Lecture 7: REST API Design & Doc

Olivier Liechti
AMT

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

Agenda

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

13h00 - 13h30	30'	**Lecture** Domain modeling, project specifications.
13h30 - 14h00	30'	**Individual work** Domain modeling for the project
14h00 - 14h30	30'	**Lecture** API design & documentation
14h30 - 15h00	30'	**Tutorial** API documentation with RAML & apidoc-seed
15h00 - 15h30	*30'*	***Break***
15h30 - 18h00	150'	**Project** REST API design & documentation for the project

Initial planning

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

Intro				Project 1						Project 2					
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
18.09	25.09	2.10	9.10	16.10	30.10	6.11	13.11	20.11	27.11	4.12	11.12	18.12	8.01	15.01	22.01

Test 1

Test 2

Submit Project 1

Submit Project 2

Revised planning

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

Intro				Project 1							Project 2				
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
18.09	25.09	2.10	9.10	16.10	30.10	6.11	13.11	20.11	27.11	4.12	11.12	18.12	8.01	15.01	22.01

Test 1

Test 2

Submit 1
08.12.2014 8:00 AM

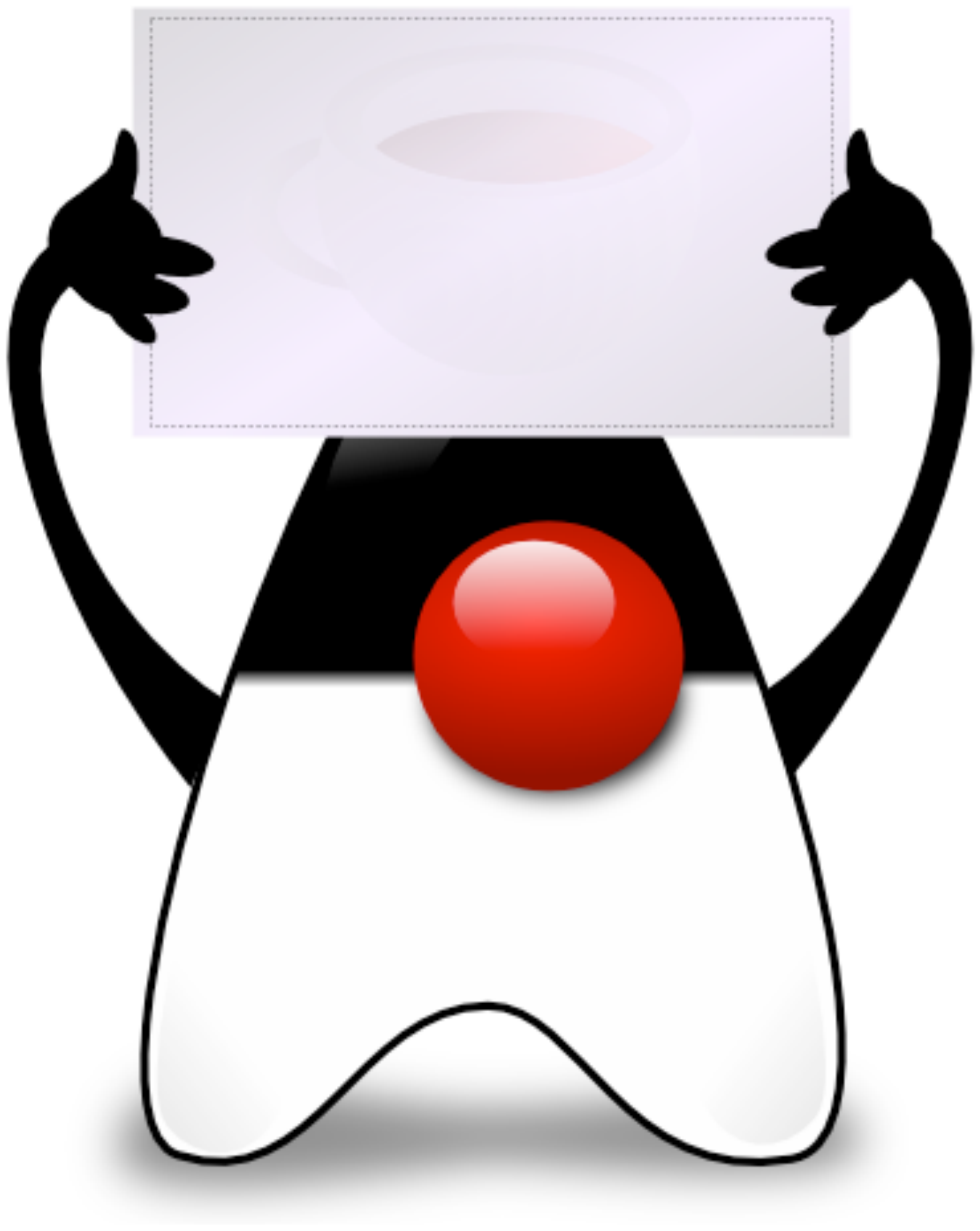
Submit P2
19.01.2014

Revised planning

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

8	9	10	11
13.11	20.11	27.11	4.12
Domain modeling REST API design REST API documentation	DTO layer impl. JAX-RS layer impl. Test data generation Sensor simulation impl.	DAO implementation JPA implementation	Test & validation Documentation

