Fork me on GitHub

RESTful Web Services

Mobile Web Services

Olivier Liechti & Yannick Iseli

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

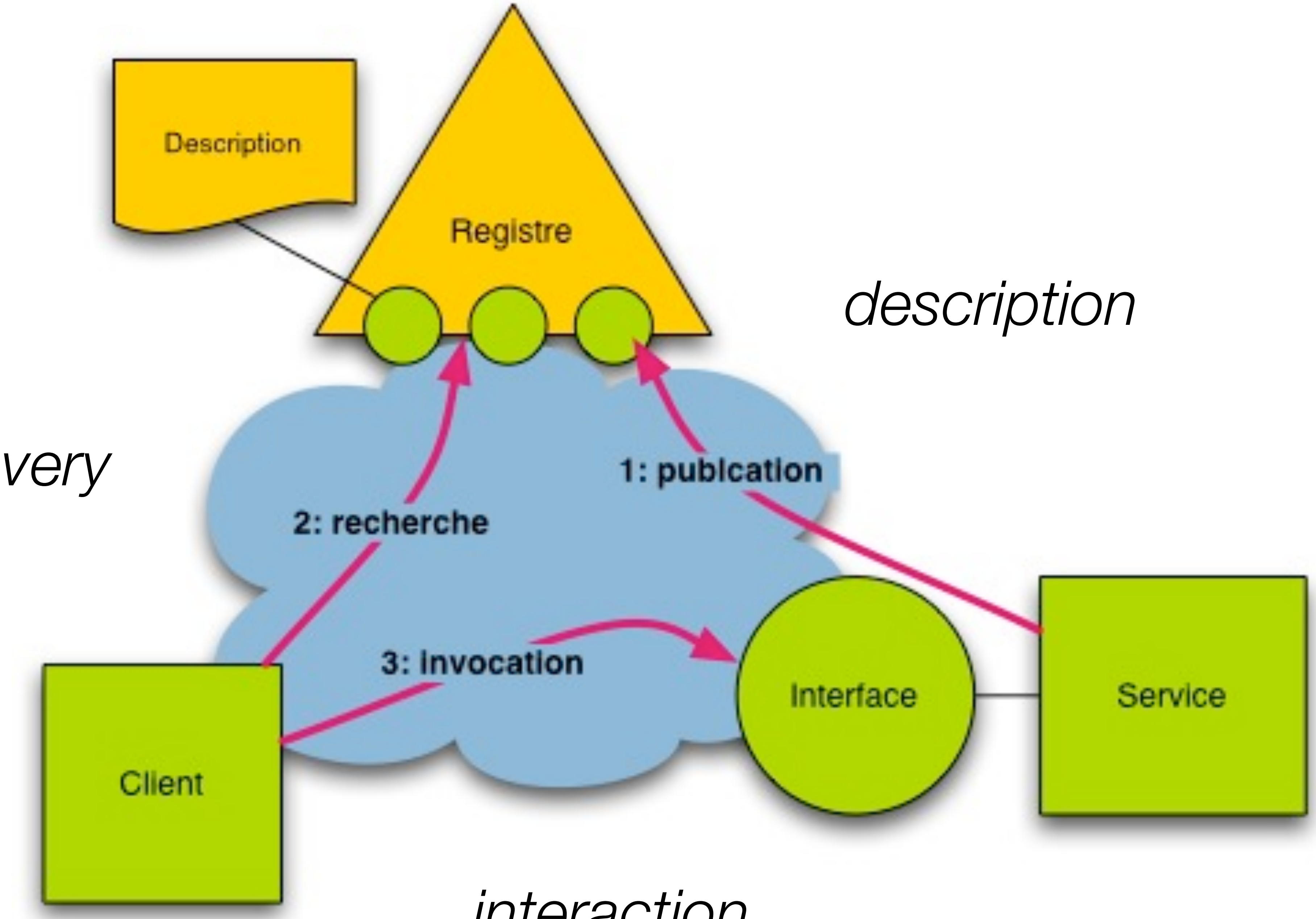
heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Two Approaches to Web Services

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

The Big Web Services Approach

The Web Services Reference Architecture



description

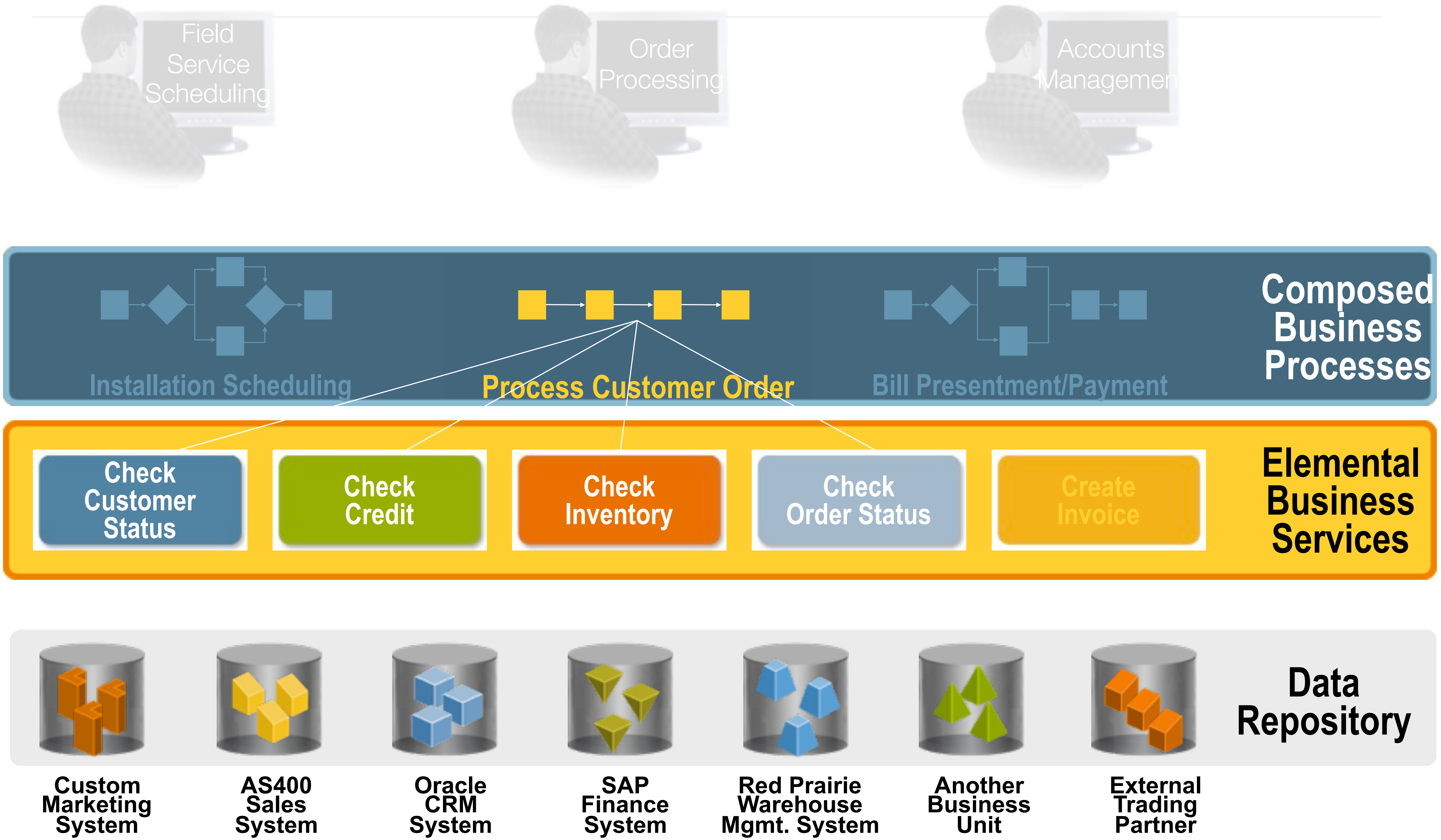
discovery

interaction

For a **Full** Service Architecture, We Need...

- A standardized format to **describe** service interfaces
  - Example: WSDL
- A standardized protocol to **invoke** services
  - Example: SOAP
- A **registry** service
  - Example: UDDI

