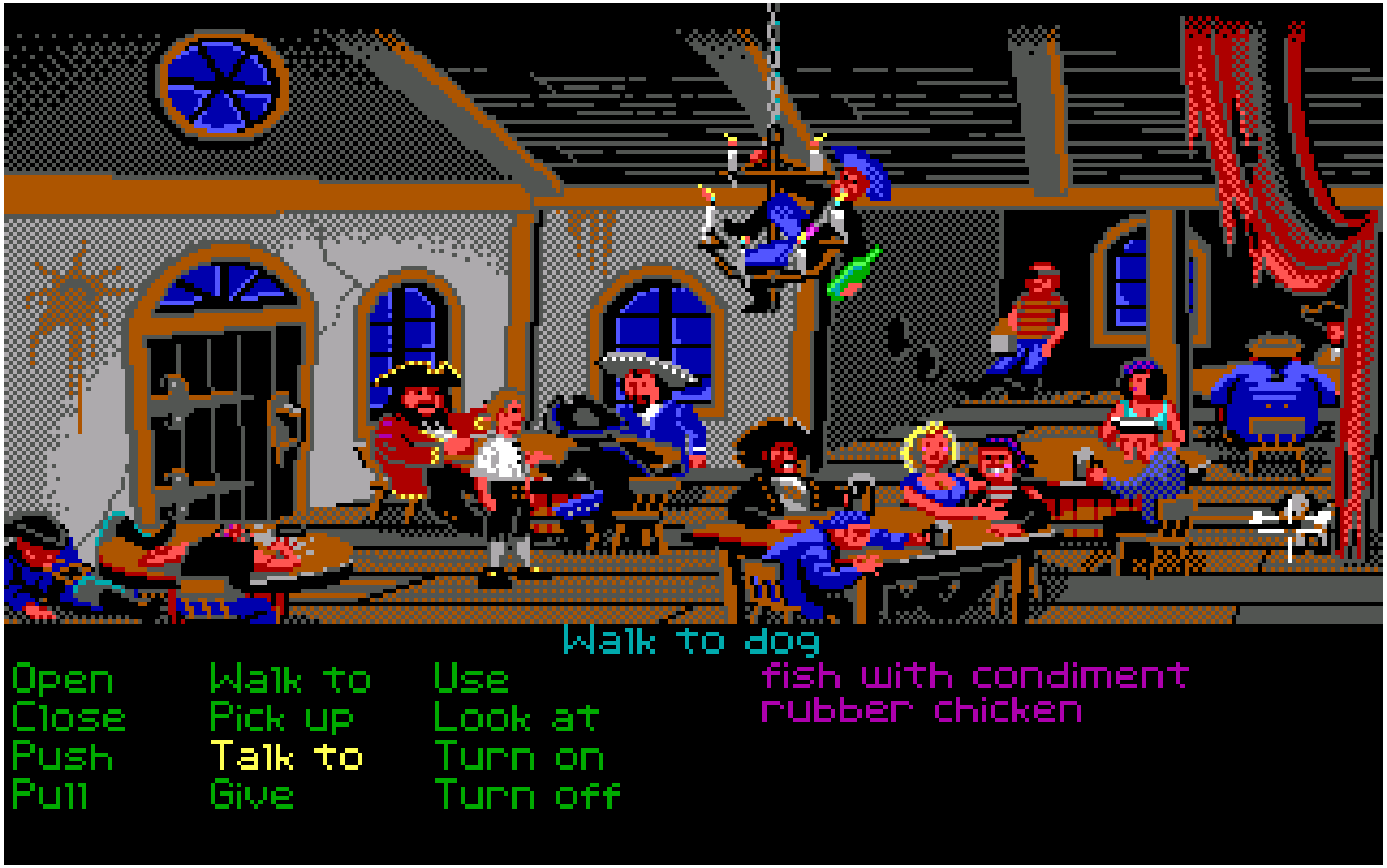
Hypermedia As The Engine Of Application State





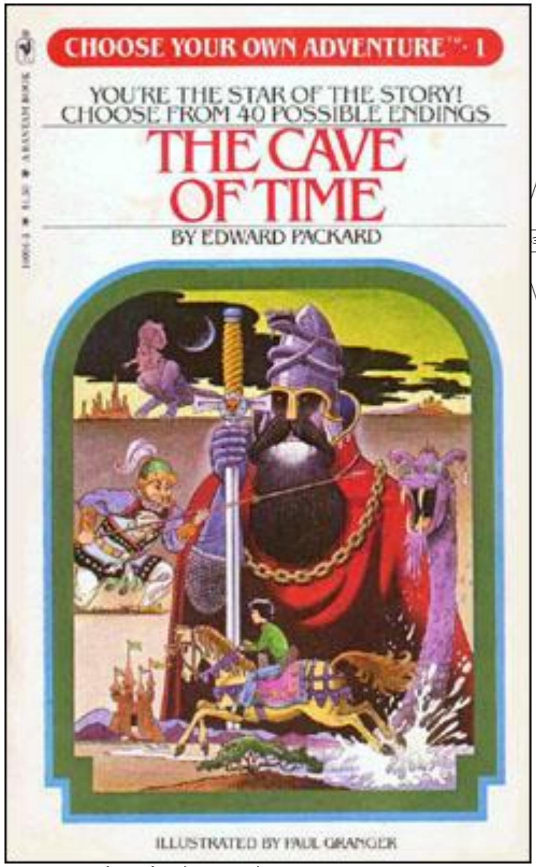
HATEOAS

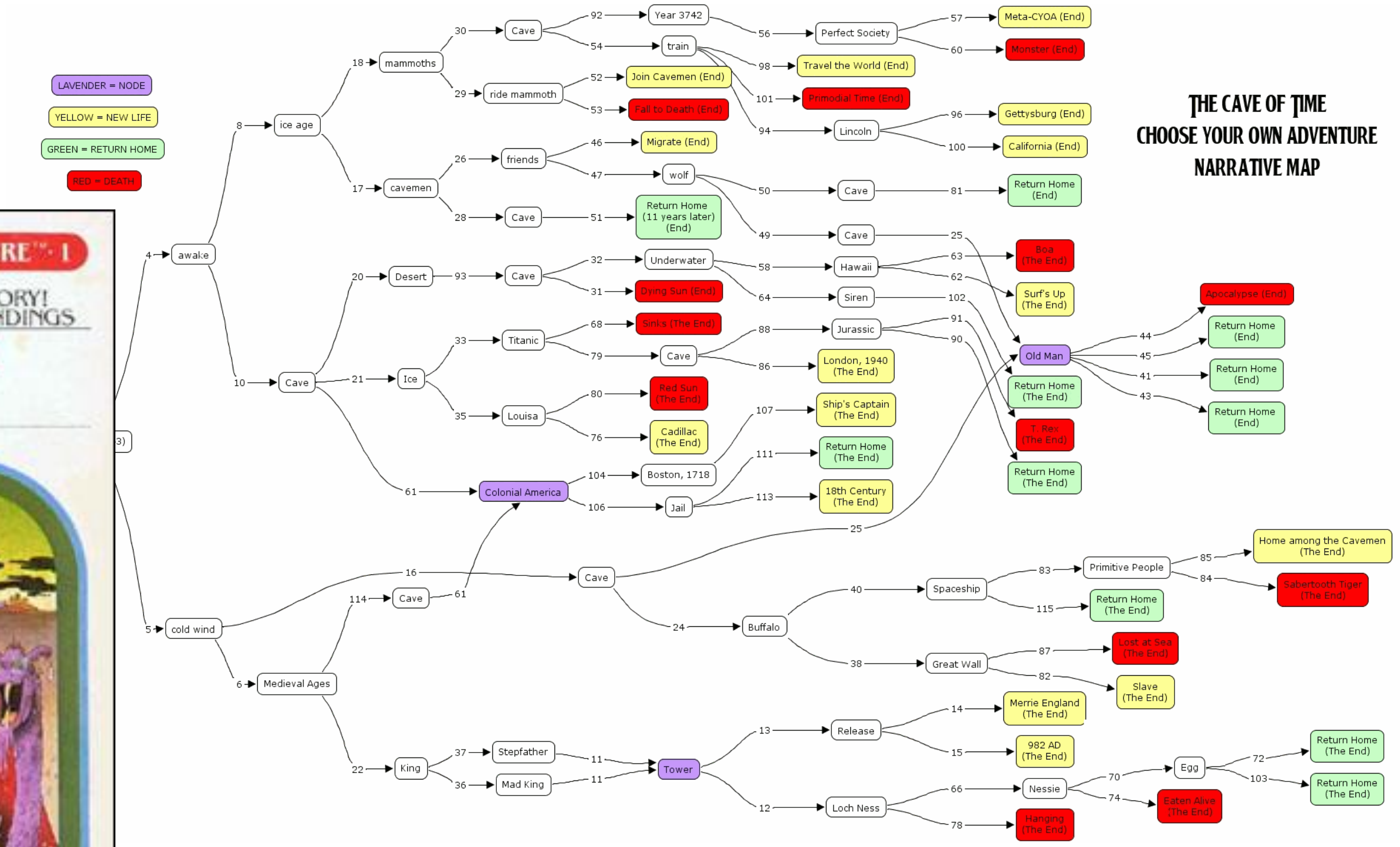
Hypermedia As The Engine Of Application State