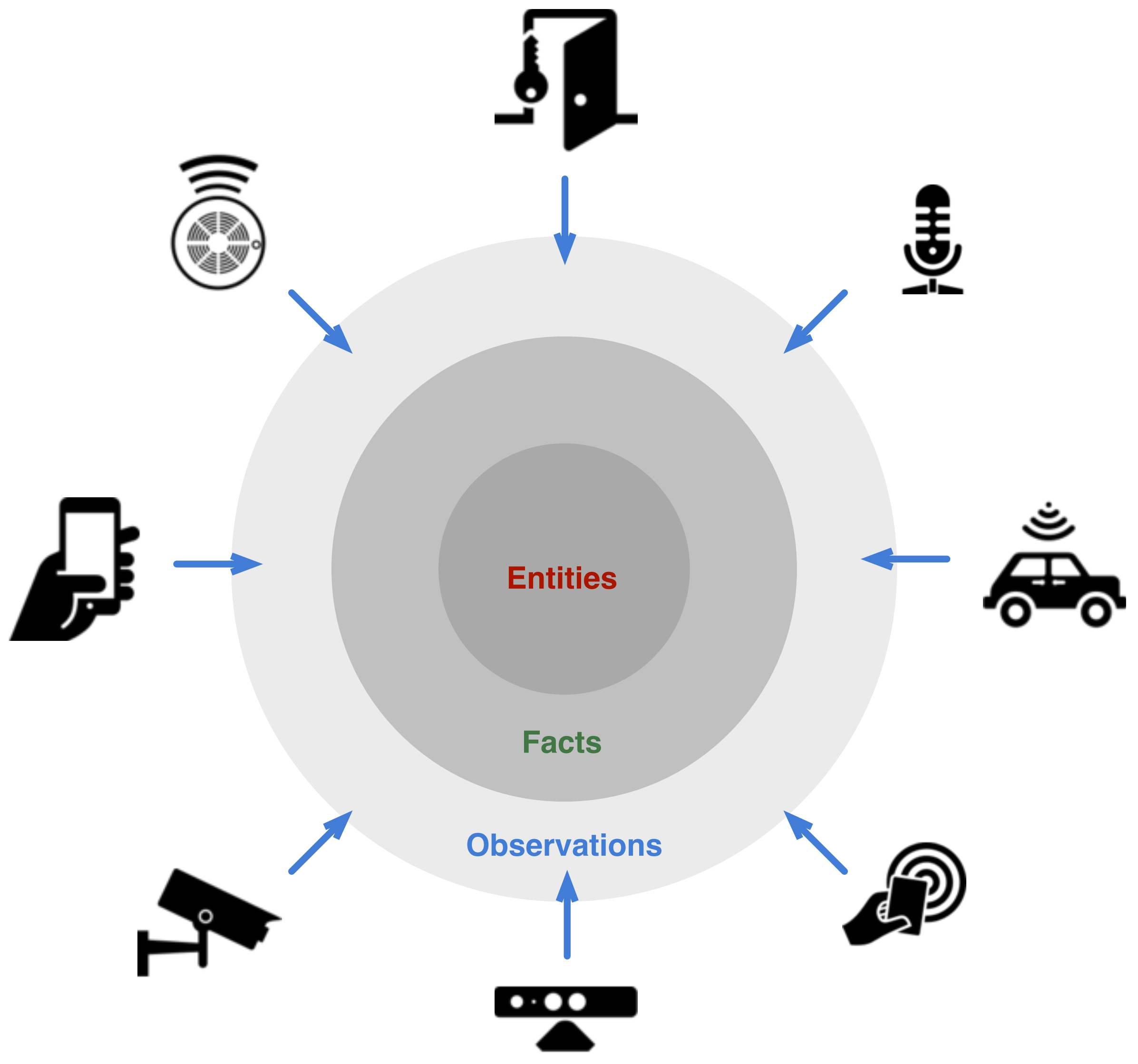
heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