SOA: Service Composition & Workflows



Field Service Scheduling

Order Processing

Accounts Managemen

Installation Scheduling

Process Customer Order

Bill Presentment/Payment

Composed Business Processes

Check Customer Status

Check Credit

Check Inventory

Check Order Status

Create Invoice

Elemental Business Services

Data Repository

Custom Marketing System

AS400 Sales System

Oracle CRM System

SAP Finance System

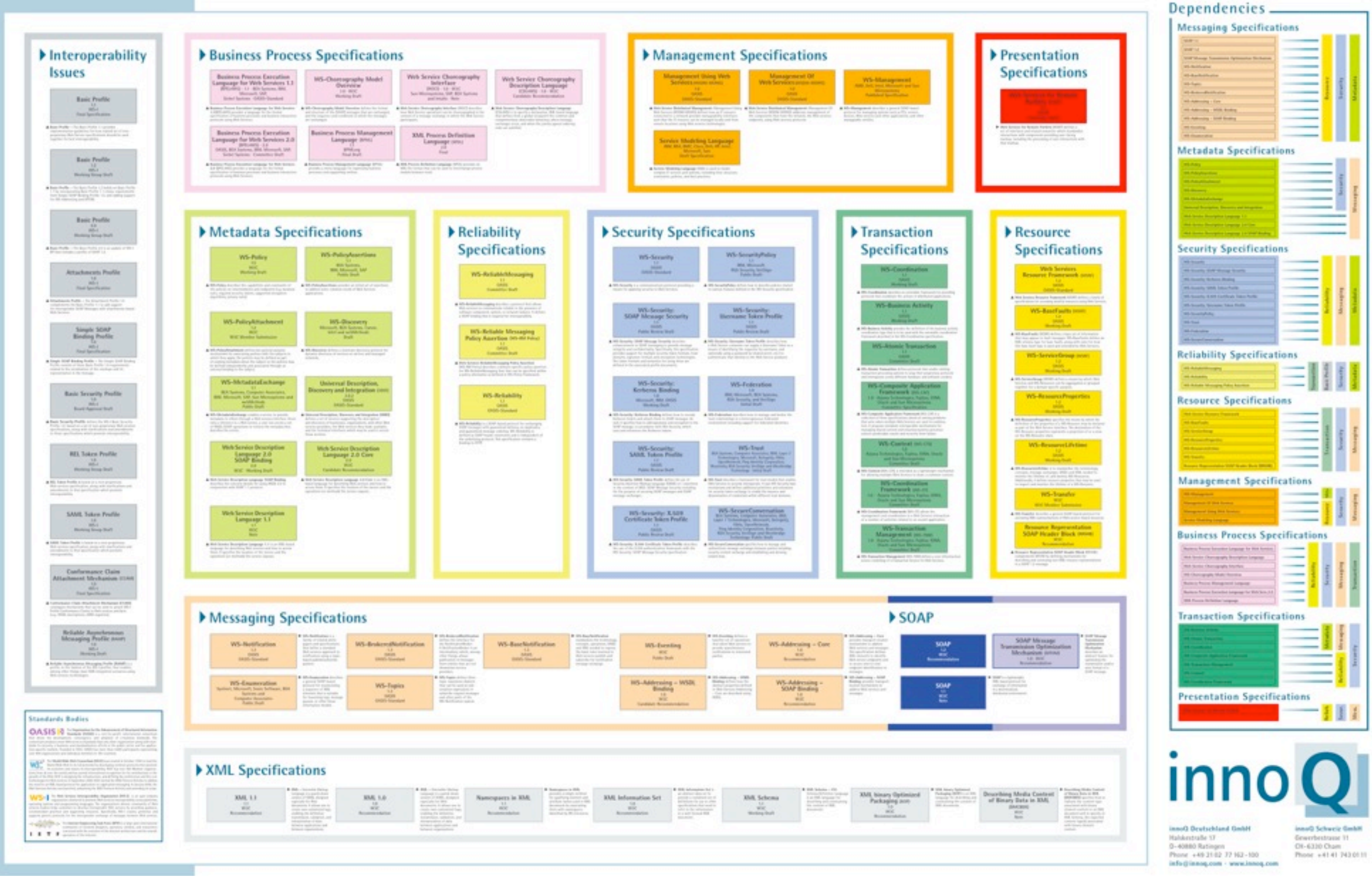
Red Prairie Warehouse Mgmt. System

Another Business Unit

External Trading Partner

© Sun Microsystems

Web Services Standards Overview


Interoperability Issues
Basic Profile
Attachments Profile
Simple SOAP Binding Profile
Basic Security Profile
REL Token Profile
SAML Token Profile
Conformance Claim Attachment Mechanism
Reliable Asynchronous Messaging Profile
Standards Bodies
OASIS
W3C
WS-I
IETF
Business Process Specifications
Business Process Execution Language for Web Services 1.1
WS-Choreography Model Overview
Web Service Choreography Interface
Web Service Choreography Description Language
Business Process Execution Language for Web Services 2.0
Business Process Management Language
XML Process Definition Language
Management Specifications
Management Using Web Services
Management Of Web Services
WS-Management
Service Modeling Language
Presentation Specifications
Metadata Specifications
WS-Policy
WS-PolicyAssertions
WS-PolicyAttachment
WS-Discovery
WS-MetadataExchange
Universal Description, Discovery and Integration
Web Service Description Language 2.0 SOAP Binding
Web Service Description Language 2.0 Core
Web Service Description Language 1.1
Reliability Specifications
WS-ReliableMessaging
WS-Reliable Messaging Policy Assertion
WS-Reliability
Security Specifications
WS-Security
WS-SecurityPolicy
WS-Security: SOAP Message Security
WS-Security: Username Token Profile
WS-Security: Kerberos Binding
WS-Federation
WS-Security: SAML Token Profile
WS-Trust
WS-Security: X.509 Certificate Token Profile
WS-SecureConversation
Transaction Specifications
WS-Coordination
WS-Business Activity
WS-Atomic Transaction
WS-Composite Application Framework
WS-Context
WS-Coordination Framework
WS-Transaction Management
Resource Specifications
Web Services Resource Framework
WS-BaseFaults
WS-ServiceGroup
WS-ResourceProperties
WS-ResourceLifetime
WS-Transfer
Resource Representation SOAP Header Block
Messaging Specifications
WS-Notification
WS-BrokeredNotification
WS-BaseNotification
WS-Eventing
WS-Addressing – Core
WS-Enumeration
WS-Topics
WS-Addressing – WSDL Binding
WS-Addressing – SOAP Binding
SOAP
SOAP Message Transmission Optimization Mechanism
XML Specifications
XML 1.1
XML 1.0
Namespaces in XML
XML Information Set
XML Schema
XML-binary Optimized Packaging
Describing Media Content of Binary Data in XML
Dependencies
Messaging Specifications
Metadata Specifications
Security Specifications
Reliability Specifications
Resource Specifications
Management Specifications
Business Process Specifications
Transaction Specifications
Presentation Specifications
innoQ

http://www.innoq.com/soa/ws-standards/poster/
http://www.innoq.com/soa/ws-standards/

Big Web Services with Java EE

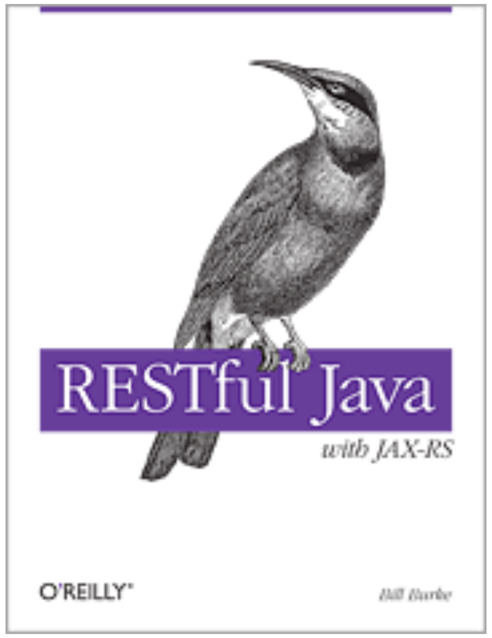
- **JAX-WS**
  - JAX-WS makes it easier to write both **web services** and **web services clients**.
  - The JAX-WS **runtime** takes care of the SOAP and WSDL details and provides you with an object-oriented interface.
- Exposing your **Stateless Session Beans** with a Web Services interface
  - Adding a single annotation will do the job.
  - JAX-WS relies on conventions for generating the WSDL interface; you can customize the schema with various annotations.

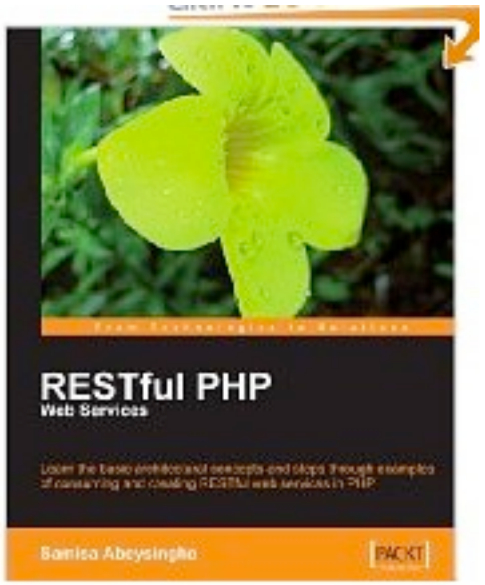
The RESTful Approach

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

RESTful Web Services

Web Services for the Real World
RESTful
Web Services
O'REILLY
Leonard Richardson & Sam Ruby
Foreword by David Heinemeier Hansson





The REST Architectural Style

- REST: REpresentational State Transfer
- REST is an architectural style for building distributed systems.
- REST has been introduced in Roy Fielding's Ph.D. thesis (Roy Fielding has been a contributor to the HTTP specification, to the apache server, to the apache community).
- The WWW is one example for a distributed system that exhibits the characteristics of a REST architecture.

HTTP is a protocol for interacting with "**resources**"

What is a “Resource”

- At first glance, one could think that a “resource” is a file on a web server:
  - an HTML document, an XML document, a PNG document
- That fits the vision of the “static content” web
- But of course, the web is now more than a huge library of hypermedia documents:
  - through the web, we interact with services and a lot of the content is dynamic.
  - more and more, through the web we interact with physical objects (machines, sensors, actuators)
  - We need a more generic definition for resources!

What is a “Resource”?

- A resource is "something" that can be named and uniquely identified:
  - Example 1: an article published in the "24 heures" newspaper
  - Example 2: the collection of articles published in the sport section of the newspaper
  - Example 3: a person’s resume
  - Example 4: the current price of the Nestlé stock quote
  - Example 5: the vending machine in the school hallway
  - Example 6: the list of grades of the student Jean Dupont
- URL (Uniform Resource Locator) is a mechanism for identifying resources
  - Exemple 1: http://www.24heures.ch/vaud/vaud/2008/08/04/trente-etudiants-partent-rencontre-patrons
  - Exemple 2: http://www.24heures.ch/articles/sport
  - Exemple 5: http://www.smart-machines.ch/customers/heig/machines/8272

Resource vs. Representation

- A "resource" can be something intangible (stock quote) or tangible (vending machine)
- The HTTP protocol supports the exchange of data between a client and a server.
- Hence, what is exchanged between a client and a server is **not** the resource. It is a **representation** of a resource.
- Different representations of the same resource can be generated:
  - HTML representation
  - XML representation
  - PNG representation
  - WAV representation
- **HTTP provides the content negotiation mechanisms!!**

How Do We Interact With Resources?

- The HTTP protocol defines the standard methods. These methods enable the interactions with the resources:
    - **GET**: retrieve whatever information is identified by the Request-URI
    - **POST**: used to request that the origin server accept the entity enclosed in the request as a new subordinate of the ressource identified by the Request-URI in the Request-Line
    - **PUT**: requests that the enclosed entity be stored under the supplied Request-URI.
    - **DELETE**: requests that the origin server delete the ressource identified by the Request-URI.
    - HEAD: identical to GET except that the server MUST NOT return a message-body in the response
    - TRACE: used for debugging (echo)
    - CONNECT: reserved for tunneling purposes

Principles of a REST Architecture

- The state of the application is captured in a **set of resources**
  - Users, photos, comments, tags, albums, etc.
- Every resource can be **identified with a standard format** (e.g. URL)
- Every resource can have **several representations**
- There is one **unique interface for interacting** with resources (e.g. HTTP methods)
- The communication protocol is:
  - client-server
  - stateless
  - cacheable
- These properties have a positive impact on systemic qualities (scalability, performance, availability, etc.).
  - Reference: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Reference

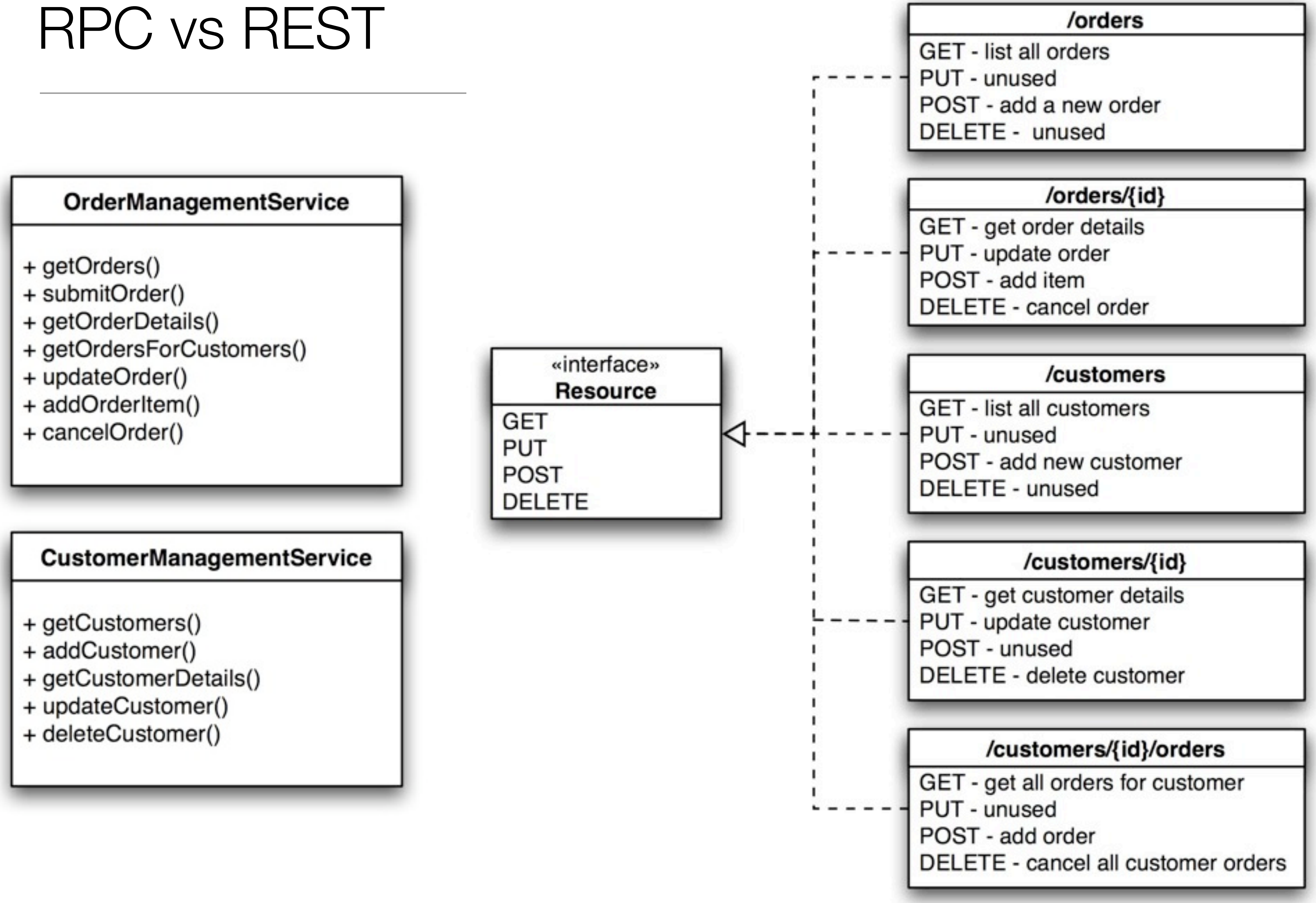
- Very good article, with presentation of key concepts and illustrative examples:
  - http://www.infoq.com/articles/rest-introduction

How should I specify/document my REST API?