HATEOAS

Hypermedia As The Engine Of Application State


CHOOSE YOUR OWN ADVENTURE™ · 1
YOU'RE THE STAR OF THE STORY! CHOOSE FROM 40 POSSIBLE ENDINGS
THE CAVE OF TIME
BY EDWARD PACKARD
ILLUSTRATED BY PAUL GRANGER



Iván Sánchez Milara

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?
    - …

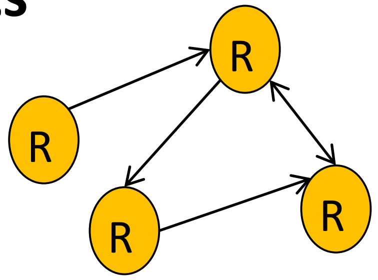
```
entities" : [
  { "class" : ["switch"],
    "href" : "/switches/4",
     "rel" : ["item"],
     "properties" : { "position" : ["up"] },
     "actions" : [
        { "name" : "flip",
          "href" : "/switches/4",
           "title" : "Flip the mysterious
switch.",
            "method": "POST"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

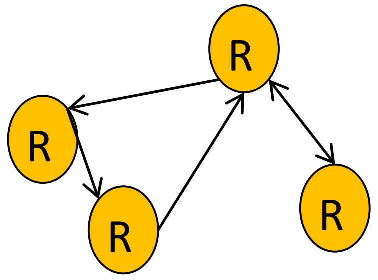
HATEOAS

Hypermedia As The Engine Of Application State

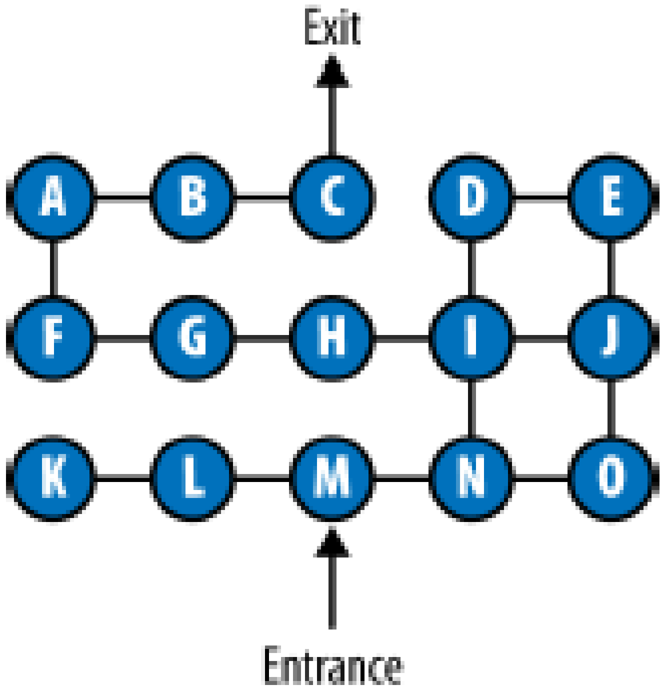
- Ideally, client just need the entry point to a service
  - The rest of the URIs (resources) are discovered through the **hypermedia controls**
    - Workflow always informed from the server using the hypermedia controls
  - **RESOURCES AS STATE IN A MACHINE DIAGRAM**

- 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”}
]}
```

```
<ul>
           <li><a href=”http://myapp/users/axel” rel=”self”>”Axel”</a></li>
           <li><a href=”http://myapp/users/bob” rel=”self”>”Bob”</a></li>
</ul>
```

LIST OF HYPERMEDIA FORMATS IN APPENDIX A: Hypermedia formats

Media Types: Collection+JSON

Mime type: application/vnd.collection+json
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