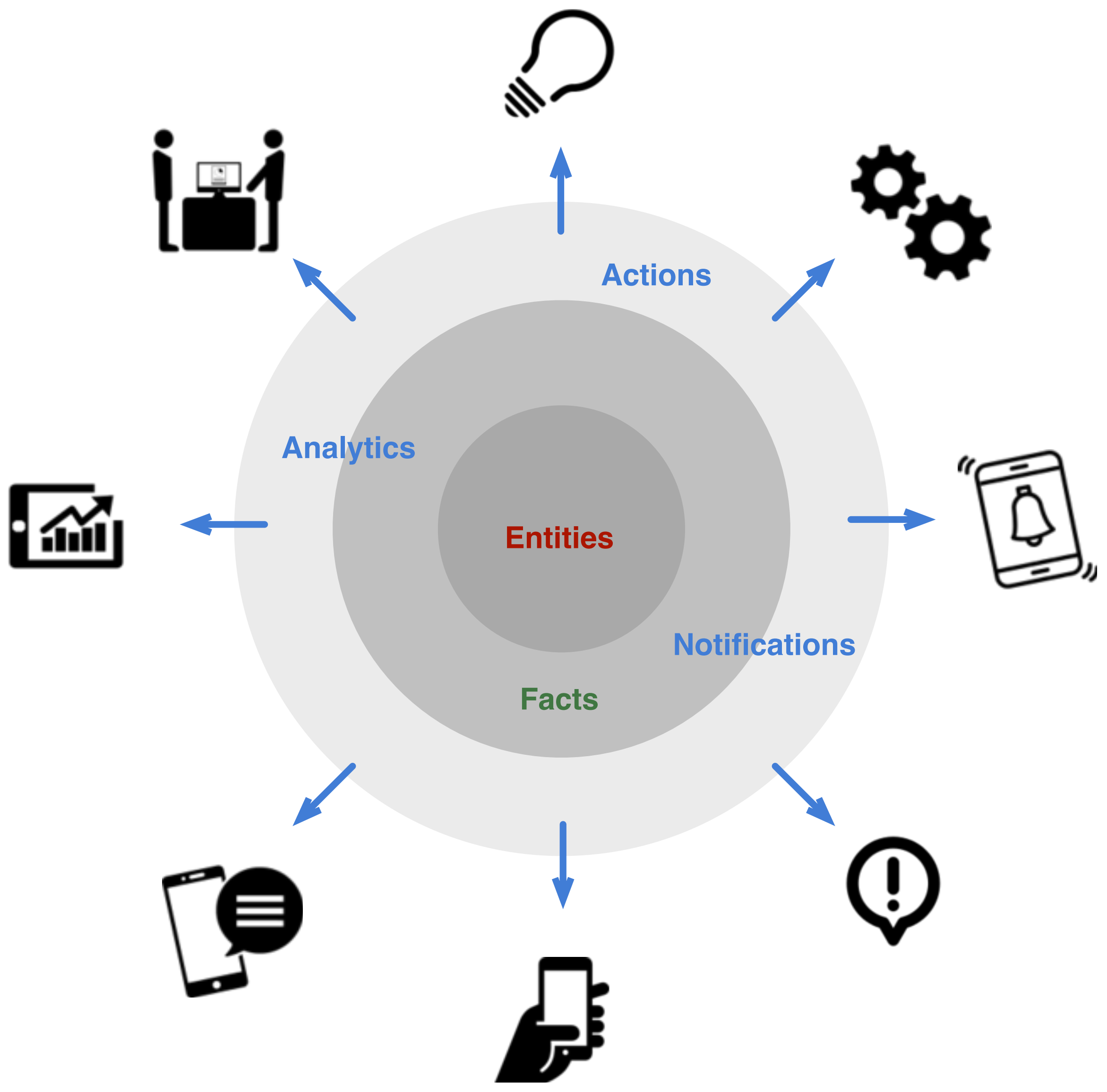


Domain Modeling

Web of Things

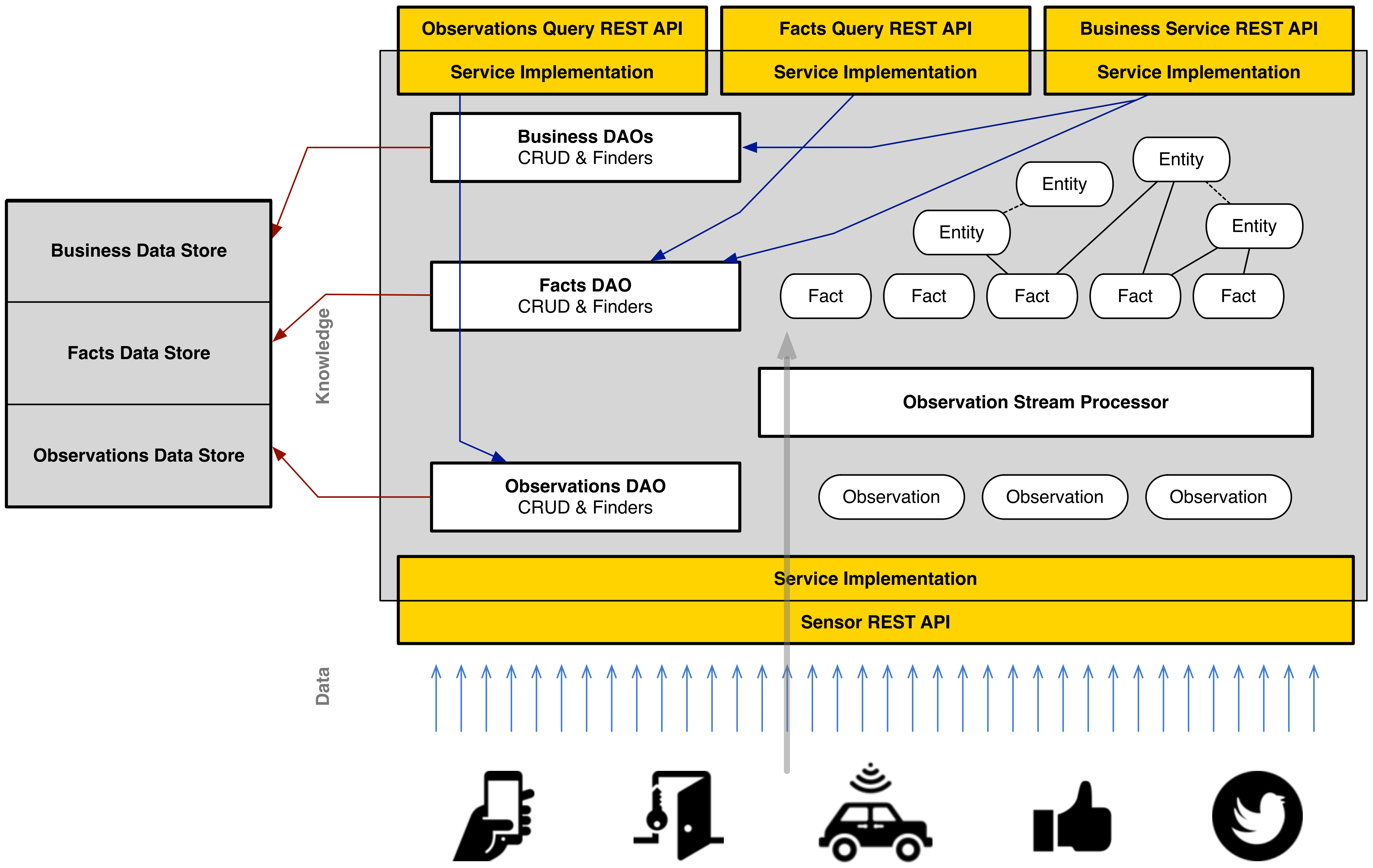
heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud





Web of Things

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



Specifications (1)

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

- You are building a Software-as-a-Service (SaaS) platform, which you are proposing to organizations (enterprises, cities, etc.).
- Users interact with your platform via different interfaces: REST APIs and Web UIs. Every user belongs to a single organization. Every organization has a contact person (one of its users).
- Organizations can create and manage sensors on the platform. Every sensor has (at least) a globally unique identifier, a name, a description, a type. A sensor can be declared as public or private. If it is declared as public, then it will be visible from other organizations (along with its associated observations).
- Sensors push observations to the platform, using a (write-only) REST API. An observation reports a sensor value at a given time.

Specifications (2)

Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- When your platform receives an observation from a sensor, several things happen:
  - The platform performs security checks to ensure that the observation is valid (authentication of the sensor, integrity of the payload).
  - The platform archives the observation in persistent storage.
  - The platform performs computation in order to create and/or update a collection of “facts”. A fact is a piece of information that has been gathered by processing one or more observations over a period of time.

Specifications (3)

- Examples of facts:
  - The sensor with an identifier of S000289 has reported a total of 8728 observations.
  - The sensor with an identifier of S000289 has reported 928 observations in March 2014.
  - The sensors belonging to organization O1 have reported a total of 129’992 observations in the last 24 hours.
  - The current temperature in room R01 is 23.2 degrees.
  - The average temperature in room R01 over the last 24 hours is 21.1 degrees.

Specifications (4)

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

- Examples of facts:
  - There are 232 sensors in the region centered around latitude X and longitude Y, with a radius of R meters.
  - 234 people have entered room R01 during the last 2 hours.
  - The room that has seen the most people enter on April 1st 2014 is room R03.
  - The room with the highest daily average temperature during last month is room R04. The average daily temperature is 22.3 degrees and the maximum daily temperature is 24.1 degrees.
- Facts also belong to organizations. They can be private (only users of the organization can consult them) or public (everybody can consult them). In any case, the facts are only read via the REST APIs.

Specifications (5)