Design a RESTful system

- Start by identifying the resources - the NAMES in your system.
- Define the structure of the URLs that will be mapped to your resources.
- Define the semantic of the operations that you want to support on all of your resources (you don't want to support GET, POST, PUT, DELETE on all resources!).
- Some examples:
  - http://www.photos.com/users/oliechti identifies a resource of type "user". A client can do a "HTTP GET" to obtain a representation of the user or a "HTTP PUT" to update the user.
  - http://www.photos.com/users identifies a resource of type "collection of users". A client can do a "HTTP POST" to add users, or an "HTTP GET" to obtain the list of users.

RPC vs REST



OrderManagementService

+ getOrders()
+ submitOrder()
+ getOrderDetails()
+ getOrdersForCustomers()
+ updateOrder()
+ addOrderItem()
+ cancelOrder()

CustomerManagementService

+ getCustomers()
+ addCustomer()
+ getCustomerDetails()
+ updateCustomer()
+ deleteCustomer()

«interface»
Resource

GET
PUT
POST
DELETE

/orders

GET - list all orders
PUT - unused
POST - add a new order
DELETE - unused

/orders/{id}

GET - get order details
PUT - update order
POST - add item
DELETE - cancel order

/customers

GET - list all customers
PUT - unused
POST - add new customer
DELETE - unused

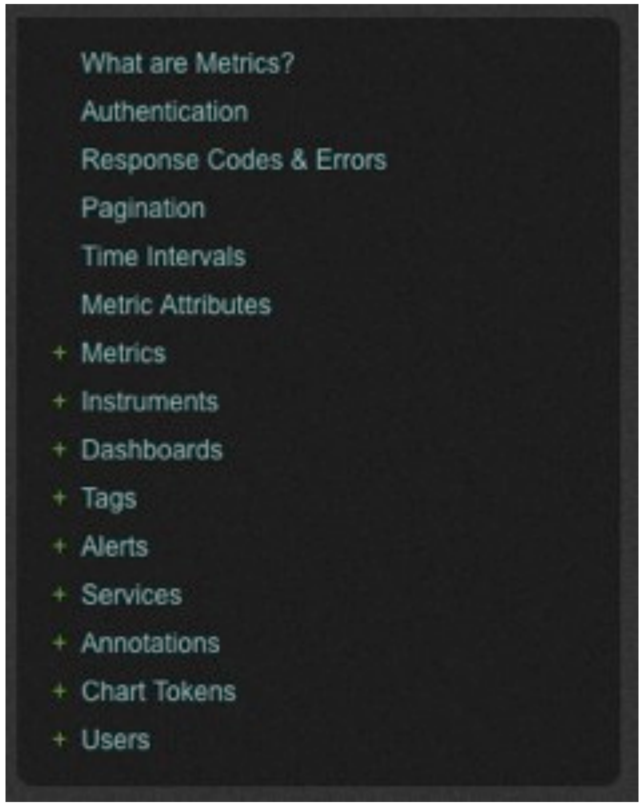
/customers/{id}

GET - get customer details
PUT - update customer
POST - unused
DELETE - delete customer

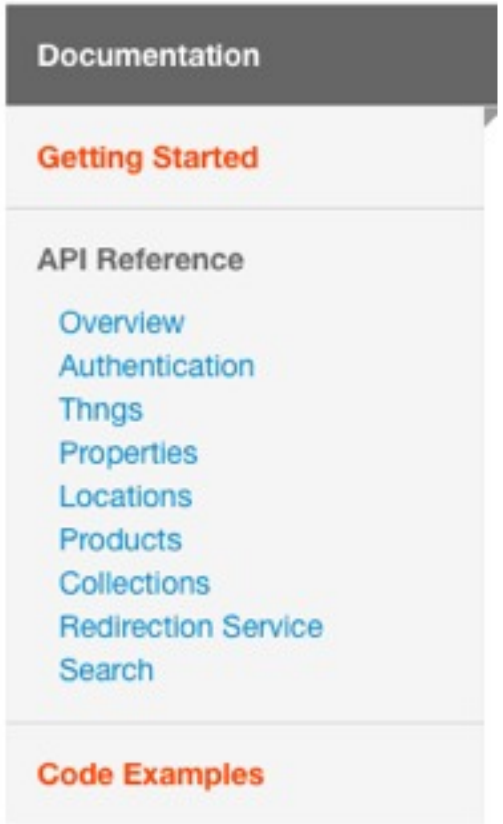
/customers/{id}/orders

GET - get all orders for customer
PUT - unused
POST - add order
DELETE - cancel all customer orders

Look at Some Examples



What are Metrics?
Authentication
Response Codes & Errors
Pagination
Time Intervals
Metric Attributes
+ Metrics
+ Instruments
+ Dashboards
+ Tags
+ Alerts
+ Services
+ Annotations
+ Chart Tokens
+ Users



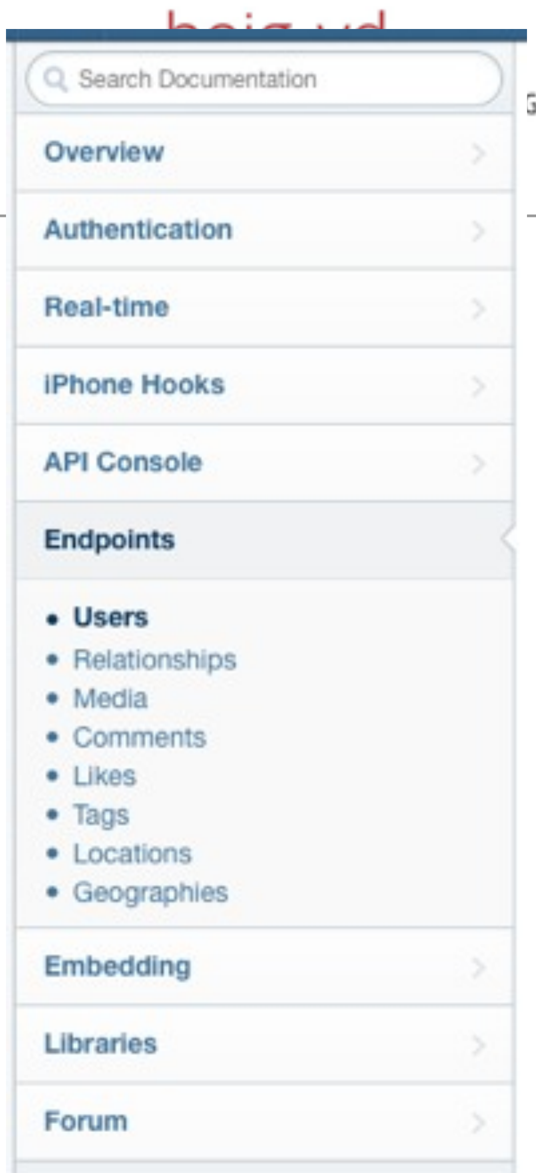
Documentation

Getting Started

API Reference

Overview
Authentication
Thngs
Properties
Locations
Products
Collections
Redirection Service
Search

Code Examples



Search Documentation

Overview
Authentication
Real-time
iPhone Hooks
API Console
Endpoints

- Users
- Relationships
- Media
- Comments
- Likes
- Tags
- Locations
- Geographies

Embedding
Libraries
Forum

http://dev.librato.com/v1

https://dev.evrythng.com/documentation/api

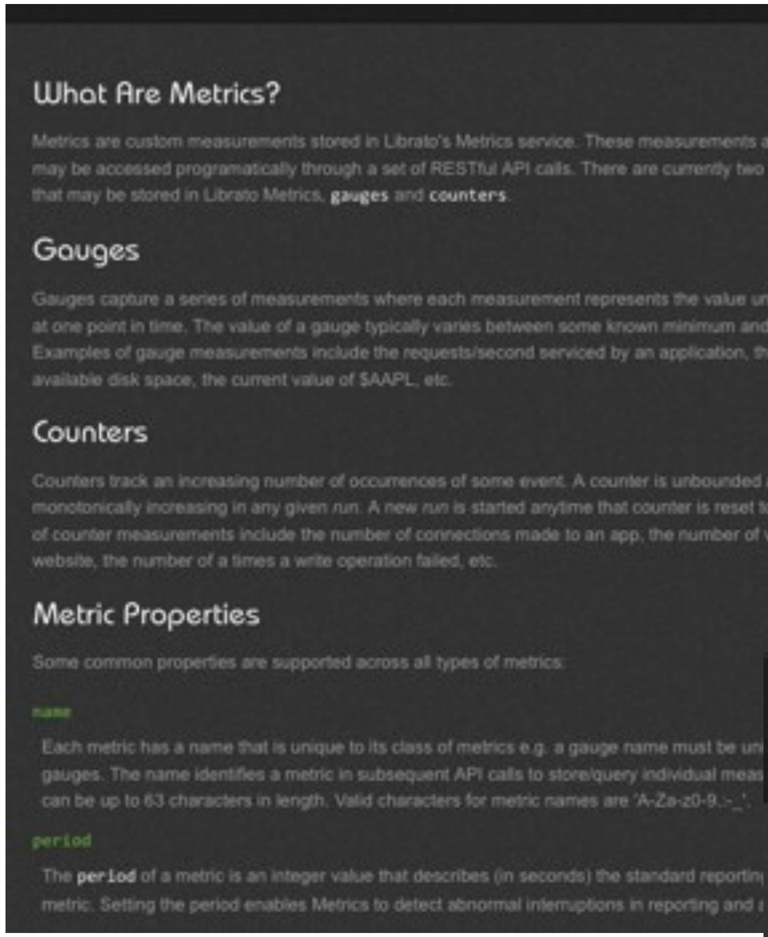
http://instagram.com/developer/endpoints/

librato

EVRYTHNG™

Instagram

Short description of the resource (domain model)



What Are Metrics?

Metrics are custom measurements stored in Librato's Metrics service. These measurements are created and may be accessed programatically through a set of RESTful API calls. There are currently two types of metrics that may be stored in Librato Metrics, `gauges` and `counters`.

Gauges

Gauges capture a series of measurements where each measurement represents the value under observation at one point in time. The value of a gauge typically varies between some known minimum and maximum. Examples of gauge measurements include the requests/second serviced by an application, the amount of available disk space, the current value of $AAPL, etc.

Counters

Counters track an increasing number of occurrences of some event. A counter is unbounded and always monotonically increasing in any given *run*. A new *run* is started anytime that counter is reset to zero. Examples of counter measurements include the number of connections made to an app, the number of visitors to a website, the number of a times a write operation failed, etc.

Metric Properties

Some common properties are supported across all types of metrics:

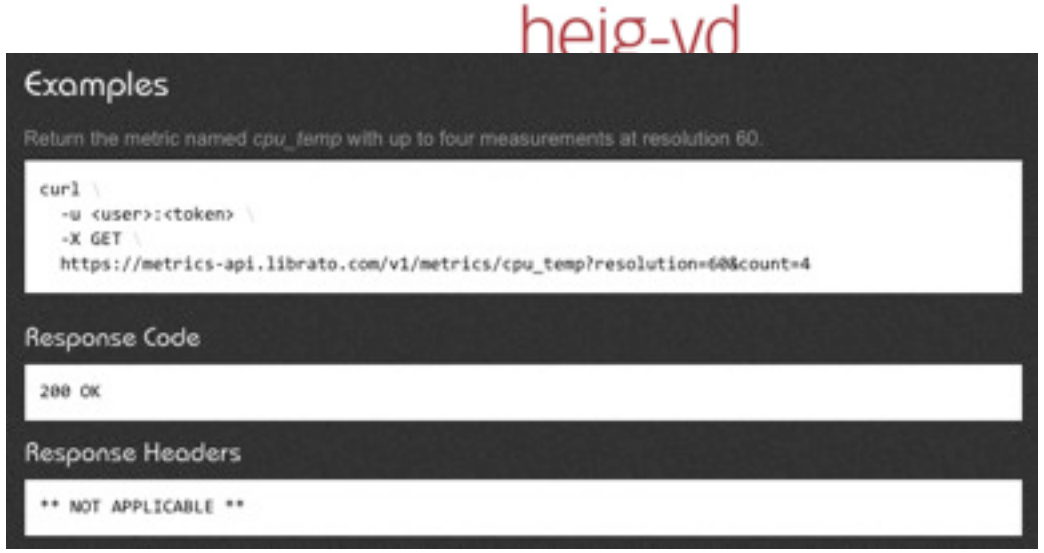
`name`

Each metric has a name that is unique to its class of metrics e.g. a gauge name must be un
gauges. The name identifies a metric in subsequent API calls to store/query individual meas
can be up to 63 characters in length. Valid characters for metric names are 'A-Za-z0-9.:-_'.