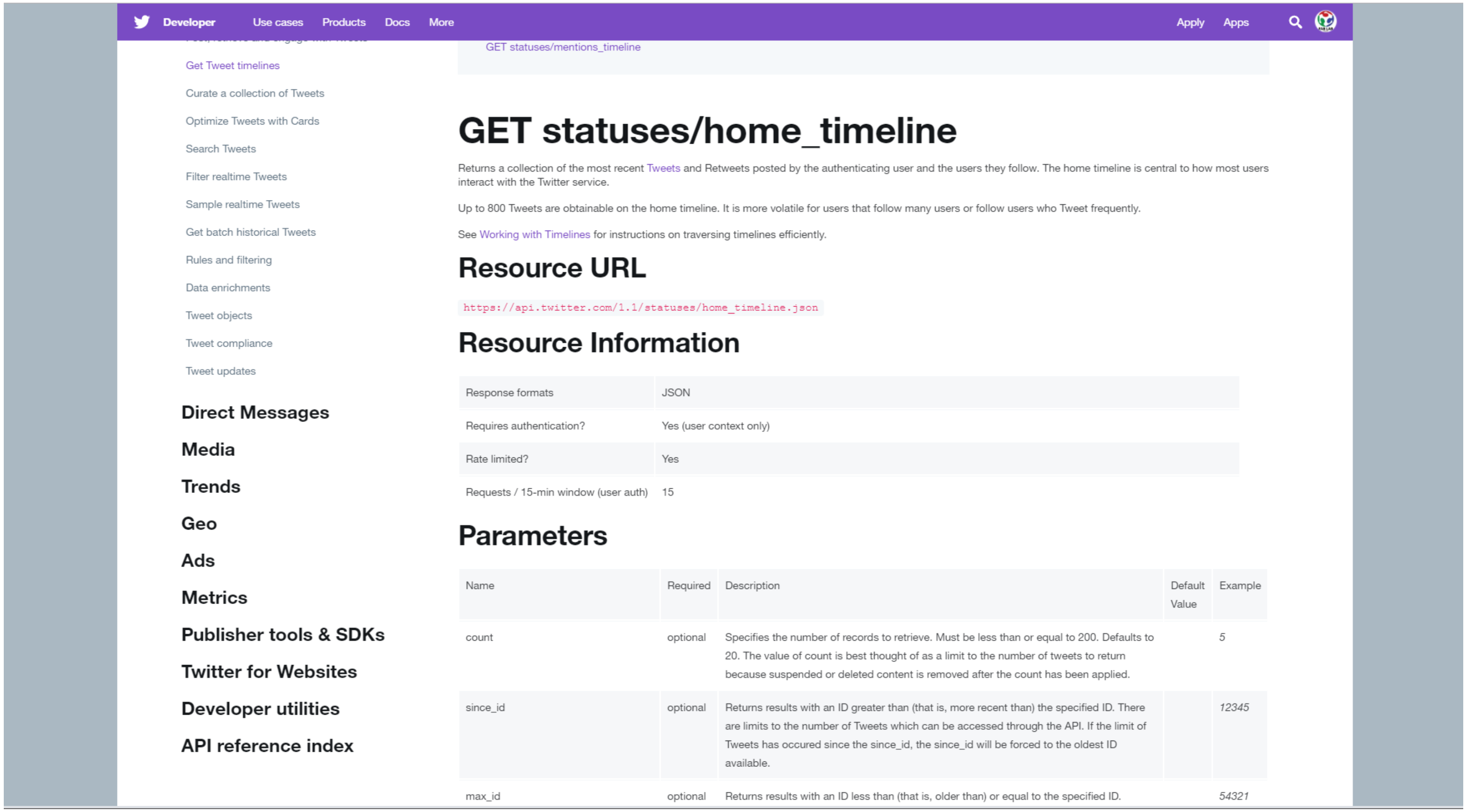
- Explains the document semantics that are not covered by its media type.
    - A profile describes the exact meaning of each **semantic descriptor**

```
<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



Get Tweet timelines
Curate a collection of Tweets
Optimize Tweets with Cards
Search Tweets
Filter realtime Tweets
Sample realtime Tweets
Get batch historical Tweets
Rules and filtering
Data enrichments
Tweet objects
Tweet compliance
Tweet updates

Direct Messages
Media
Trends
Geo
Ads
Metrics
Publisher tools & SDKs
Twitter for Websites
Developer utilities
API reference index

GET statuses/mentions_timeline

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`

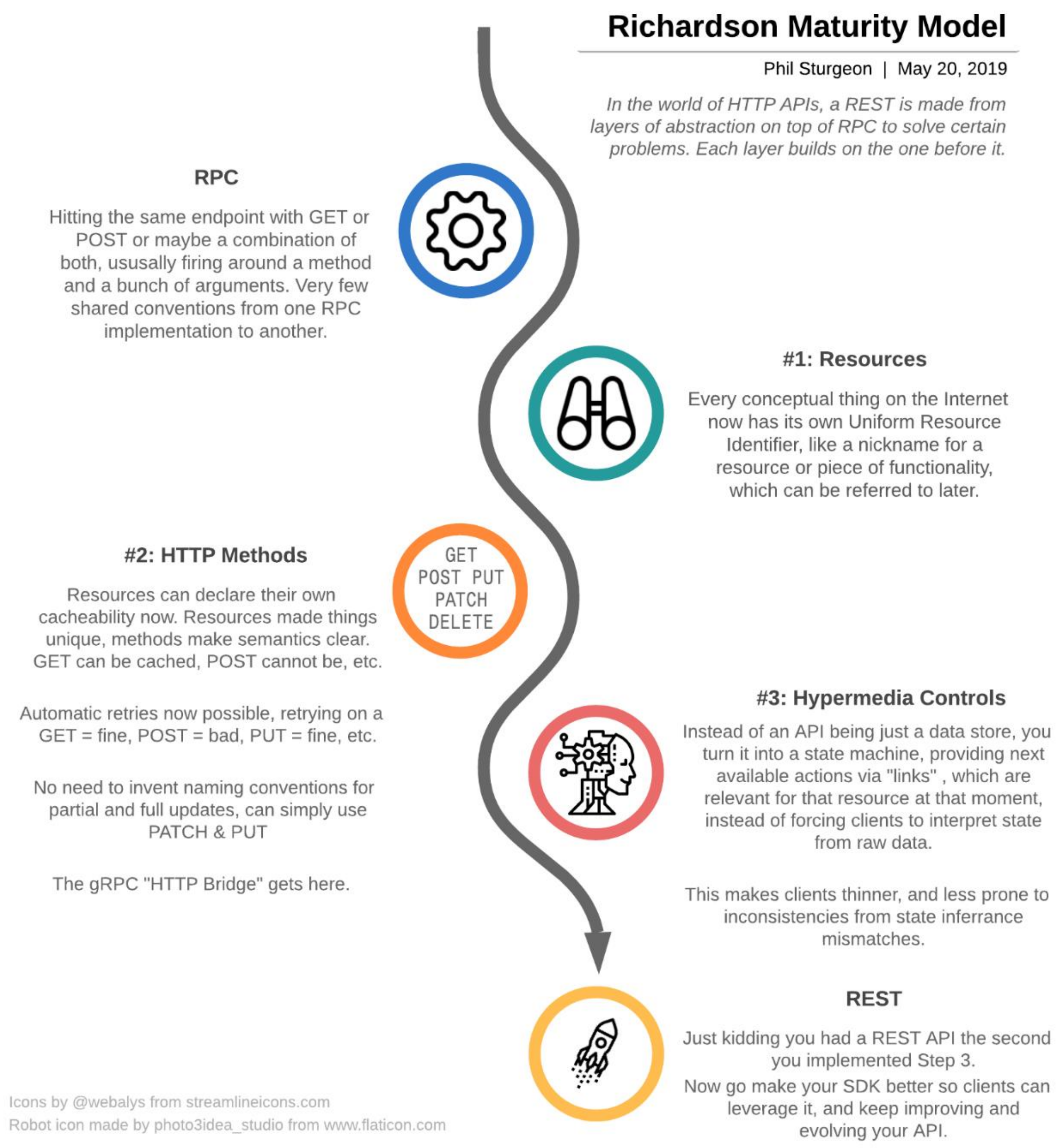
Resource Information

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 occured 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, ususally 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 inferrance mismatches.

REST

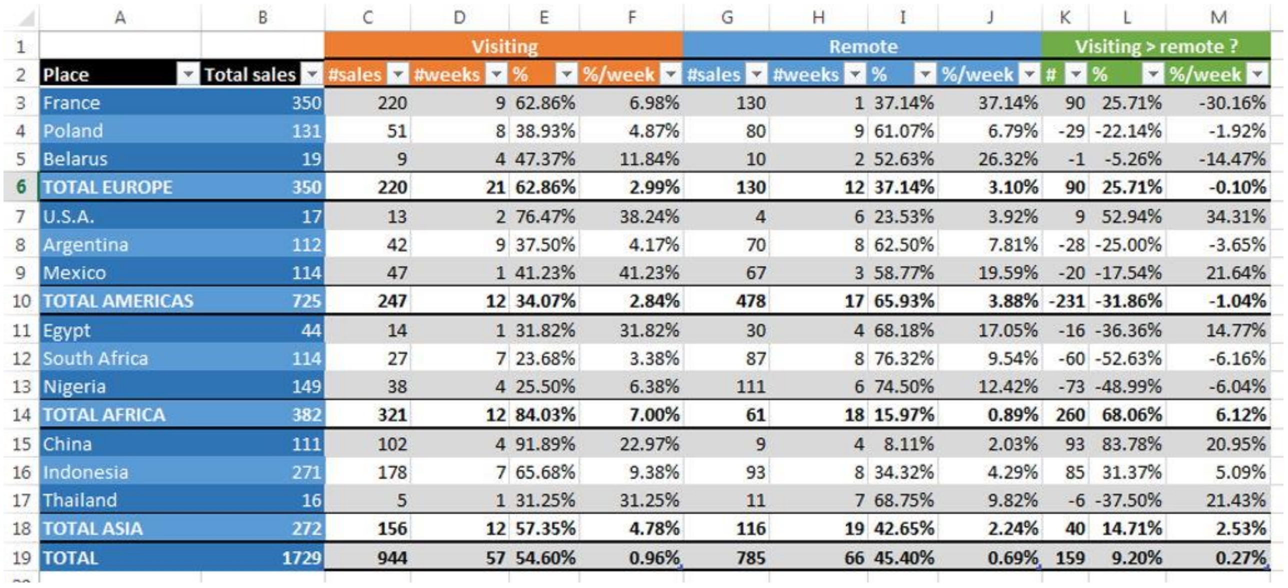
Just kidding you had a REST API the second you implemented Step 3.

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

Icons by @webalys from streamlineicons.com
Robot icon made by photo3idea_studio from www.flaticon.com

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

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	TOTAL EUROPE	350	220	21	62.86%	2.99%	130	12	37.14%	3.10%	90	25.71%	-0.10%
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	TOTAL AMERICAS	725	247	12	34.07%	2.84%	478	17	65.93%	3.88%	-231	-31.86%	-1.04%
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	TOTAL AFRICA	382	321	12	84.03%	7.00%	61	18	15.97%	0.89%	260	68.06%	6.12%
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	TOTAL ASIA	272	156	12	57.35%	4.78%	116	19	42.65%	2.24%	40	14.71%	2.53%
19	TOTAL	1729	944	57	54.60%	0.96%	785	66	45.40%	0.69%	159	9.20%	0.27%

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 hardcode 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

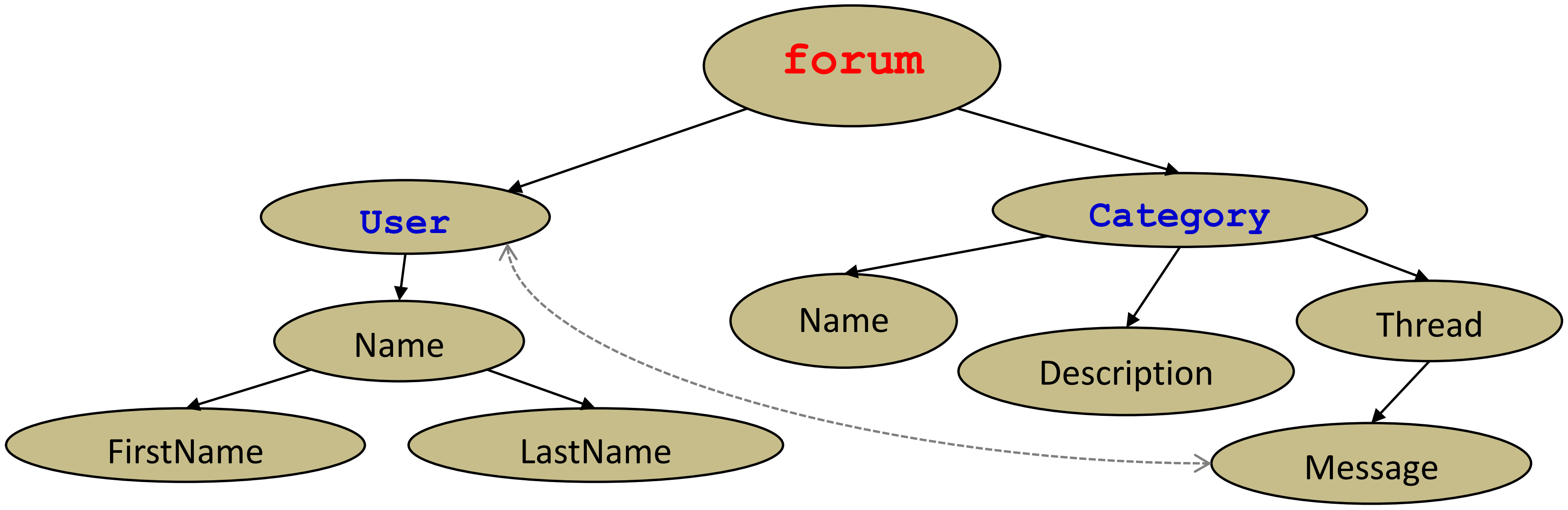
- REST APIs must be hypertext driven according to Fielding: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven
- HATEOAS
  - Hypermedia describes the actions that you can perform with the resources.
    - Client does not memorize operations nor workflow. Everything is in the messages
- 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?

Forum Resource hierarchy



Step 1 - Figure out the data set

- Define the concepts that you are going to expose in the Web API
- Describe the relations between them

Forum example

- Forum API permits <u>users</u> to publish new <u>messages</u>
- Users can post messages to different <u>categories</u>
- Users can reply to other users' messages
- Every user has a <u>public profile</u> and a <u>private profile</u>
  - Every user can check other users' public profiles
  - A private profile is shown only to that user's friends
- Users can check the last messages anyone has posted and commented
- Users can search messages in the forum using several criteria: keywords, user, popularity, date published, date commented, ...

Step 2 - Split the data into resources (I)

- RESTful Web services expose 3 kinds of resources:
  - <u>Predefined one-off resources</u> for important aspects of the application
    - They are usually repository for other resources.
      - Also known as **Collections.**
    - They cannot be deleted and their state cannot be modified directly
      - State only changes by modifying children resources

Forum example:
List of all users; list of all messages

  - <u>A resource for every object exposed through the service</u>
    - A service may expose many kinds of objects, each with its own resource set
    - Most services expose a large number of these resources

Forum example:
message categories (eg, Science category); particular users; particular messages