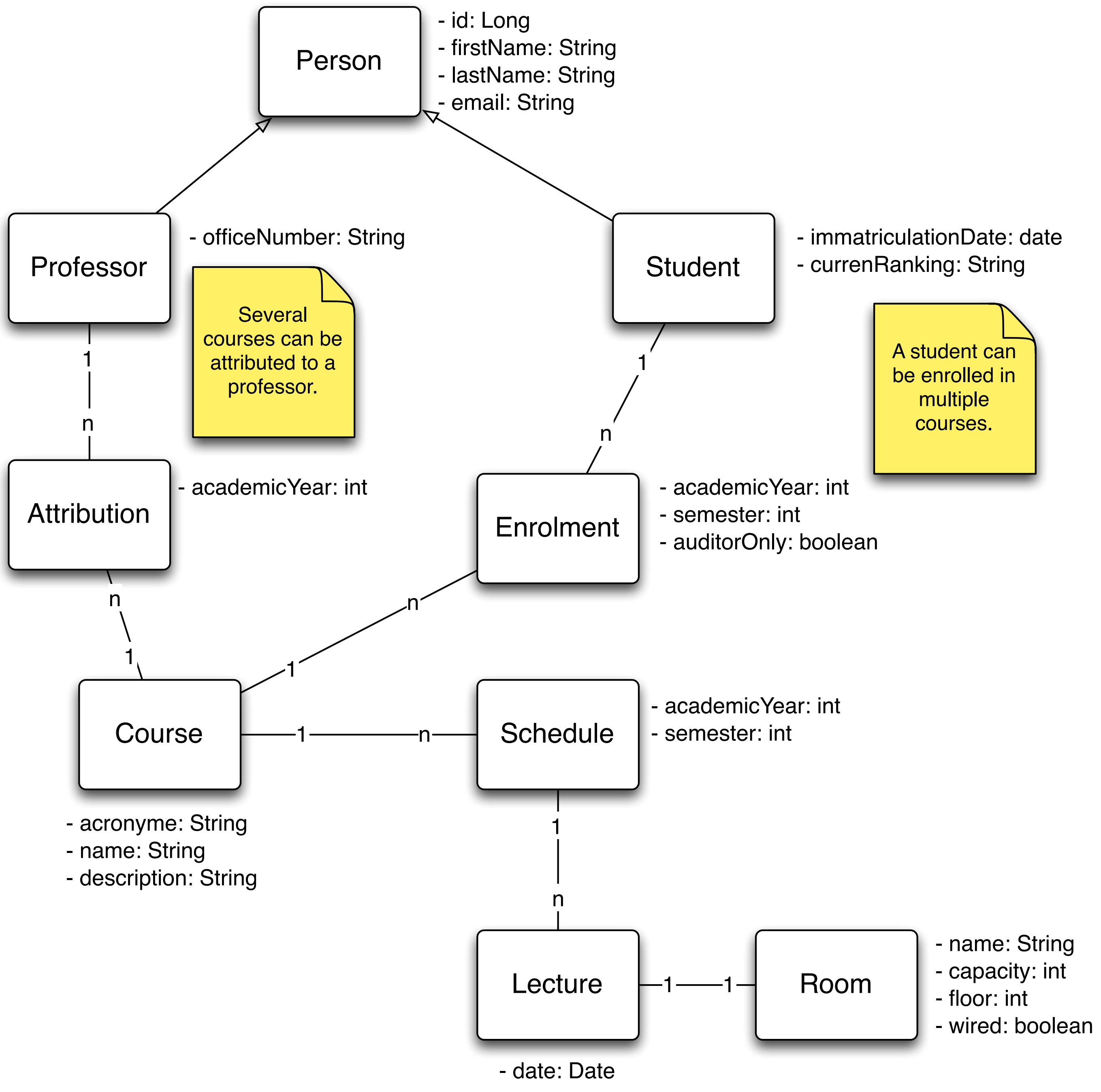
heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

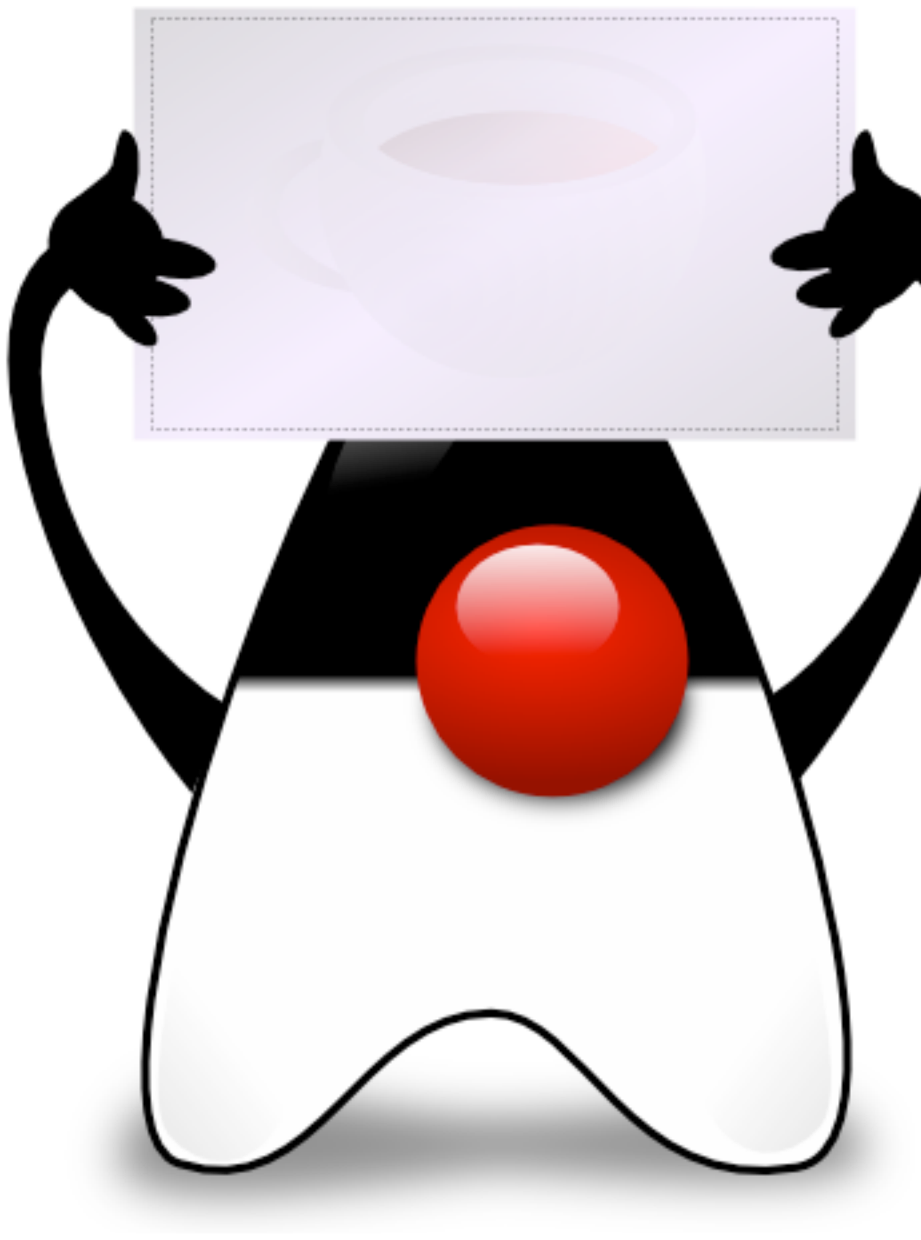
- While the notion of fact is generic, in the first project phase we will consider two simple examples:
    - A type of fact that keeps track of the total number of observations reported by a sensor.
    - A type of fact that keeps track of the number of observations, as well as their min/max/average values for one specific day (e.g. January 2nd 2012).

Individual work

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

- Based on the previous specifications, create a **domain model** for the project.
- Identify the **business concepts** and give them a name.
- Identify and document their **attributes** (name and type).
- Identify and document the **relationships** between business concepts.
- In 30', I will ask 2 students to present their diagram.