`period`

The **`period`** of a metric is an integer value that describes (in seconds) the standard reportin
metric. Setting the period enables Metrics to detect abnormal interruptions in reporting and a

Examples & payload structure



Examples

Return the metric named cpu_temp with up to four measurements at resolution 60.

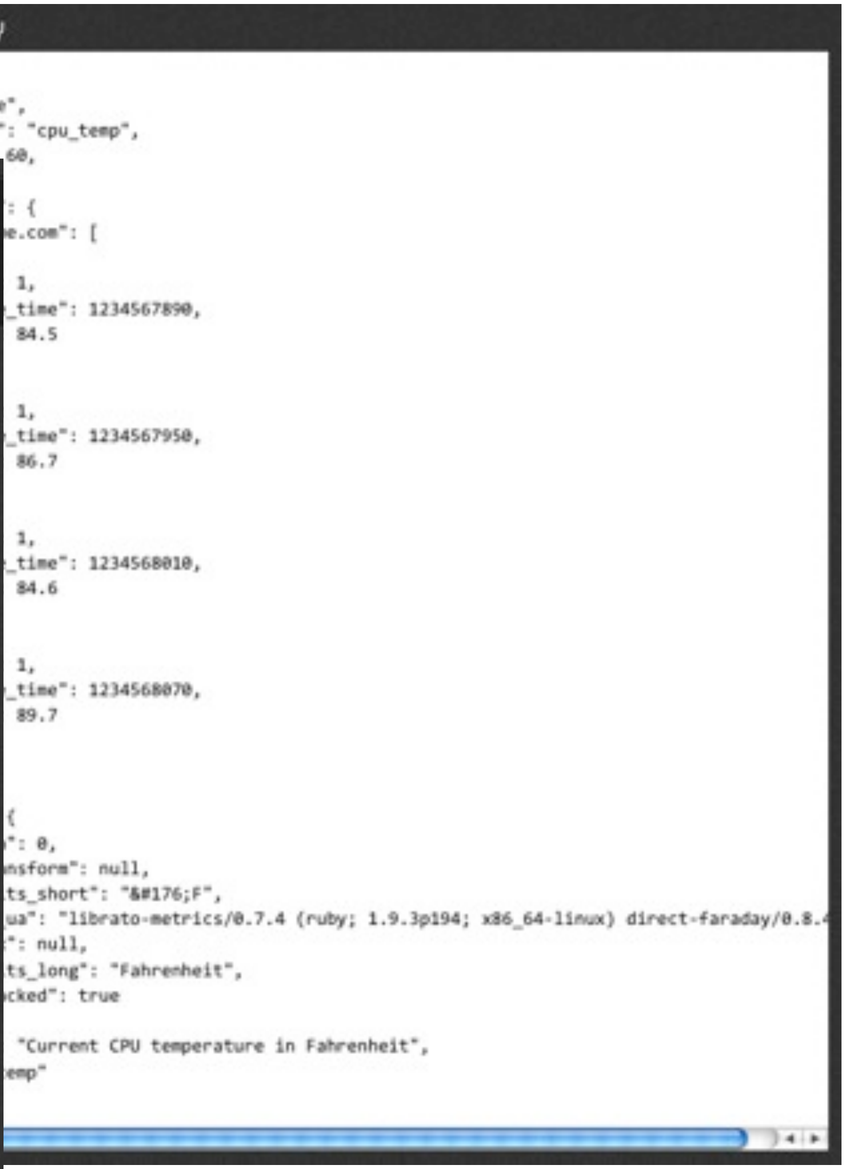
```
curl \
  -u <user>:<token> \
  -X GET \
  https://metrics-api.librato.com/v1/metrics/cpu_temp?resolution=60&count=4
```

Response Code

```
200 OK
```

Response Headers

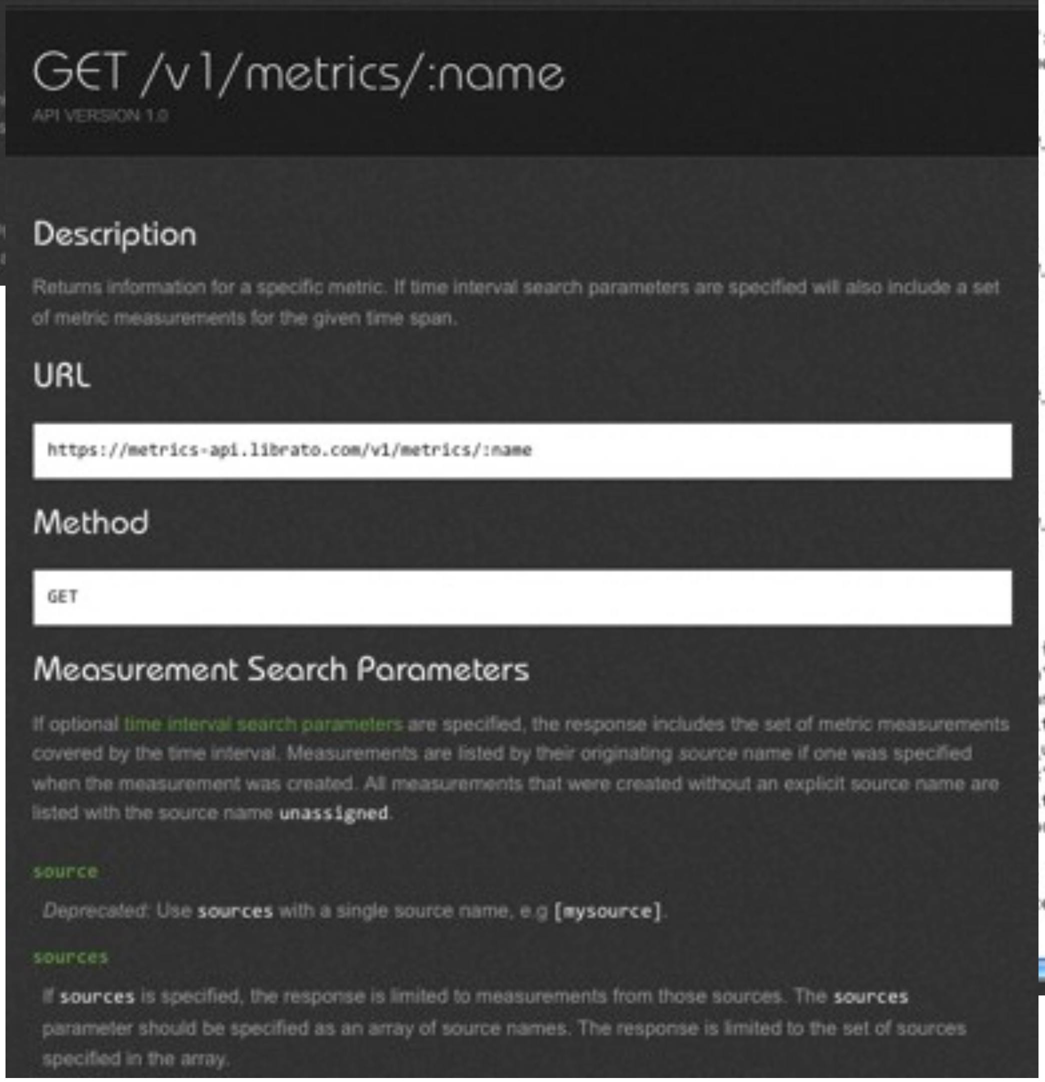
```
** NOT APPLICABLE **
```



Response Body

```
{
  "type": "gauge",
  "display_name": "cpu_temp",
  "resolution": 60,

          : {
          e.com": [

            1,
            _time": 1234567890,
            84.5


            1,
            _time": 1234567950,
            86.7


            1,
            _time": 1234568010,
            84.6


            1,
            _time": 1234568070,
            89.7



          {
          ": 0,
          nsform": null,
          ts_short": "&#176;F",
          ua": "librato-metrics/0.7.4 (ruby; 1.9.3p194; x86_64-linux) direct-faraday/0.8.4
          ": null,
          ts_long": "Fahrenheit",
          cked": true

            "Current CPU temperature in Fahrenheit",
          emp"
```

CRUD method description



GET /v1/metrics/:name

API VERSION 1.0

Description

Returns information for a specific metric. If time interval search parameters are specified will also include a set of metric measurements for the given time span.

URL

```
https://metrics-api.librato.com/v1/metrics/:name
```

Method

```
GET
```

Measurement Search Parameters

If optional time interval search parameters are specified, the response includes the set of metric measurements covered by the time interval. Measurements are listed by their originating *source* name if one was specified when the measurement was created. All measurements that were created without an explicit source name are listed with the source name **`unassigned`**.

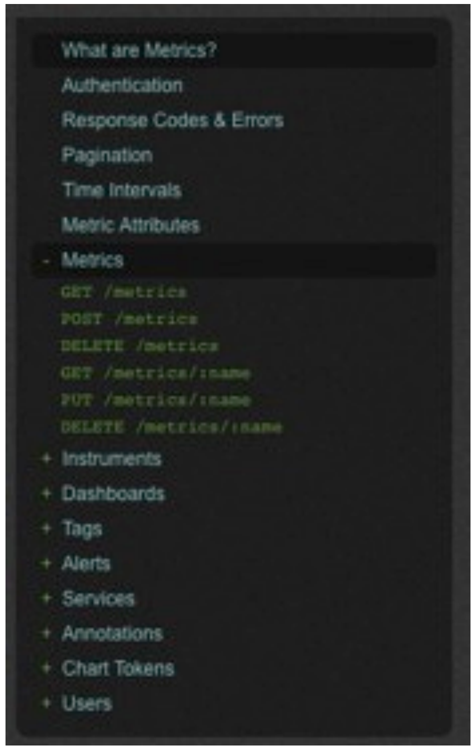
`source`

Deprecated: Use **`sources`** with a single source name, e.g **`[mysource]`**.

`sources`

If **`sources`** is specified, the response is limited to measurements from those sources. The **`sources`** parameter should be specified as an array of source names. The response is limited to the set of sources specified in the array.

navigator

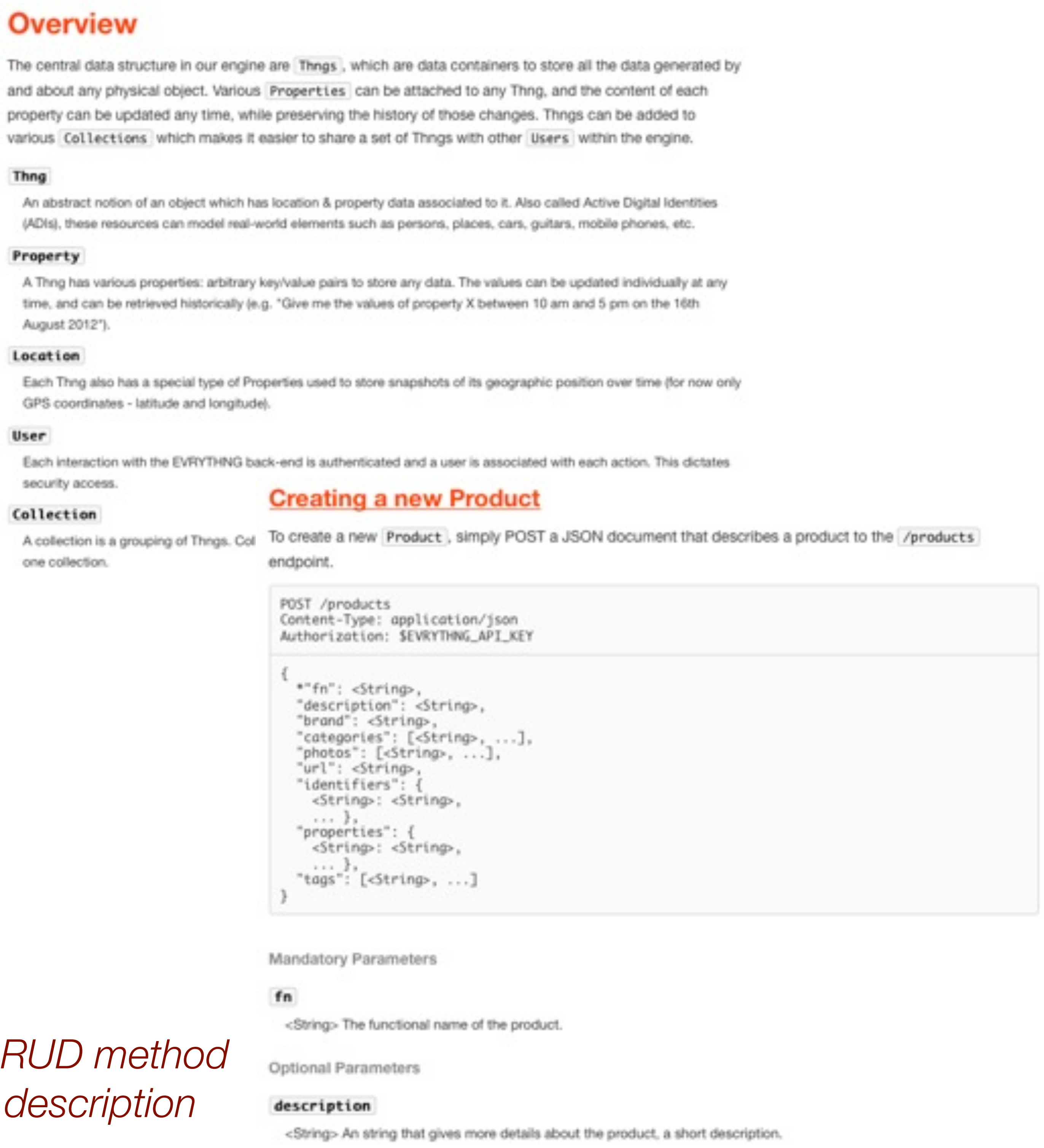


EVRYTHNG™

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Short description of the whole domain model