  - <u>Resources representing the results of algorithms applied to the data set</u>
    - Collections of resources, which are usually the results of queries

Forum example:
List of messages sent by a certain user; messages of a certain category

Step 2 - Split the data into resources (II)

- Resources are ordered in a hierarchical way
  - Hierarchy can be represent using a graph diagram
  - Consider carefully the hierarchy when resources which represent results of algorithms are involved; what is the result of the action?
- STEPS:
  - Define all possible types of resources the Web service is intended to expose
  - Give a name to each resource type

Forum example:
Some of the resource types are: message, user, category

  - Define the hierarchy
  - Define how those types of resources fit in the hierarchy
  - Take into account the platform you are going to use
    - Some platforms make it easier to create resources in certain way

Step 3 - Name the resources with URIs (I)

- Associate each resource type with a URI pattern
  - In a resource-oriented service the URI contains all the scoping info
- Design principles:
  1. <u>URIs should be descriptive</u>
     - The resource and its URI should be naturally and intuitively linked
  2. <u>Every URI designates exactly one resource</u>
     - Two resources can **NOT** share the same URI
     - Two different resources may point to the same data (but they are different resources!!!)
     - Forum Example:
       - At some moment the resources **`/forum/users/user_id/last_message`** and **`/forum/message/message_1`** could point to the same data (a forum message), but the resources are different!
  3. <u>The same resource can have one or many URIs</u>

Step 3 - Name the resources with URIs (II)

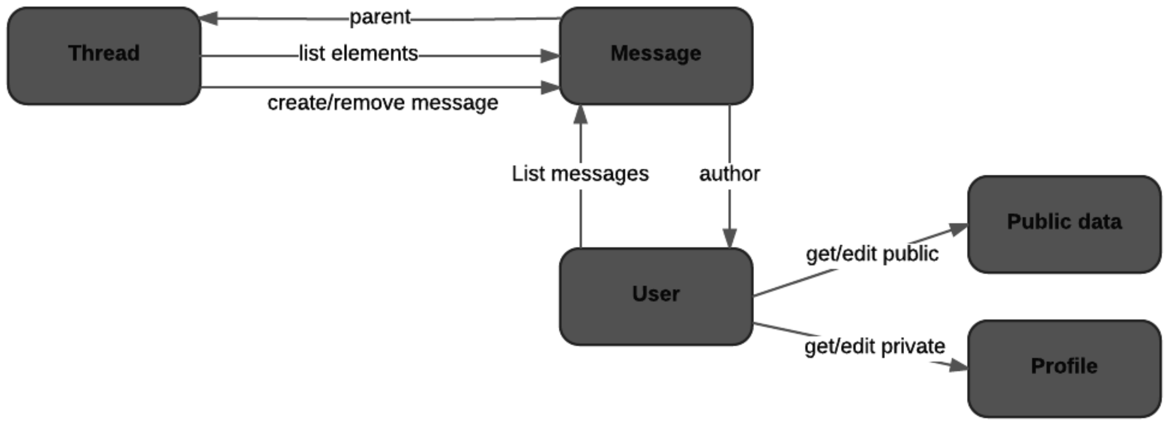
4. URIs should have a clear structure
   - Variation should be predictable – a client knowing the structure of the service's URI should be able of building URIs
   - Example:
     - **`http://forum.com/users/user_1/public_profile`**
   - Then, to get the public profile of user_2 the URI should be
     - Correct: **`/users/user_2/public_profile`**
     - Incorrect: **`/get_public_profile/user_2`**
   - Use the following convention:
     1) Use path variables to encode hierarchy: **`/parent/child`**
     2) Use punctuation characters in path variables if there is no hiearchical relation: **`/parent/child1;child2`**
        - » Use commas when the order of the scoping is important
        - » Use semicolon in other cases
     3) Use query variables to imply inputs for an algorithm

Forum example:

- ❑ **`http://forum.example.com/Users/user1`**
- ❑ **`http:://forum.example.com/messages/message1;message2`**
- ❑ **`http://forum.example.com/Categories/Science`**
- ❑ **`http://forum.example.com/Users/user1/history?last=5`**
  - ➢ Returns a list of the last 5 messages posted by user1

Step 4 – Establish the relation among resources

- State diagram of the application



- Will help later to design the hypermedia

Step 5 - Expose a subset of the uniform interface (I)

- Explain what happens to each resource when it is exposed to any of the methods of the uniform interface
  - Remember: A resource DOES NOT have to expose all the methods
  - If your resource is read-only, then expose two methods: GET and/or HEAD
  - If your resource can be created or modified you need to implement PUT, POST and/or DELETE
- Avoid creating your own methods (by *overloading* POST)
  - If you think you need an extra method,
    change the verb into a noun and create a resource
  - Example: If you think you need a method named **publish** just create a resource named **publication.** Use the uniform interface operations to modify it (e.g. POST a publication)

Step 5 - Expose a subset of the uniform interface (II)

- **Forum examples:**
  - Get all messages from the Sports category