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

REST API Design

URL Structure (1)

- In most applications, you have **several types of resources** and there are relationships between them:
  - In a blog management platform, **Blog** authors create **BlogPosts** that can have associated **Comments**.
  - In a school management system, **Courses** are taught by **Professors** in **Rooms**.
- When you design your REST API, you have to define URL patterns to give access to the resources. There are often different ways to define these:
  - `/blogs/amtBlog/posts/892/comments/`
  - `/comments?blogId=amtBlog&postId=892`
  - `/professors/liechti/courses/`
  - `/courses?professorName=liechti`

URL Structure (2)

- How do you choose between these two approaches (flat vs deep structure)? There is no “right or wrong” answer and you find both styles in popular REST APIs.
- One rule that you can use to make your decision is whether you have an **aggregation** or **composition** relationship between resources. In other words, does the existence of one resource depend on the existence of another one. If that is the case, then it probably makes sense to use a hierarchical URL pattern.
- For instance, the existence of a **comment** depends on the existence of a **blog entry**. For this reason, I would probably go for:
  - `/blogs/amtBlog/posts/892/comments/`
- On the other hand, the existence of a **course** is not dependent on the existence of a **professor** (if one of you murders me, someone else will takeover the AMT course). Therefore, I would likely go for:
  - `/courses?professorName=liechti`

Resources & Actions (1)

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

- In some situations, it is fairly easy to **identify resource** and to map related **actions** to HTTP request patterns.
- For instance, in an **academic management system**, one would probably come up with a Student resource and the associated HTTP request patterns:
  - `GET /students` to retrieve a list of students
  - `GET /students/{id}` to retrieve a student by id
  - `POST /students` to create a student
  - `PUT /students/{id}` to update a student
  - `DELETE /students/{id}` to delete a student

Resources & Actions (2)

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

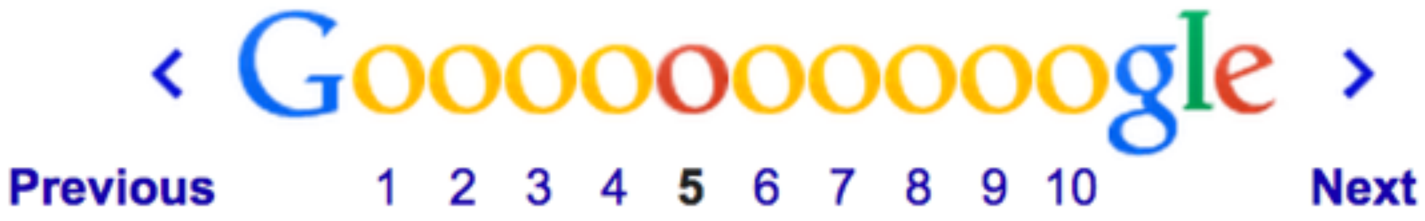
- Some situations are not as clear and subject to debate. For instance, let us imagine that with your system, you can **exclude students** if they have cheated at an exam. How do you implement that with a REST API?
- Some people would propose something like this:
  - `POST /students/{id}/exclude`
  - Notice that "exclude" is a verb. In that case, there is no request body and we do not introduce a new resource (we only have student).
- Other people (like me) would prefer something like this:
  - `POST /students/{id}/exclusions/`
  - In that case, we have introduced a new resource: an exclusion request (think about a form that the Dean has to fill out and file). In that case, we would have a request body (with the reasons for the exclusion, etc.).

Pagination (1)

- In most cases, you need to deal with **collections of resources that can grow** and where it is not possible to get the list of resources in a single HTTP request (for performance and efficiency reasons).
  - `GET /phonebook/entries?zip=1700`

- Instead, you want to be able to **successively retrieve chunks of the collection.** The typical use case is that you have a UI that presents a "page" of *n* resources, with controls to move to the previous, the next or an arbitrary page.

- In terms of API, it means that you want to be able to request a page, by providing an offset and a page size. In the response, you expect to find the number of results and a way to display navigation links.



Pagination (2)

- **At a minimum, what you need to do:**
  - When you **process an HTTP request**, you need a **page number** and a **page size.** You can use these to query a page from the database (do not transfer the whole table from the DB to the business tier!). You need to decide how the client is sending these values (query params, headers, defaults values).
  - When you generate the **HTTP response**, you need to **send the total number of results** (so that the client can compute the number of pages and generate the pagination UI), **and/or** send **ready-to-use links** that point to the first, last, prev and next pages. You use HTTP headers to send these informations.

```
Link: <https://api.github.com/user/repos?page=3&per_page=100>; rel="next",
  <https://api.github.com/user/repos?page=50&per_page=100>; rel="last"
```

Pagination (3)

- **Examples:**
  - http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api#pagination
  - https://developer.github.com/v3/#pagination
  - https://dev.evrythng.com/documentation/api

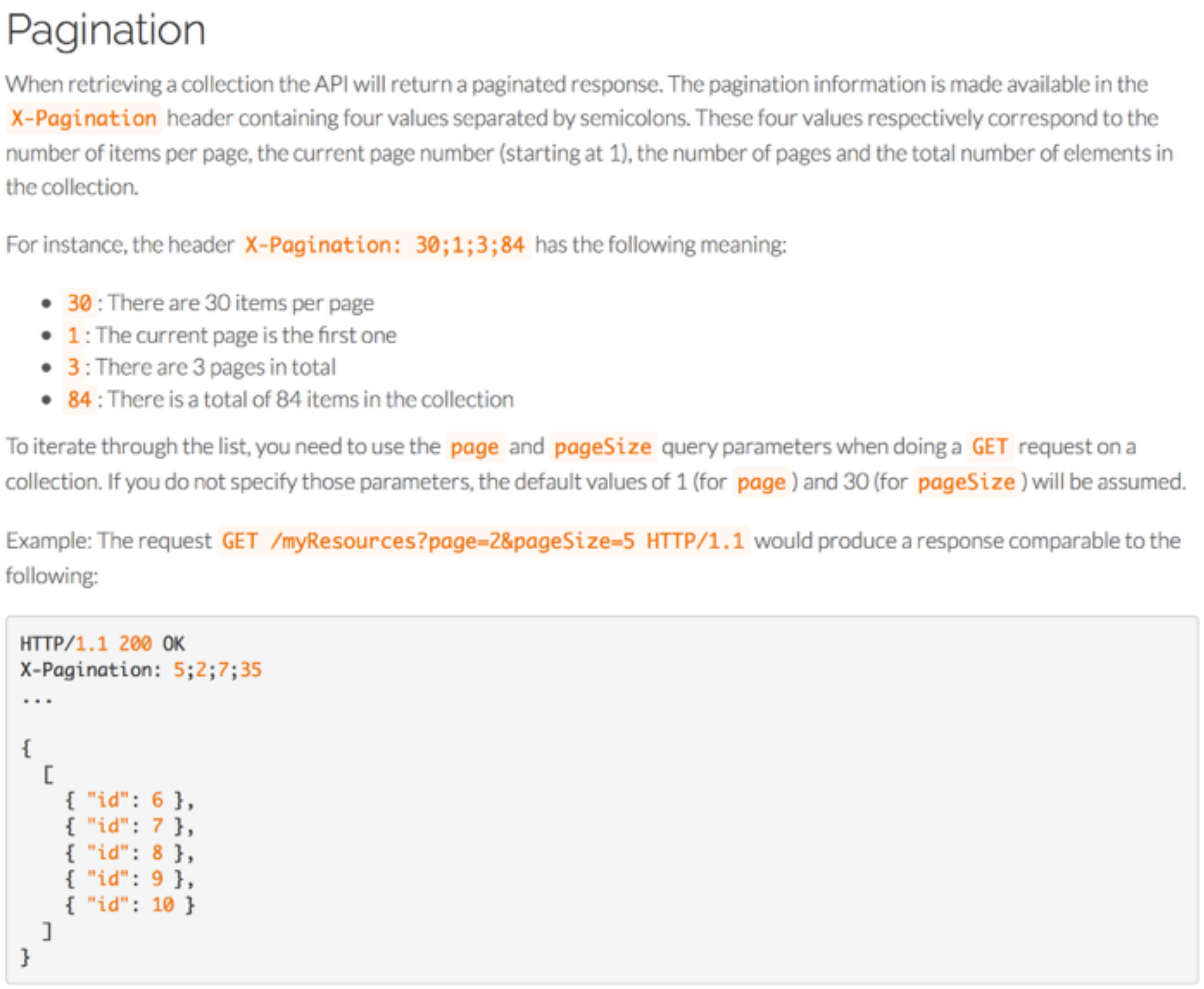
http://tools.ietf.org/html/rfc5988#page-6

```
Link: <https://api.github.com/user/repos?page=3&per_page=100>; rel="next",
  <https://api.github.com/user/repos?page=50&per_page=100>; rel="last"
```