More details about the Product resource (domain model) & payload structure



Overview

The central data structure in our engine are `Thngs`, which are data containers to store all the data generated by and about any physical object. Various `Properties` can be attached to any Thng, and the content of each property can be updated any time, while preserving the history of those changes. Thngs can be added to various `Collections` which makes it easier to share a set of Thngs with other `Users` within the engine.

`Thng`

An abstract notion of an object which has location & property data associated to it. Also called Active Digital Identities (ADIs), these resources can model real-world elements such as persons, places, cars, guitars, mobile phones, etc.

`Property`

A Thng has various properties: arbitrary key/value pairs to store any data. The values can be updated individually at any time, and can be retrieved historically (e.g. "Give me the values of property X between 10 am and 5 pm on the 16th August 2012").

`Location`

Each Thng also has a special type of Properties used to store snapshots of its geographic position over time (for now only GPS coordinates - latitude and longitude).

`User`

Each interaction with the EVRYTHNG back-end is authenticated and a user is associated with each action. This dictates security access.

`Collection`

A collection is a grouping of Thngs. Col
one collection.

Creating a new Product

To create a new `Product`, simply POST a JSON document that describes a product to the `/products` endpoint.

```
POST /products
Content-Type: application/json
Authorization: $EVRYTHNG_API_KEY
```

```
{
  *"fn": <String>,
  "description": <String>,
  "brand": <String>,
  "categories": [<String>, ...],
  "photos": [<String>, ...],
  "url": <String>,
  "identifiers": {
    <String>: <String>,
    ... },
  "properties": {
    <String>: <String>,
    ... },
  "tags": [<String>, ...]
}
```

Mandatory Parameters

`fn`

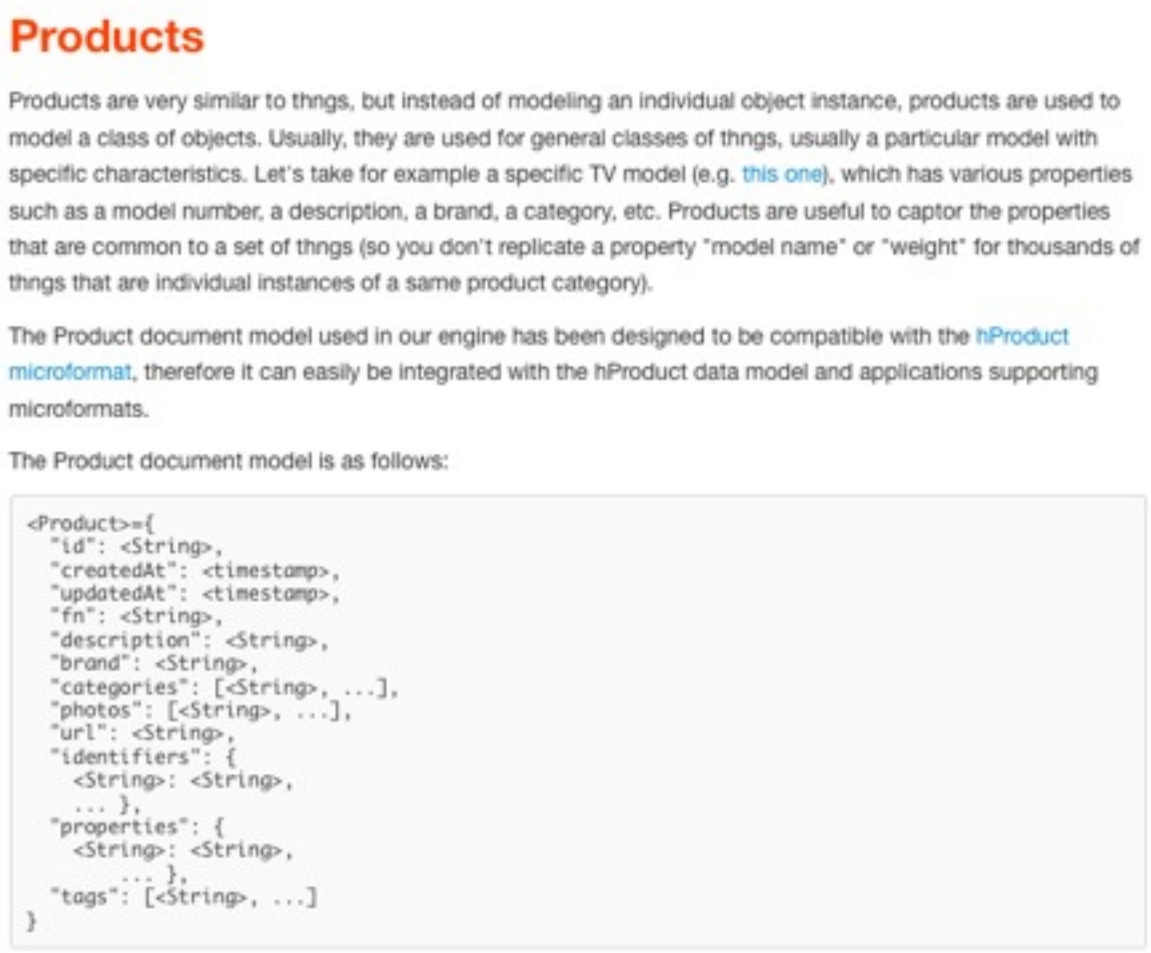
<String> The functional name of the product.

Optional Parameters

`description`

<String> An string that gives more details about the product, a short description.

CRUD method description



Products

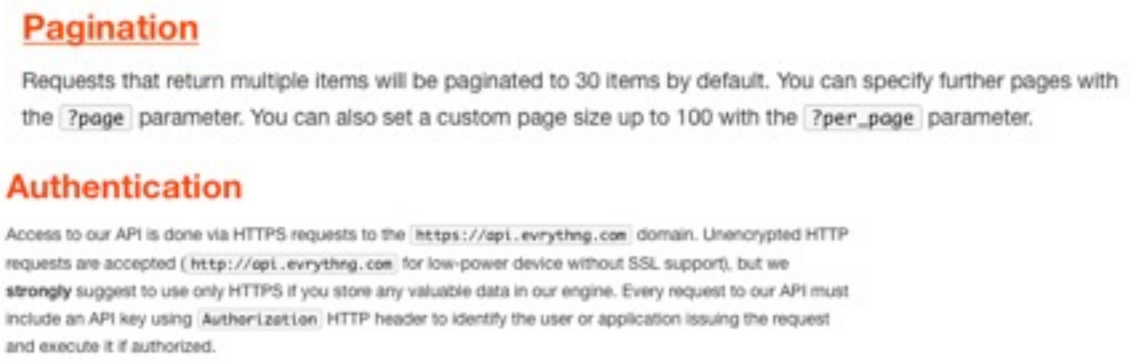
Products are very similar to thngs, but instead of modeling an individual object instance, products are used to model a class of objects. Usually, they are used for general classes of thngs, usually a particular model with specific characteristics. Let's take for example a specific TV model (e.g. this one), which has various properties such as a model number, a description, a brand, a category, etc. Products are useful to captor the properties that are common to a set of thngs (so you don't replicate a property "model name" or "weight" for thousands of thngs that are individual instances of a same product category).

The Product document model used in our engine has been designed to be compatible with the hProduct microformat, therefore it can easily be integrated with the hProduct data model and applications supporting microformats.

The Product document model is as follows:

```
<Product>={
  "id": <String>,
  "createdAt": <timestamp>,
  "updatedAt": <timestamp>,
  "fn": <String>,
  "description": <String>,
  "brand": <String>,
  "categories": [<String>, ...],
  "photos": [<String>, ...],
  "url": <String>,
  "identifiers": {
    <String>: <String>,
    ... },
  "properties": {
    <String>: <String>,
    ... },
  "tags": [<String>, ...]
}
```

Cross-cutting concerns



Pagination

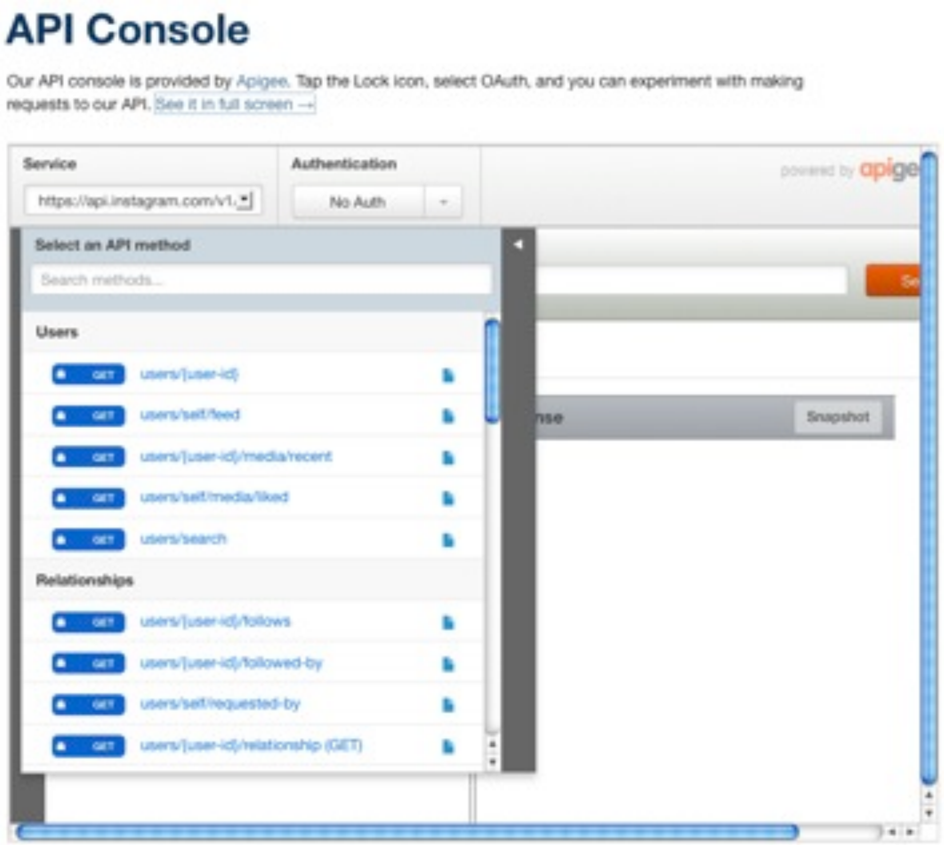
Requests that return multiple items will be paginated to 30 items by default. You can specify further pages with the `?page` parameter. You can also set a custom page size up to 100 with the `?per_page` parameter.