```
GET  http://forum.example.com/Category/Sports
```

  - Create a new User

```
PUT  http://forum.example.com/Users/nicky
```

    - User information in the HTTP request body
  - Post a message into Science category

```
POST http://forum.example.com/Category/Science/Messages
```

    - Message content and details of the user are in the message body
  - Delete the message msg-4

```
DELETE http://forum.example.com/Category/Computers/Messages/msg-4
```

Steps 6. Design the resource representation using hypermedia formats

- Assign to each resource representation a format to transfer the resource state between client and server
  - The same resource can have different representation formats, but:
    - The server must understand all representations sent by the clients
    - The server must use a representation format the clients can understand
      - A client can ask for a specific format in the URI:
        - » Eg: **`http://forum.example.com/users/user_1.xml`**
      - A client can send HTTP headers indicating the formats it accepts:
        - » RFC2616 defines the following headers: Accept, Accept-Encoding…

- **NOTE: The resources representations sent from the client does not need to use hypermedia: JSON OR XML IS ENOUGH**

Media types (I)

- Domain specific standards
  - Defines application level and protocol level semantics
  - **OpenSearch, SVG, VoiceXML**
- Standard for specific patterns (e.g. collection pattern)
  - Defines protocol level but not application level standards
    - **Collection+JSON, Atom, Odata**
- Microformat and microdata
  - Defines protocol level but not application level
  - Microformat:
    - Extension of HTML4. Allows using the **class** attribute to define semantics
  - Microdata:
    - Extension of HTML5. Use itemprop, itemscope and itemtype attributes to define the semantics
  - Lots in schema.org

- General purpose media-types.
  - Allows personalizing the the protocol semantics and application level semantics
  - **HAL, HTML, SIREN**
- **Be careful with fake hypermedia: XML and JSON**

Media types (II)

- If you use your own media type be sure that hypermedia controls:
  - The URI of the remote resource
  - The relation of the current resource with the remote one
  - Try to include protocol information
    - E.g. which method I need to execute / what is the format of the request body
- If you are using XHTML:
  - Use **<a>** to have a link to another resource
  - Use **<form>** when you:
    - Include in the URI a query string
    - Represent URIs that follow a certain pattern

Domain specific media types. Creating links (I)

- Using xlink (http://www.w3.org/1999/xlink) attributes to create links:
  - Establish a relation between a local XML element and remote resources
    - **`xlink:type="simple"`** : Simple relation between current XML element and remote resource
    - **`xlink:href="`*`uri`*`"`** : Provides the path to the linked resource
    - Other voluntary attributes are:
      - **`xlink:role`** is a URI which indicates the relation between two resources
      - **`xlink:title`** is a human readable label which describes the link

```
<users xmlns:xlink="http://www.w3.org/1999/xlink>
   <user xlink:type="simple" xlink:href="http://forum/users/axel">
          <nickname>Axel</nickname>
   </user>
   <user xlink:type="simple" xlink:href="http://forum/users/bob">
          <nickname>Bob</nickname>
   </user>
</users>
```

  - More complicated relations can be established using other association types: extended, locator, arc, resource

Domain specific media types. Creating links (II)

- Using atom:link (http://www.w3.org/2005/Atom)
  - Element **`<atom:link>`** contains attributes to establish a relation between resources:
    - **`href`**: indicates the URI of the linked resource
    - **`rel`**: establish the semantic association between the resources. Different values:
      - **`self:`** the link points to the resource itself
      - More values on next slide
    - **`type`**: indicates the mime type of the representation

XML