Pagination (4)

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

- **Example:**



Pagination

When retrieving a collection the API will return a paginated response. The pagination information is made available in the `X-Pagination` header containing four values separated by semicolons. These four values respectively correspond to the number of items per page, the current page number (starting at 1), the number of pages and the total number of elements in the collection.

For instance, the header `X-Pagination: 30;1;3;84` has the following meaning:

- `30` : There are 30 items per page
- `1` : The current page is the first one
- `3` : There are 3 pages in total
- `84` : There is a total of 84 items in the collection

To iterate through the list, you need to use the `page` and `pageSize` query parameters when doing a `GET` request on a collection. If you do not specify those parameters, the default values of 1 (for `page`) and 30 (for `pageSize`) will be assumed.

Example: The request `GET /myResources?page=2&pageSize=5 HTTP/1.1` would produce a response comparable to the following:

```
HTTP/1.1 200 OK
X-Pagination: 5;2;7;35
...

{
  [
    { "id": 6 },
    { "id": 7 },
    { "id": 8 },
    { "id": 9 },
    { "id": 10 }
  ]
}
```

Sorting and Filtering

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

- Most REST APIs provide a mechanism to sort and filter collections.
- Think about GETting the list of all students who have a last name starting with a ‘B’, or all students who have an average grade above a certain threshold.
- Think about GETting the list of all students, sorted by rank or by age.
- The standard way to specify the sorting and filtering criteria is to use query string parameters.
- IMPORTANT: be consistent across your resources. The developer of client applications should be able to use the same mechanism (same parameter names and conventions) for all resources in your API!

Authentication

- In most cases, REST APIs are invoked over a secure channel (**HTTPS**).
- For that reason, the **basic authentication scheme** is often considered acceptable.
- Every request contains an “Authorization” header that contains either **user credentials** (user id + password) or some kind of **access token** previously obtained by the user.
- When the server receives an HTTP request, it extracts the credentials from the HTTP header, validates them against what is stored in the database and either grants/rejects the access.

OAuth

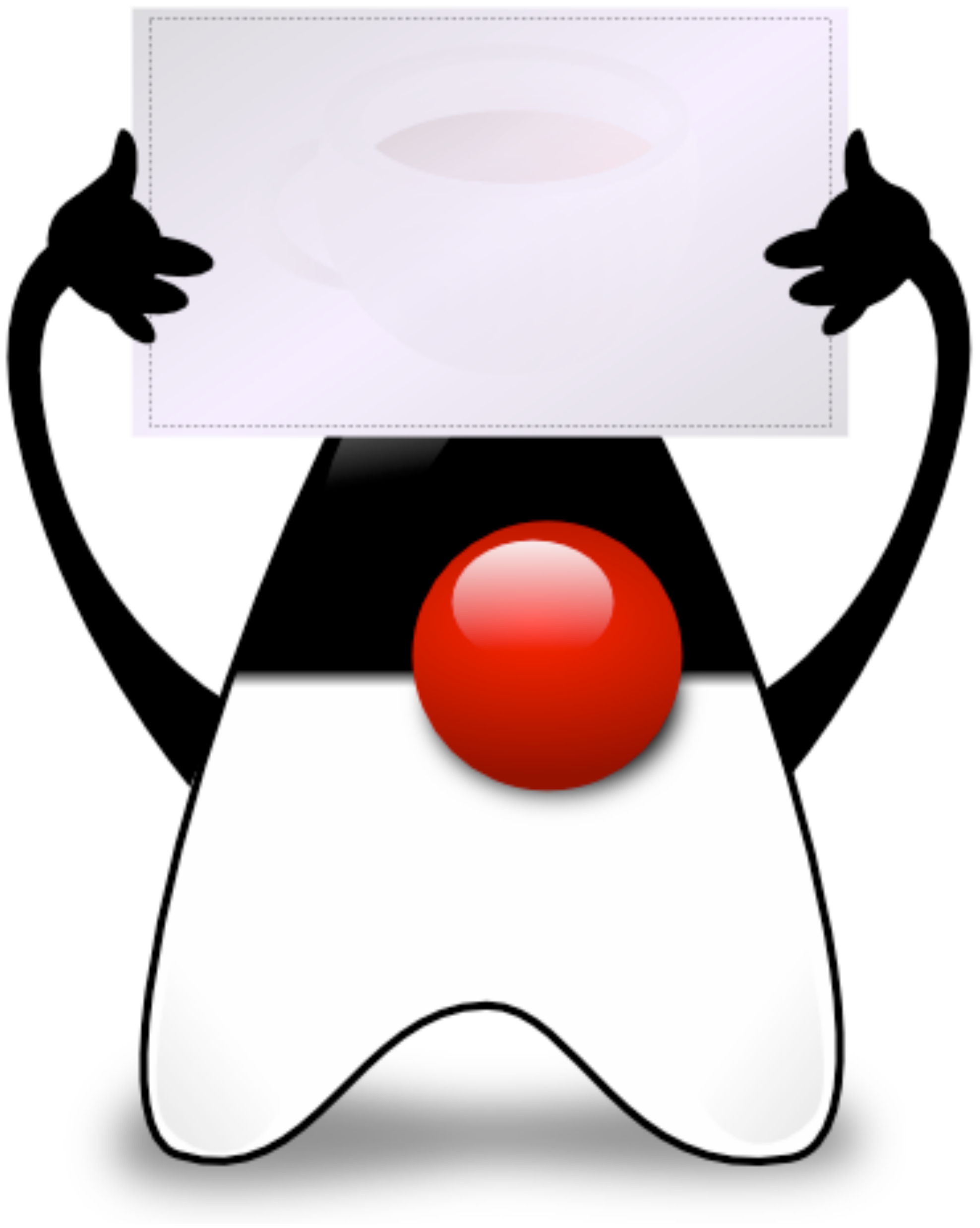
heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- In many REST APIs, OAuth 2.0 is used for **authorization** and **access delegation**:
    - when you use a **Facebook Application** (e.g. a game), you are asked whether you agree to **authorize** this third-party Application to access some of **your Facebook data** (and actions, such as posting to your wall).
    - If you agree, the Facebook Application receives a **bearer token**. When it sends HTTP requests to the **Facebook API**, it sends this token in a HTTP header (typically in the Authorization header). Because the Application has a valid token, Facebook grants access to your data.
    - In other words, using OAuth is similar to handing your car keys to a concierge.