Authentication

Access to our API is done via HTTPS requests to the `https://api.evrythng.com` domain. Unencrypted HTTP requests are accepted (`http://api.evrythng.com` for low-power device without SSL support), but we **strongly** suggest to use only HTTPS if you store any valuable data in our engine. Every request to our API must include an API key using `Authorization` HTTP header to identify the user or application issuing the request and execute it if authorized.

Instagram

Interactive test console

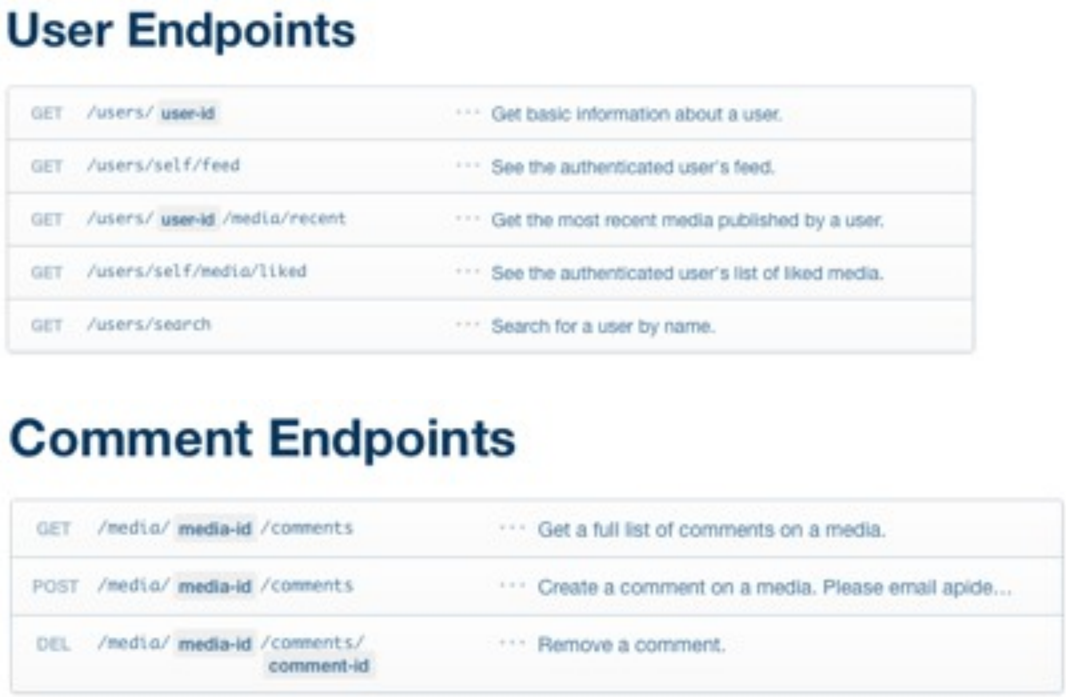


API Console

Our API console is provided by Apigee. Tap the Lock icon, select OAuth, and you can experiment with making requests to our API. See it in full screen →

Service
https://api.instagram.com/v1
Authentication
No Auth
powered by apigee
Select an API method
Search methods...
Users
GET users/[user-id]
GET users/self/feed
GET users/[user-id]/media/recent
GET users/self/media/liked
GET users/search
Relationships
GET users/[user-id]/follows
GET users/[user-id]/followed-by
GET users/self/requested-by
GET users/[user-id]/relationship (GET)
nse
Snapshot

List of supported CRUD methods for each resource (R, R/W)

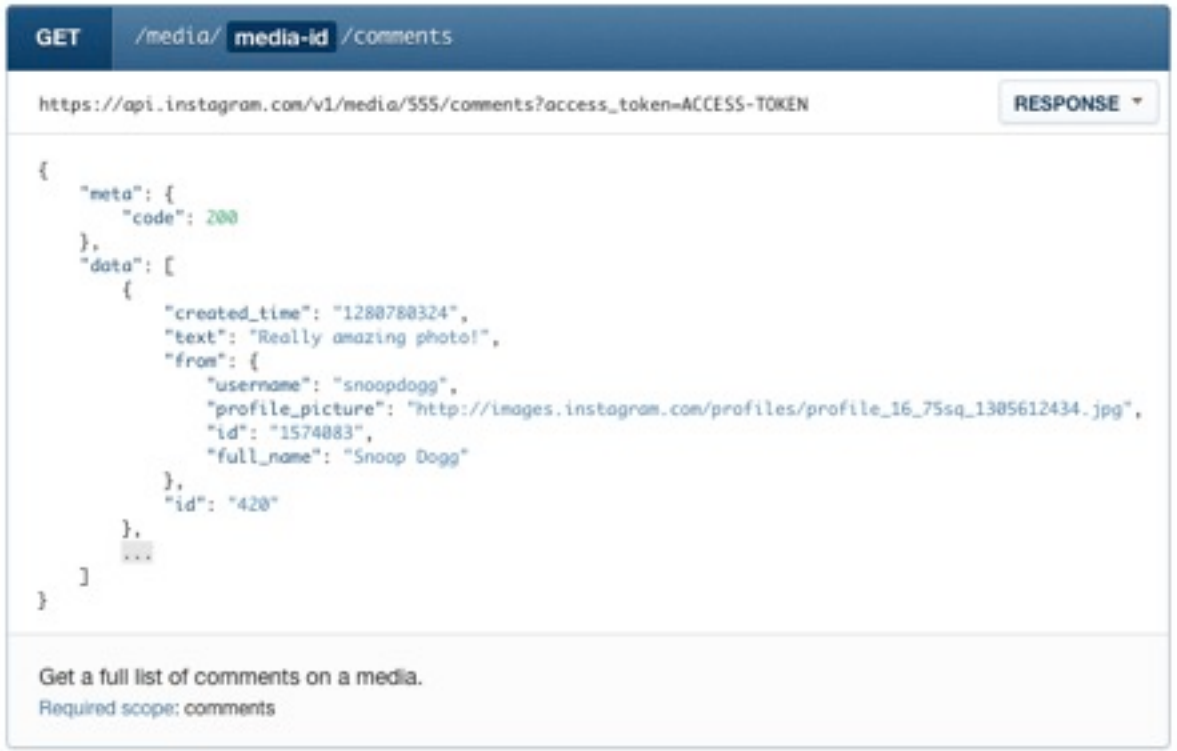


User Endpoints

GET	/users/ user-id	··· Get basic information about a user.
GET	/users/self/feed	··· See the authenticated user's feed.
GET	/users/ user-id /media/recent	··· Get the most recent media published by a user.
GET	/users/self/media/liked	··· See the authenticated user's list of liked media.
GET	/users/search	··· Search for a user by name.

Comment Endpoints

GET	/media/ media-id /comments	··· Get a full list of comments on a media.
POST	/media/ media-id /comments	··· Create a comment on a media. Please email apide...
DEL	/media/ media-id /comments/ comment-id	··· Remove a comment.



GET /media/ media-id /comments

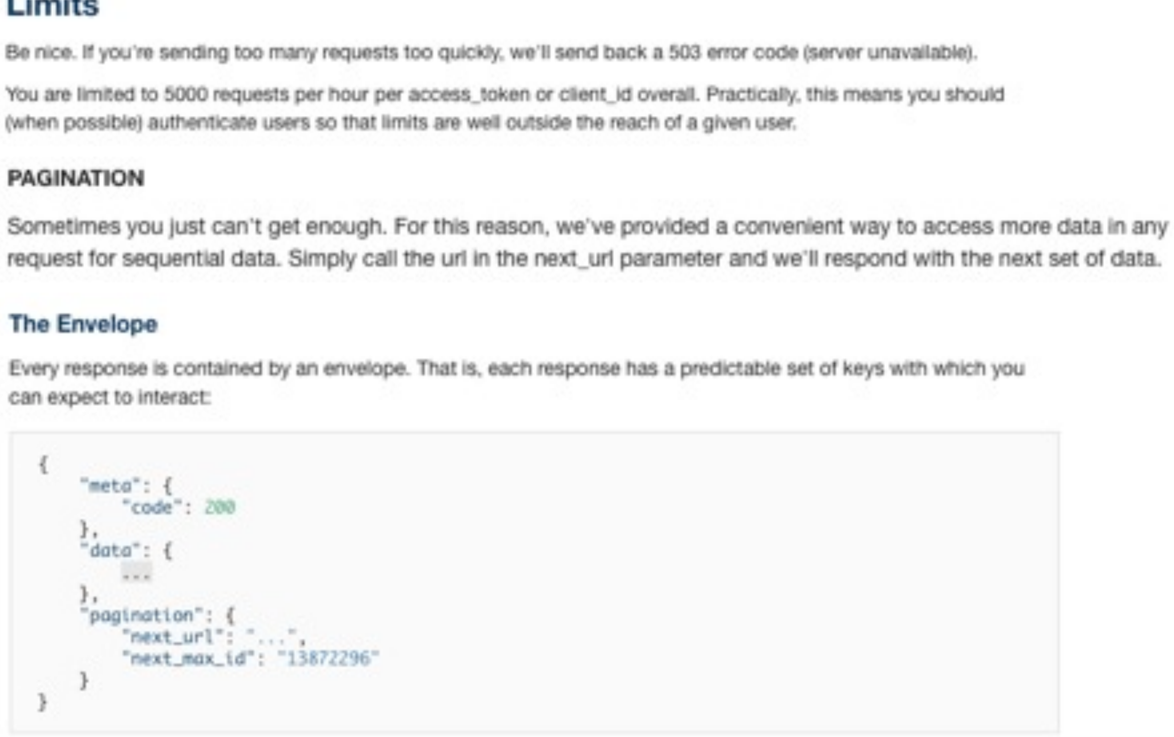
https://api.instagram.com/v1/media/555/comments?access_token=ACCESS-TOKEN

```
{
    "meta": {
        "code": 200
    },
    "data": [
        {
            "created_time": "1280780324",
            "text": "Really amazing photo!",
            "from": {
                "username": "snoopdogg",
                "profile_picture": "http://images.instagram.com/profiles/profile_16_75sq_1305612434.jpg",
                "id": "1574083",
                "full_name": "Snoop Dogg"
            },
            "id": "420"
        },
        ...
    ]
}
```

Get a full list of comments on a media.
Required scope: comments

CRUD method description

Cross-cutting concerns



Limits

Be nice. If you're sending too many requests too quickly, we'll send back a 503 error code (server unavailable).

You are limited to 5000 requests per hour per access_token or client_id overall. Practically, this means you should (when possible) authenticate users so that limits are well outside the reach of a given user.

PAGINATION

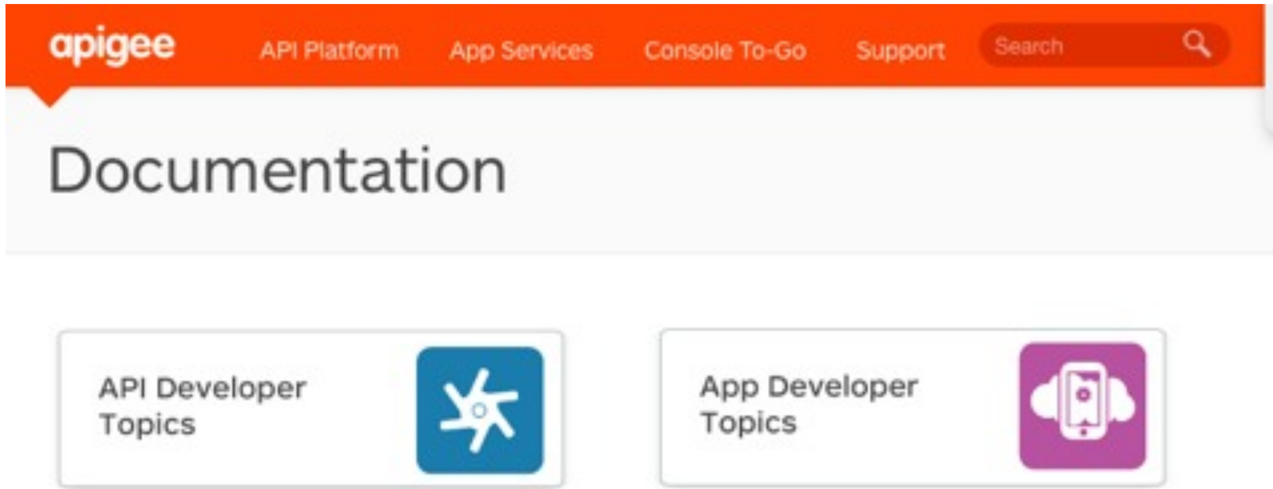
Sometimes you just can't get enough. For this reason, we've provided a convenient way to access more data in any request for sequential data. Simply call the url in the next_url parameter and we'll respond with the next set of data.