```
<users xmlns:atom=”http://www.w3.org/2005/Atom”>
   <user>
          <atom:link rel=”self” href=”http://forum/users/axel”>
          <nickname>Axel</nickname>
   </user>
   <user>
          <atom:link rel=”self” href=”http://forum/users/bob”>
          <nickname>Bob</nickname>
   </user>
</users>
```

JSON

```
{users:[
          user:{nickname:”Axel”,link:{rel:”self”,href=” http://forum/users/axel”}},
          user:{nickname:”Bob”,link:{rel:”self”,href=” http://forum/users/bob”}}
]}
```

Domain specific media types. Creating links (III)

- Using atom:link (cont)
  - More values for **`rel`** attribute:
    - **`alternate:`** alternate representation of the same resource
    - **`edit:`** clients can edit the resource using this link
    - **`related:`** the linked resource has certain relation with the current reource
    - **`via`**: identifies the source for the information of current resource
    - **`enclosure`**: the link is a resource which contains current resource
    - **`previous, next:`** previous and next element in a list
    - **`first, last:`** first and last element of a list
    - Application developer can create application specific relations, expressed as URI
      - Very useful to manage application flow
        Drawback: it is application dependant

Domain specific media types. Creating links (IV)

- **URI templates** permit exposing an unlimited number of resources of the same type using just one URI
  - Parametrize URIs with variables that can be substituted at runtime
    - Variable names are shown between {}
  - Useful for the client to deliver parameters for an algorithm:
    - `http://forum/messages?older_than={timestamp}&maxReturned={max_returned}`
  - And to access a resource from a large set:
    - `http://forum/users/{user_id}`
    - In this case the client should have some knowledge on possible values
- URI templates are generated in the servers
  - They are parts of the links to other resources included in a resource representation; clients can fill the templates
- However, there are no conventions for representing URI templates
- Do not abuse URI templates
  - If you doubt then do not use URI templates
  - Use links when the set of results is known
- Use URI templates for:
  - Documentation
  - To identify resources in servers that accept URI template syntax

Profiles

- A profile must define:
  - **Link relations**:
    - Describing the state transition that will happen if the client triggers a hypermedia control (protocol semantics)
    - Usually implemented as ’rel’ attribute
      - http://www.iana.org/assignments/link-relations/link-relations.xhtml
      - Must be documented unless rel attribute is defined by IANA
    - Do not forget to include the method that is utilized
  - **Semantic descriptors**:
    - Describing the meaning of properties in the representation (application semantics)

IANA link relations

- Global register containing about 60 relations.
  - http://www.iana.org/assignments/link-relations/link-relations.xhtml
  - Some useful relations:
    - `collection` and `item` to create collections.
    - `first`, `last`, `next` and `previous` for pagination
    - `replies` to described message thread
    - `latest-version`, `successor-version`, `working-copy` for history of a resource state
    - `edit` and `edit-media` to cover update/delete a resource
- Some document media types defines its own possible relations
- Some profiles include also relations
- If you wanna use your own link relation
  - Use extension relations: *http://mydoma.in/myrelation*

- Microformats Wiki also contains a big set of relations:
  - http://microformats.org/wiki/existing-rel-values
  - DO NOT USE THEM AS SUCH IF YOU HAVE NOT DEFINED THEM IN YOUR PROFILE

Linking to a profile

- Using the **`profile`** Link relation:
  - RFC 6906 defines a `rel` called **`profile`**
  - Can be used in any `rel` attribute: `links` (Siren or Collection+Json); `link` defined in HTML, HAL or in the `Link` HTTP header.

```
<html>
  <head>
    <link href="http://microformats.org/wiki/hcard" rel="profile">
```

- Using the **`profile`** Media Type parameter:
  - Added as parameter in the Content-Type header

```
Content-Type = application/collection+json;profile=http://myprofile
```

- Using special purpose hypermedia controls defined in some media types.

Steps 6. Design the resource representation using hypermedia formats

Forum example. Message resource.

Media type: HAL