API versioning

- If you think about the **medium and long term evolution of your service** (think about Twitter), your API is very likely to evolve over time:
  - You may add new types of resources
  - You may add/remove query string parameters
  - You may change the structure of the payloads
  - You may introduce new mechanisms (authentication, pagination, etc.)
- When you introduce a change in your API (and in the corresponding documentation), you will have a **compatibility issue**. Namely, you will have to support **some clients that still use the old version** of the API and **others that use the new version of the API**.
- For this reason, when you receive an HTTP request, you need to know which version is used by the client talking to you. As usual, there are different ways to pass this information (path element, query string parameter, header).
- A lot of REST APIs include the version number in the path, e.g. `http://www.myservice.com/api/v2/students/7883`

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



REST API Documentation

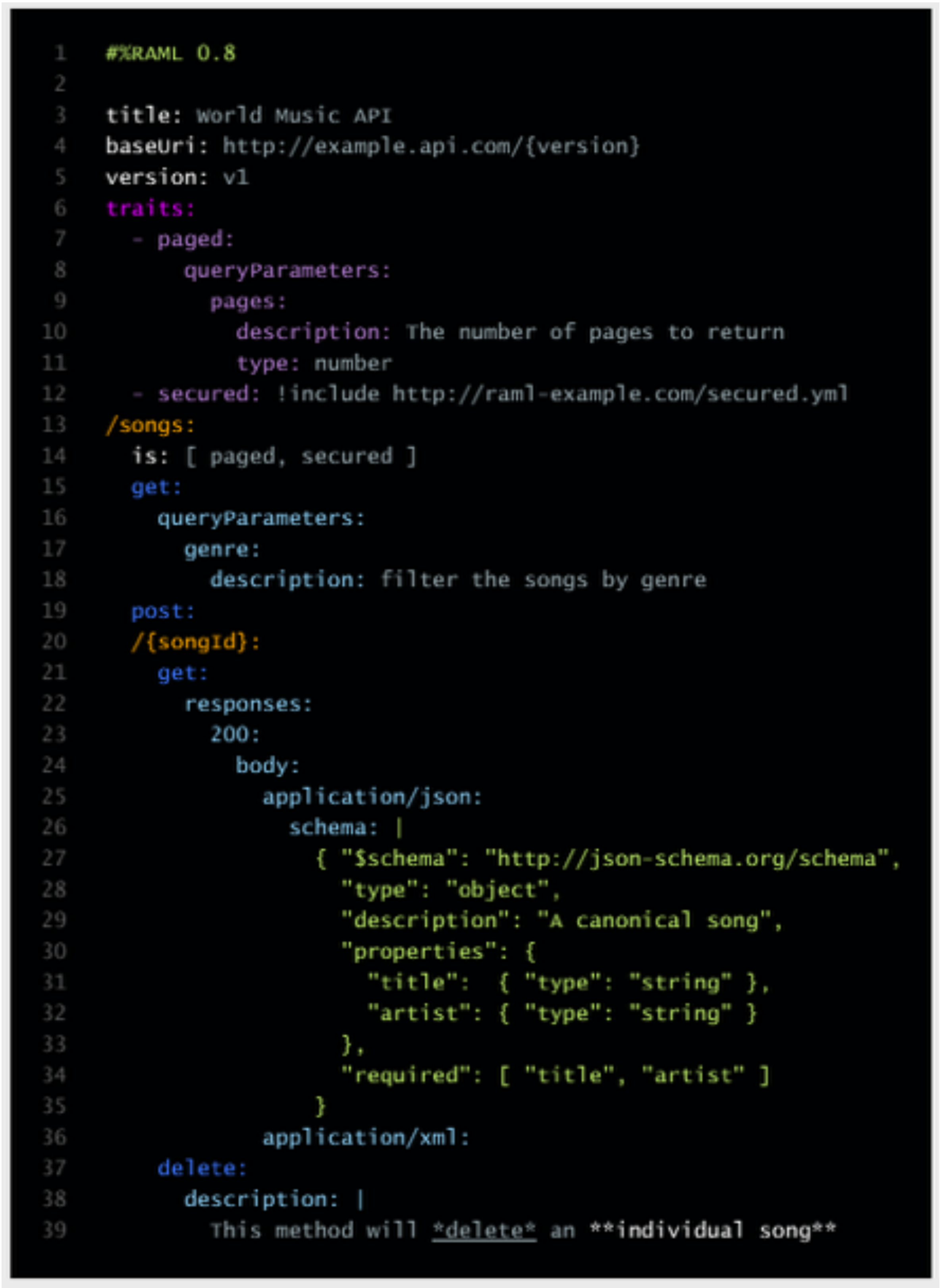
API Documentation

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

- When you are designing and implementing a REST API, you are most often doing it for **third-party developers**:
    - Think about Twitter, Instagram or Amazon exposing services to external developers.
    - Think about an enterprise (e.g. car manufacturer) exposing services to business partners (e.g. suppliers, subcontractors, distributors).
- The documentation of your API is the first thing that third-party developers (your **customers**) will see. You want to **seduce** them.
- The documentation of your API will have a big impact on its **learnability** and **ease of use**.
- **Best practices** and **tools** have emerged. **Evaluate and apply them!**