The Envelope

Every response is contained by an envelope. That is, each response has a predictable set of keys with which you can expect to interact:

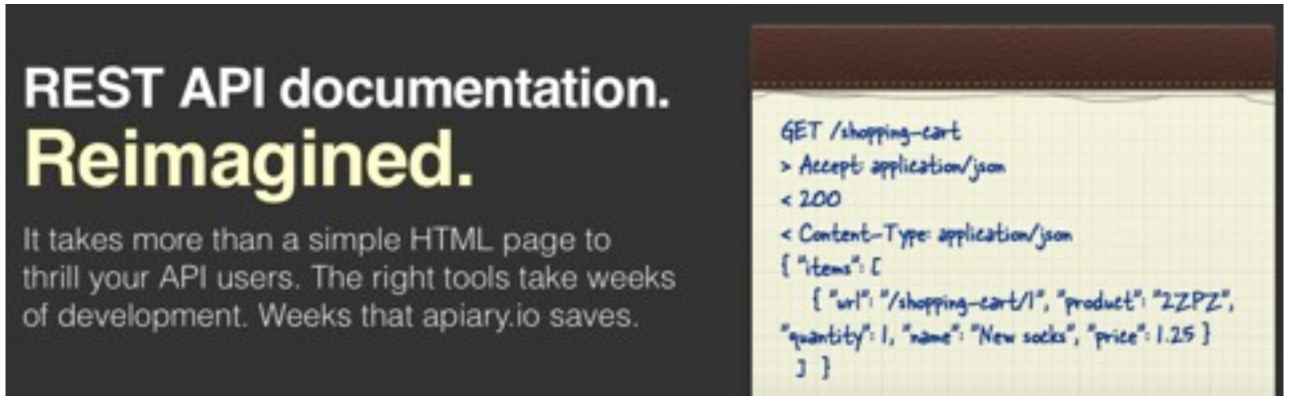
```
{
    "meta": {
        "code": 200
    },
    "data": {
        ...
    },
    "pagination": {
        "next_url": "...",
        "next_max_id": "13872296"
    }
}
```

Some Tools that Might Help/Inspire You



http://apigee.com/docs/

apiary.io beta



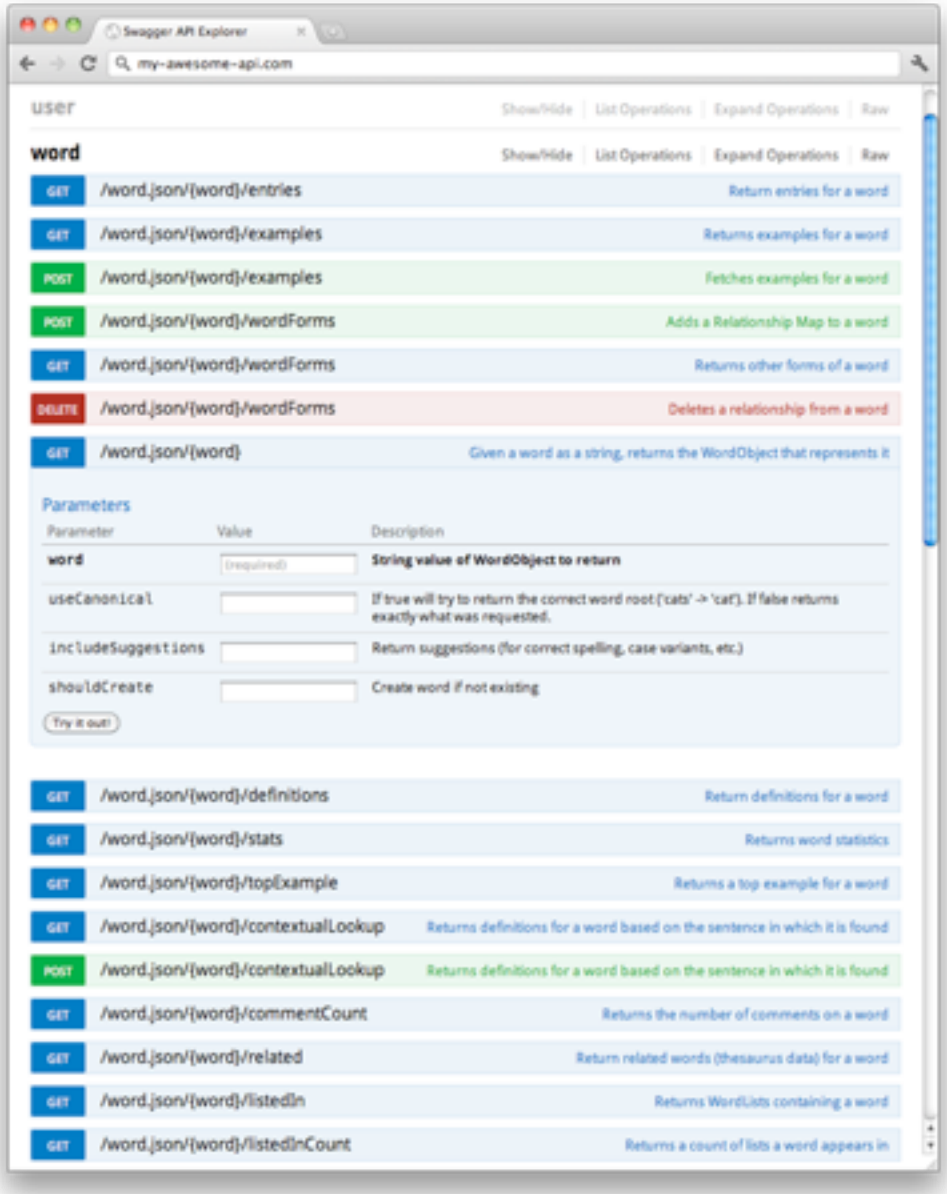
REST API documentation.
Reimagined.

It takes more than a simple HTML page to thrill your API users. The right tools take weeks of development. Weeks that apiary.io saves.

```
GET /shopping-cart
> Accept: application/json
< 200
< Content-Type: application/json
{ "items": [
    { "url": "/shopping-cart/1", "product": "2ZPZ",
"quantity": 1, "name": "New socks", "price": 1.25 }
  ] }
```

http://apiary.io/

FOR DEVELOPERS
Swagger


Swagger API Explorer
my-awesome-api.com
user
Show/Hide | List Operations | Expand Operations | Raw
word
Show/Hide | List Operations | Expand Operations | Raw
Try it out!

GET	/word.json/{word}/entries	Return entries for a word
GET	/word.json/{word}/examples	Returns examples for a word
POST	/word.json/{word}/examples	Fetches examples for a word
POST	/word.json/{word}/wordForms	Adds a Relationship Map to a word
GET	/word.json/{word}/wordForms	Returns other forms of a word
DELETE	/word.json/{word}/wordForms	Deletes a relationship from a word
GET	/word.json/{word}	Given a word as a string, returns the WordObject that represents it

Parameters

Parameter	Value	Description
word	(required)	**String value of WordObject to return**
useCanonical		If true will try to return the correct word root ('cats' -> 'cat'). If false returns exactly what was requested.
includeSuggestions		Return suggestions (for correct spelling, case variants, etc.)
shouldCreate		Create word if not existing

GET	/word.json/{word}/definitions	Return definitions for a word
GET	/word.json/{word}/stats	Returns word statistics
GET	/word.json/{word}/topExample	Returns a top example for a word
GET	/word.json/{word}/contextualLookup	Returns definitions for a word based on the sentence in which it is found
POST	/word.json/{word}/contextualLookup	Returns definitions for a word based on the sentence in which it is found
GET	/word.json/{word}/commentCount	Returns the number of comments on a word
GET	/word.json/{word}/related	Return related words (thesaurus data) for a word
GET	/word.json/{word}/listedIn	Returns WordLists containing a word
GET	/word.json/{word}/listedInCount	Returns a count of lists a word appears in

https://developers.helloreverb.com/swagger/

How to write a “RESTful” Web Service?

- On the server side, one could do everything in a FrontController servlet:
  - Parse URLs
  - Do a mapping between URLs and Java classes that represent resources
  - Generate the different representations of resources
  - etc.
- But of course, there are frameworks that do exactly that for us.
- It is true for nearly every platform and language, including Java.
- There is even a JSR for that: JAX-RS (JSR 311).
  - Oracle provides the reference implementation, in the Jersey project (open source).

Big Web Services with Java EE

- **JAX-RS**
  - JAX-RS provides a programming model, classes and annotations for easily building RESTful APIs.
  - Jersey is the name of the standard JAX-RS implementation, which is bundled with the Glassfish application server.
- **JAXB**
  - You do not have to worry about the serialization of your business objects to XML or JSON. The framework will take care of (most of) the details for you.
- For all of your business "resources", create a **JAX-RS resource class**
  - Use **annotations** to **route HTTP requests to your resource class and methods** (based on target URI, HTTP method, HTTP accept header, etc.)