```
{

  "_links":{
    "self":{"href":"/forum/api/messages/msg-
2/" "profile":"http://atlassian.virtues.fi:8090/display/PWP/Exercise+3#Exercise3-Forum_Message"},
    "collection":{"href":"/forum/api/messages/", "type":"application/vnd.collection+json", "profile":"http://atlassi
an.virtues.fi:8090/display/PWP/Exercise+3#Exercise3-Forum_Message"},
    "author":{"href":"/forum/api/users/AxelW/", "type":"application/hal+json", "profile":"http://atlassian.virtues.f
i:8090/display/PWP/Exercise+3#Exercise3-Forum_User"},
    "in-reply-to":{"href":null "profile":"http://atlassian.virtues.fi:8090/display/PWP/Exercise+3#Exercise3-
Forum_Message"}
  }
  "template" : {
      "data" : [
        {"prompt" : "", "name" : "headline", "value" : "", "required":true},
        {"prompt" : "", "name" : "articleBody", "value" : "", "required":true},
        {"prompt" : "", "name" : "editor", "value" : "", required:false},
        {"prompt" : "", "name" : "author", "value" : "", required:false},
      ]
    }
  "articleBody":"I am using a float layout on my website but I've run into some problems with Internet Explorer. I
have set the left margin of a float to 100 pixels, but IE uses a margin of 200px instead. Why is that? Is this one
of the many bugs in IE?",
  "headline":"CSS: Margin problems with IE",
  "editor":null,
  "author":"AxelW"
}
```

Step 7. Define protocol specific attributes

- The resource representation is encapsulated in the HTTP request/response message
  - The HTTP body contains the representation
  - The HTTP entity headers contain metadata about the representation e.g. Its media type. Some important headers are:
    - **`Content-Type`**: mime-type of the representation format
      - A list of mime types can be found in RFC2045 and RFC2046
    - **`Content-Length`**: size of the body
    - **`Accept`**: formats a client understands (only in HTTP request)
    - **`Accept-Encoding`**: encoding accepted for the body
  - Other headers can be used for other purposes:
    - caching, authorization...

Step 7 - Define protocol specific attributes

- An HTTP response includes a **status code** indicating how the request was processed in the server
  - Headers provide additional information
- Response code + headers indicating success:
  - **GET**

200 OK	No headers	Successful request
304 Not Modified	No headers	The client must get the resource from the cache

  - **DELETE**

200 OK	No headers	Successful request. The HTTP body might contain a status message

  - **POST** and **PUT**

201 Created	Location	Successful creation. Location header indicates the URI of the resouce
200 OK	No headers	The resource existed and has been modified. The Body could contain the new resource
301 Moved permanently	Location	The data sent caused the resource URI to be changed

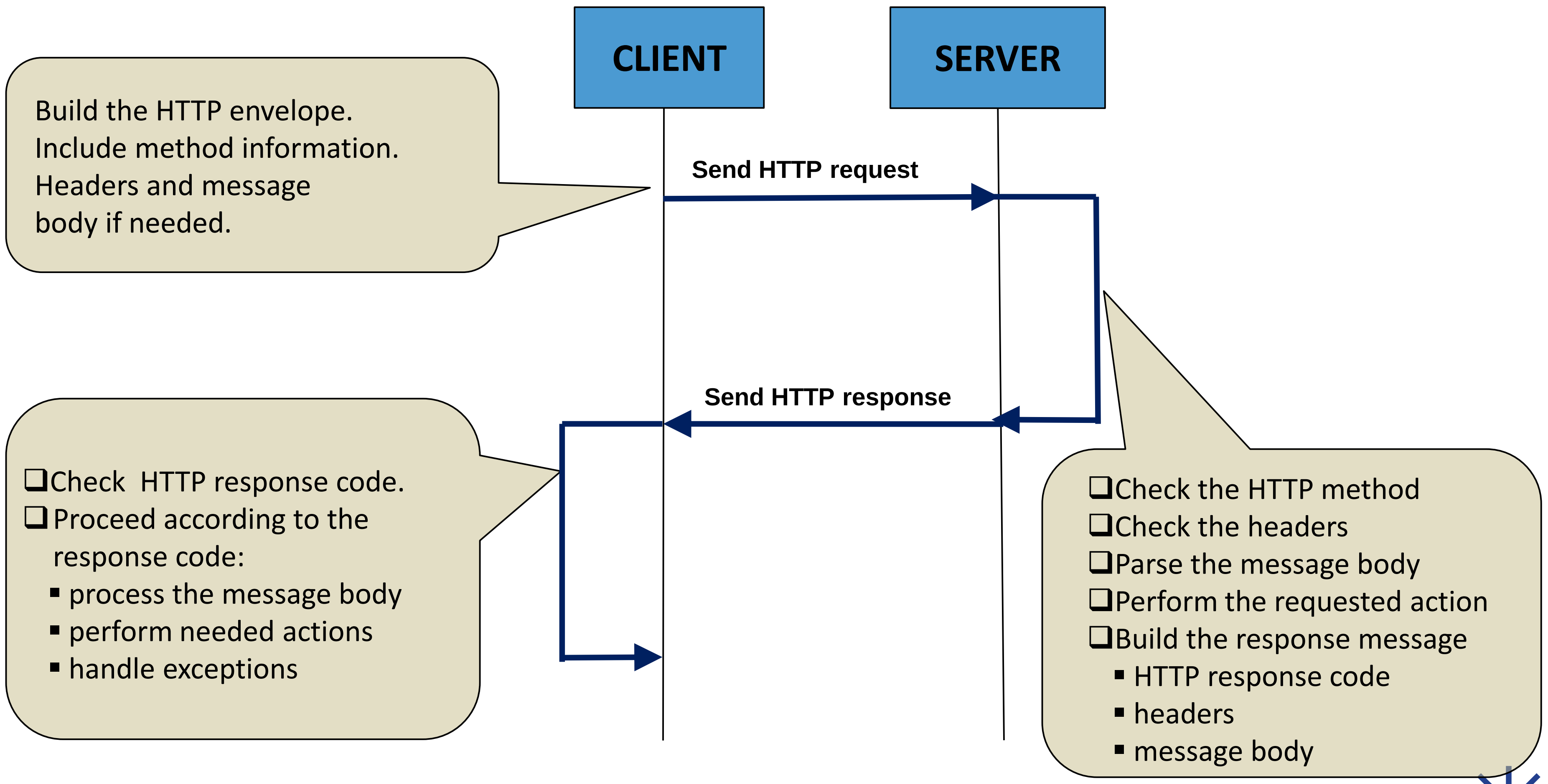
Step 8 - Define possible errors

- Define when and how a request could fail
  - Define the error message in the response body. It should be another resource
- Define also the response status codes and the headers of the response:
  - **GET** and **DELETE**

404 Not Found	No headers	Resource was not found. HTTP body message might have contained an error message.
303 See Other	Location	The resource was not found. Location header provides a related resource.
400 Bad Request	No header	The URI contained some erroneous fields or parameters

415 Unsopported Media Type	No headers	The representation format is not supported by the server
409 Conflict	No header	The representation tried to change the resource to a state that is not allowed
400 Bad Request	No header	The resource representation contained an invalid value

Basic workflow between client and web service



Build the HTTP envelope.
Include method information.
Headers and message
body if needed.

- ❑ Check HTTP response code.
- ❑ Proceed according to the response code:
  - process the message body
  - perform needed actions
  - handle exceptions

- ❑ Check the HTTP method
- ❑ Check the headers
- ❑ Parse the message body
- ❑ Perform the requested action
- ❑ Build the response message
  - HTTP response code
  - headers
  - message body

Forum example - GET

- Get all messages from the Sports category
  - **HTTP Method:** `GET`
  - **URI:** `http://forum.example.com/Category/Sports`
  - **Returns:**
    - On success: `200 OK` + XML message body
    - On error: `401 Unauthorized` or `404 Not found`

Request HTTP envelope

```
GET Category/Sports/ HTTP/1.1
Host: forum.example.com
Accept: text/xml
Accept-Encoding: gzip,deflate
Accept-Charset: windows-
1251,utf-8;q=0.7,*;q=0.7
```

Successful HTTP response envelope

```
HTTP/1.1 200 OK
Date: Sun, 12 Sep 2010 11:30:12 GMT
Transfer-Encoding: chunked
Content-Type: text/xml;
Content-Length: length;
<?xml version="1.0" encoding="UTF-8"?>
<msg:Thread>
  <msg:Message messageID="msg-3">
    <msg:Registered userID="user-7">
      <user:Nickname>HockeyFan</user:Nickname>
      <user:Avatar file="avatar_7.jpg"/>
    </msg:Registered>
    <msg:Title>Edmonton's goalie</msg:Title>
    <msg:Body>Does anyone know where Jussi...
   (...)
  </msg:Message>
(…)
<msg:Thread>
```

Forum example - POST

- Post the message into Science category
  - **HTTP Method:** POST
  - **URI:** http://forum.example.com/Category/Science/Messages
  - **Request:** XML message body
  - **Returns:**
    - On success: 201 Created (Location header tells the URI of created message)
    - On error: 400 Bad Request or 409 Conflict

Request HTTP envelope

```
POST Category/Science/Messages HTTP/1.1
Host: forum.example.com
Accept: text/xml
Accept-Encoding: gzip,deflate
Accept-Charset: windows-1251,utf-
8;q=0.7,*;q=0.7
Content-Type: text/xml;charset=utf-8
Content-Length: length
<?xml version="1.0" encoding="UTF-8"?>
<msg:Message messageID=“" replyTo="msg-1">
  <msg:Anonymous>Science guru</msg:Anonymous>
          (...)
            </msg:Message>
```

Successful HTTP response envelope

```
HTTP/1.1 201 Created
Date: Tue, 19 Sep 2010 06:11:22 GMT
Content-Type: text/xml; charset=iso-8859-1
Content-Length: length
Location:
http://forum.example.com/Category/Science/Messages/msg-4
<?xml version="1.0" encoding="UTF-8"?>
<msg:Message messageID="msg-4" replyTo="msg-1">
  <msg:Anonymous>Science guru</msg:Anonymous>
  <msg:Title>In case</msg:Title>
  <msg:Body>Just in case you can't ...
(...)
</msg:Message>
```

Forum example - DELETE

- Delete certain message
  - **HTTP Method:** DELETE
  - **URL:** http://forum.example.com/Category/Science/Messages/msg-4
  - **Returns:**
    - On success: 204 No Content
    - On error: 401 Unauthorized or 404 Not Found

Request HTTP envelope

```
DELETE
Category/Science/Messages/msg-4
HTTP/1.1
Host: forum.example.com
Accept: text/xml, text/html
Accept-Encoding: gzip,deflate
Accept-Charset: windows-1251,utf-
8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

Error HTTP response

```
HTTP/1.1 404 Not Found
Date: Tue, 19 Sep 2010 06:11:22 GMT
Content-Type: text/html; charset=iso-8859-1
Content-Length: length
Keep-Alive: timeout=15, max=96
Connection: Keep-Alive
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
 <head>
  <title>404 Not Found</title>
 </head>
 <body>
  <h1>Not Found</h1>
  <p>The requested message msg-4 was not found on
this server.</p>
 </body>
</html>
```

HYPERMEDIA DRIVEN DESIGN

Resource driven vs Hypermedia driven

- **Resource driven design**
  - MOST utilized approach nowadays when people talk about REST
  - Nouns is the most important

- **Hypermedia driven design**
  - ACTION is the most important
  - Acknowledges that the state transitions are even more important than the state itself.
    - I want to do a thing.
    - Which verbs should I use to do that?
  - Previous state transitions will provide 'affordances' that indicates what actions I can perform next and a way of figuring out more information about those affordances if we do not know it already.

Advantages and disadvantages

PROS

- promote scalability
- allow resilience towards future changes
  - Clients and serves evolve separately
- promote decoupling and encapsulation
  - All request are self-contained
  - Facilitates evolvability
- Code on demand promotes extensibility

CONS

- NOT latency-tolerant design
- caches can get stale
- Not as efficient on an individual request level as other designs
- More verbose request / responses
- Usually, more complex clients

OULUN YLIOPISTO

Design process (I)

1. Evaluate processes
2. Create state machine
3. Evaluate media types
4. Create or choose media types
5. Implementation!
6. Refinements

Design process (II)

- Documenting a REST API => defining the media types.

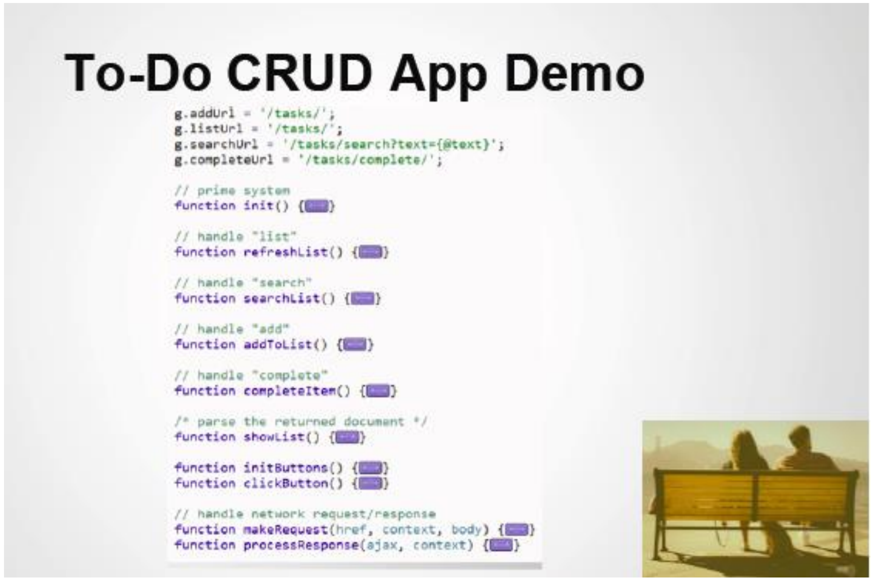
"A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types. Any effort spent describing what methods to use on what URIs of interest should be entirely defined within the scope of the processing rules for a media type (and, in most cases, already defined by existing media types)"

Roy Fielding. REST APIs must be hypertext-driven

- The media type is the only sort of contract between the client and the server

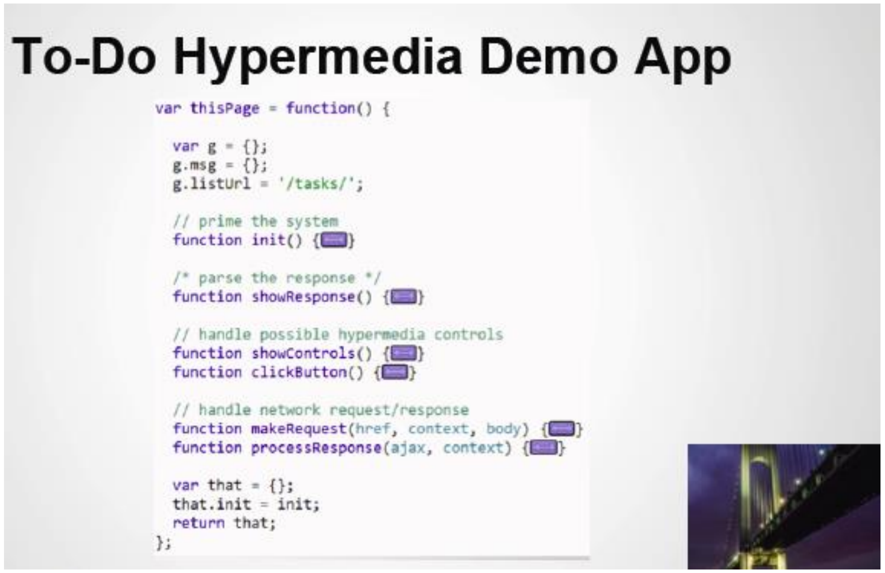
Hypermedia driven APIs. Examples.

Simpler clients. No memorize workflow, objects or URL. Just implement how to process hypermedia controls.



To-Do CRUD App Demo

```
g.addUrl = '/tasks/';
g.listUrl = '/tasks/';
g.searchUrl = '/tasks/search?text={@text}';
g.completeUrl = '/tasks/complete/';

// prime system
function init() {…}

// handle "list"
function refreshList() {…}

// handle "search"
function searchList() {…}

// handle "add"
function addToList() {…}

// handle "complete"
function completeItem() {…}

/* parse the returned document */
function showList() {…}

function initButtons() {…}
function clickButton() {…}

// handle network request/response
function makeRequest(href, context, body) {…}
function processResponse(ajax, context) {…}
```



To-Do Hypermedia Demo App

```
var thisPage = function() {

  var g = {};
  g.msg = {};
  g.listUrl = '/tasks/';

  // prime the system
  function init() {…}

  /* parse the response */
  function showResponse() {…}

  // handle possible hypermedia controls
  function showControls() {…}
  function clickButton() {…}

  // handle network request/response
  function makeRequest(href, context, body) {…}
  function processResponse(ajax, context) {…}

  var that = {};
  that.init = init;
  return that;
};
```

Mike Amundsen. REST, Hypermedia, and the Semantic Gap: Why "RMM Level-3 REST" is not enough.

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

References

1. “RESTful Web Services” by Leonard Richardson and Sam Ruby
2. “RESTful Web APIs” by Leonard Richardson, Mike Amundsen and Sam Ruby
3. “RESTful Web Services Cookbook” by Subbu Allamaraju
4. “REST in practice. Hypermedia and Systems Architecture” by Jim Webber, Savas Parastidis and Ian Robinson.
5. Representational State Transfer (REST), Roy Thomas Fielding. Available at http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf
6. “Peer-to-Peer Systems and Applications” Ralf Steinmetz KlausWehrle (Eds.)

   Available at http://www.springerlink.com/content/g6h805426g7t/#section=586017&page=1
7. ATOM http://www.ietf.org/rfc/rfc4287.txt
8. HTTP 1.1 http://tools.ietf.org/html/rfc2616
9. JSON http://www.json.org/

OULUN YLIOPISTO