Java

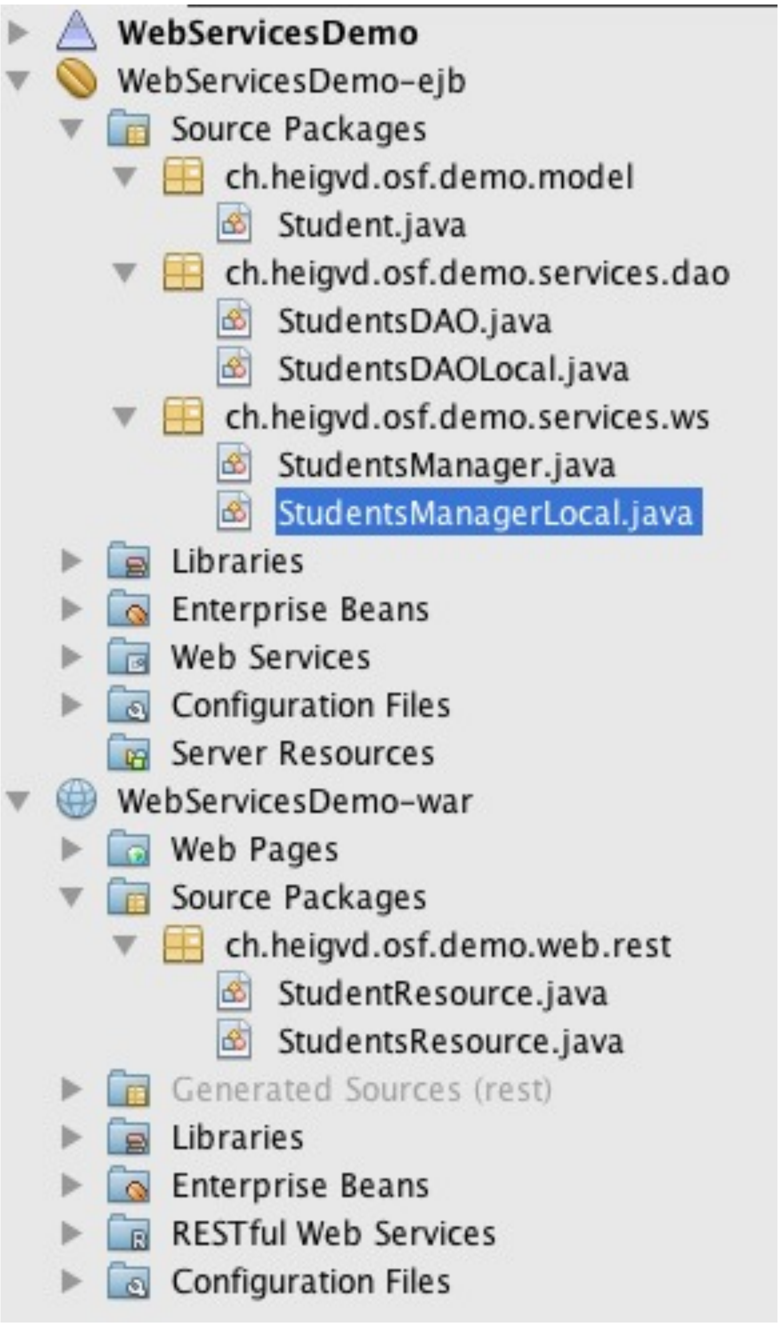
JavaOne℠

Developing RESTful Web Services with JAX-RS

Marc Hadley
Paul Sandoz
Sun Microsystems, Inc

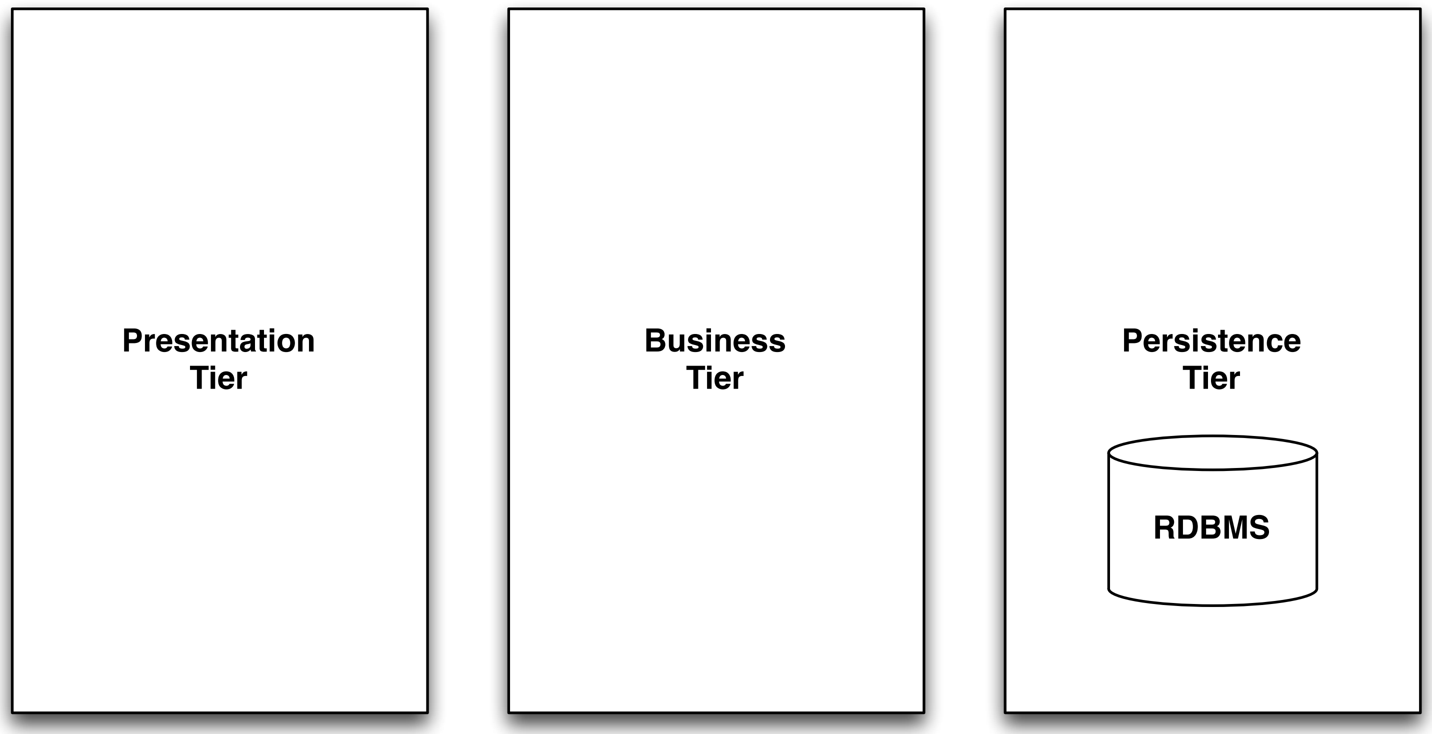
Example

```
@Path("/students")
public class StudentsResource {

      StudentsDAOLocal studentsDAO = lookupStudentsDAOLocal();

      @Context
      private UriInfo context;


      /**
       * Creates a new instance of StudentsResource
       */
      public StudentsResource() {
      }

      /**
       * Retrieves representation of the collection resource
       * @return an instance of List<Student>
       */
      @GET
      @Produces("application/xml, application/json")
      public List<Student> getXml() {
            // Let's generate random students for demo purposes...
            // don't try to understand the logic of this
            List<Student> dummyResult = new LinkedList<Student>();
            dummyResult.add(studentsDAO.findStudentById(42));
            dummyResult.add(studentsDAO.findStudentById(42));
            dummyResult.add(studentsDAO.findStudentById(42));
            dummyResult.add(studentsDAO.findStudentById(42));
            dummyResult.add(studentsDAO.findStudentById(42));
            return dummyResult;
      }

  ...
}
```

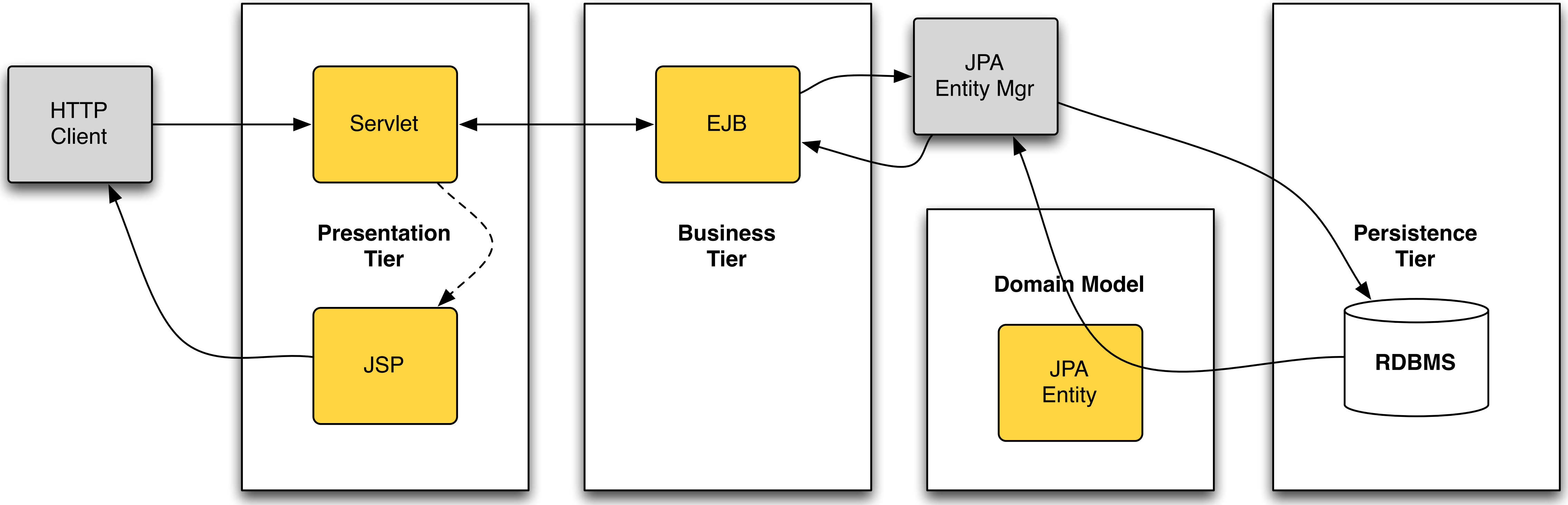


```
▶ WebServicesDemo
▼ WebServicesDemo-ejb
    ▼ Source Packages
        ▼ ch.heigvd.osf.demo.model
              Student.java
        ▼ ch.heigvd.osf.demo.services.dao
              StudentsDAO.java
              StudentsDAOLocal.java
        ▼ ch.heigvd.osf.demo.services.ws
              StudentsManager.java
              StudentsManagerLocal.java
    ▶ Libraries
    ▶ Enterprise Beans
    ▶ Web Services
    ▶ Configuration Files
      Server Resources
▼ WebServicesDemo-war
    ▶ Web Pages
    ▼ Source Packages
        ▼ ch.heigvd.osf.demo.web.rest
              StudentResource.java
              StudentsResource.java
    ▶ Generated Sources (rest)
    ▶ Libraries
    ▶ Enterprise Beans
    ▶ RESTful Web Services
    ▶ Configuration Files
```

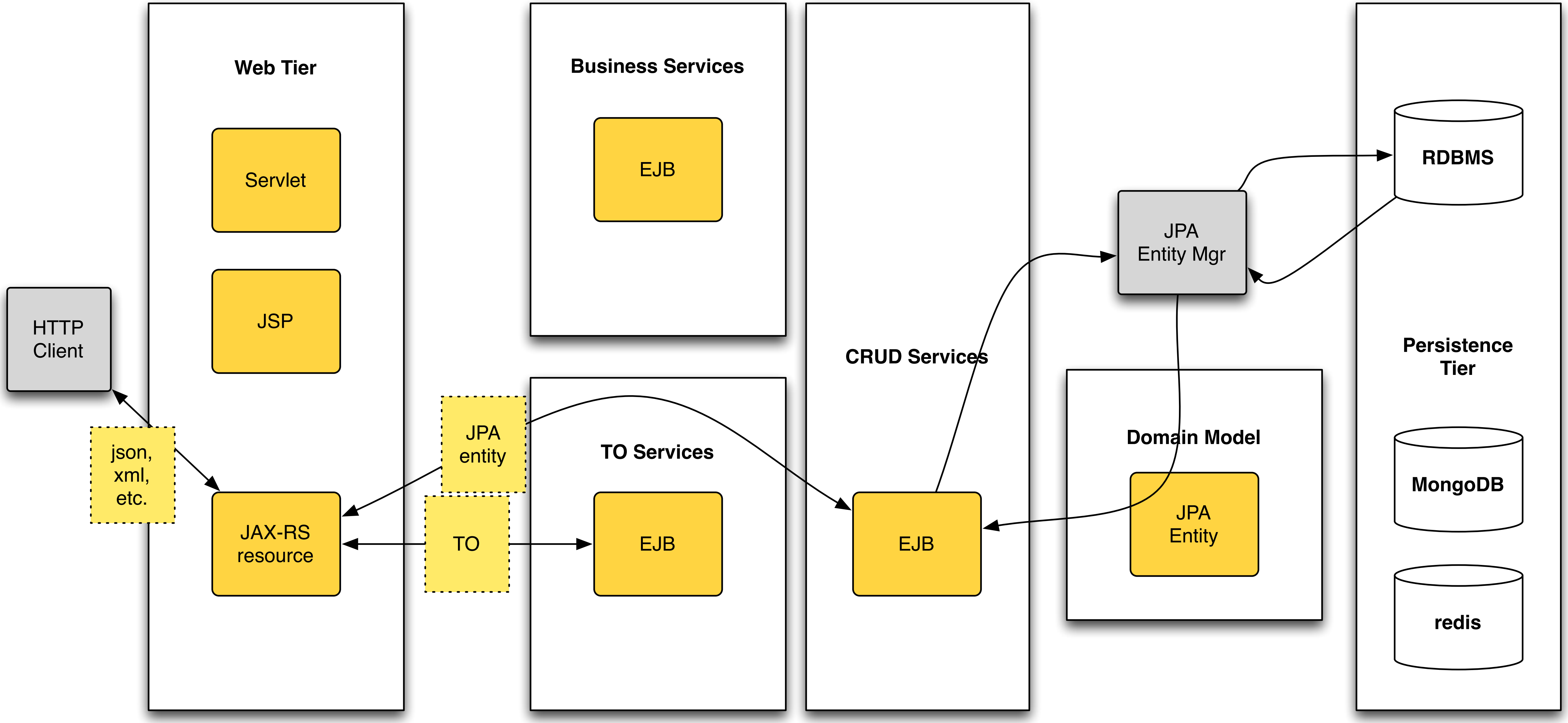
Reference Architecture



Reference Architecture



Reference Architecture



Week 1: RESTful Web Services

- **Monday morning**
  - Intro to EJBs & JPA
  - Design your Observations, Entities and Facts
- **Monday afternoon**
  - (Partial) implementation of your Observations, Entities and Facts.
  - (Partial) implementation of the corresponding DAOs
- **Tuesday morning**
  - Intro to REST & JAX-RS
  - Design of the different REST APIs for your service
- **Tuesday afternoon**
  - (Partial) implementation of your REST APIs
  - Implementation of test clients and/or of sensor simulators
- **Wednesday & Thursday**
  - Implementation & Demo