RAML

- **RE**STful **A**PI **M**odeling **L**anguage
- RAML is a language that has been developed to facilitate the **design** and **documentation** of REST APIs.
- It allows you to describe **resources**, **methods**, **parameters**, **headers** and **payloads** in a succinct manner (support for abstraction and reuse).
- From a RAML file, it is possible to **generate a user-friendly documentation** (e.g. in HTML) with various **tools**.
- Other tools support import/export exchange with REST frameworks (e.g. JAX-RS).



```
#%RAML 0.8

title: World Music API
baseUri: http://example.api.com/{version}
version: v1
traits:
  - paged:
      queryParameters:
        pages:
          description: The number of pages to return
          type: number
  - secured: !include http://raml-example.com/secured.yml
/songs:
  is: [ paged, secured ]
  get:
    queryParameters:
      genre:
        description: filter the songs by genre
  post:
  /{songId}:
    get:
      responses:
        200:
          body:
            application/json:
              schema: |
                { "$schema": "http://json-schema.org/schema",
                  "type": "object",
                  "description": "A canonical song",
                  "properties": {
                    "title":  { "type": "string" },
                    "artist": { "type": "string" }
                  },
                  "required": [ "title", "artist" ]
                }
            application/xml:
    delete:
      description: |
        This method will *delete* an **individual song**
```

RAML

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

- RAML is pretty straightforward to use:
  - You describe the list of resources managed by your application, document the support HTTP verbs, enlist the query parameters, etc.
  - If you use **Sublime Text**, you can take advantage of an extension that provides **syntax highlighting**.
  - See **RAML 100 tutorial** (http://raml.org/docs.html)
- RAML has advanced features that can make your specifications less verbose (by abstracting and reusing common elements):
  - includes
  - resource types and schemas
  - traits
  - See **RAML 200 tutorial** (http://raml.org/docs-200.html)

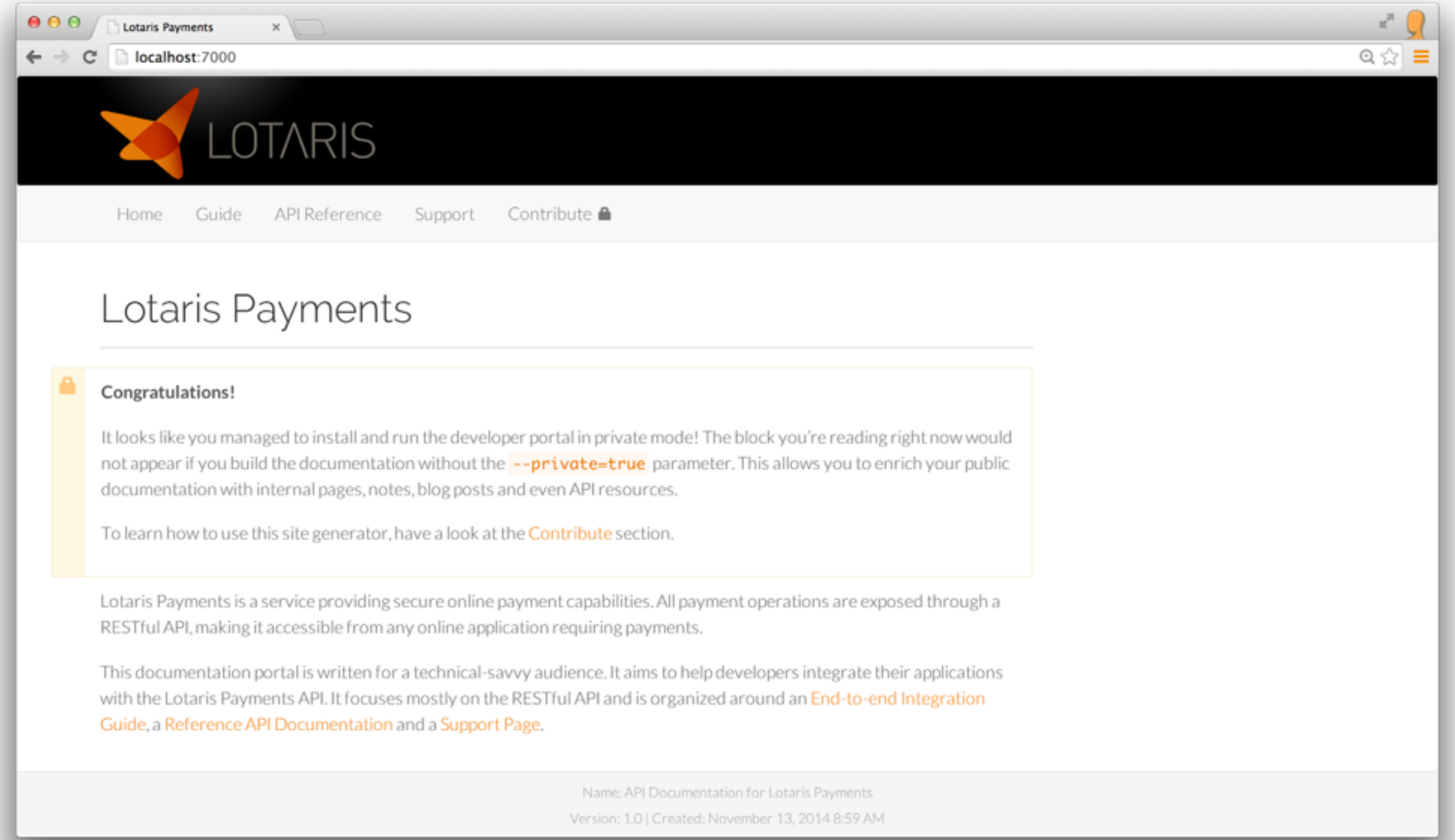
apidoc-seed

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

- **apidoc-seed** is an open source tool that is provided by Lotaris, which makes it easy to generate a complete HTML site for documenting your REST API.
- To use the tool, **clone the GitHub repo** (https://github.com/lotaris/apidoc-seed):
  - install **node.js** and **grunt** (`npm install -g grunt-cli`)
  - modify the **directory structure** to add/remove items in the main menu
  - **edit/create jade templates and markdown documents** to provide documentation for your service (general service information, usage guides, support information, etc.).
  - **edit/create RAML files to document your REST APIs**. Depending on the complexity of your APIs, you can **split** the documentation into several files.
  - follow instructions in the README.md file and generate the documentation site.
- The tool supports a notion of "**private**" API elements. This is used if your API has resources, methods or parameters that you don't want to publicly expose yet (note that this is documentation level only, nothing will prevent a user to send a request!).

apidoc-seed

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



Lotaris Payments

Congratulations!

It looks like you managed to install and run the developer portal in private mode! The block you're reading right now would not appear if you build the documentation without the `--private=true` parameter. This allows you to enrich your public documentation with internal pages, notes, blog posts and even API resources.

To learn how to use this site generator, have a look at the Contribute section.

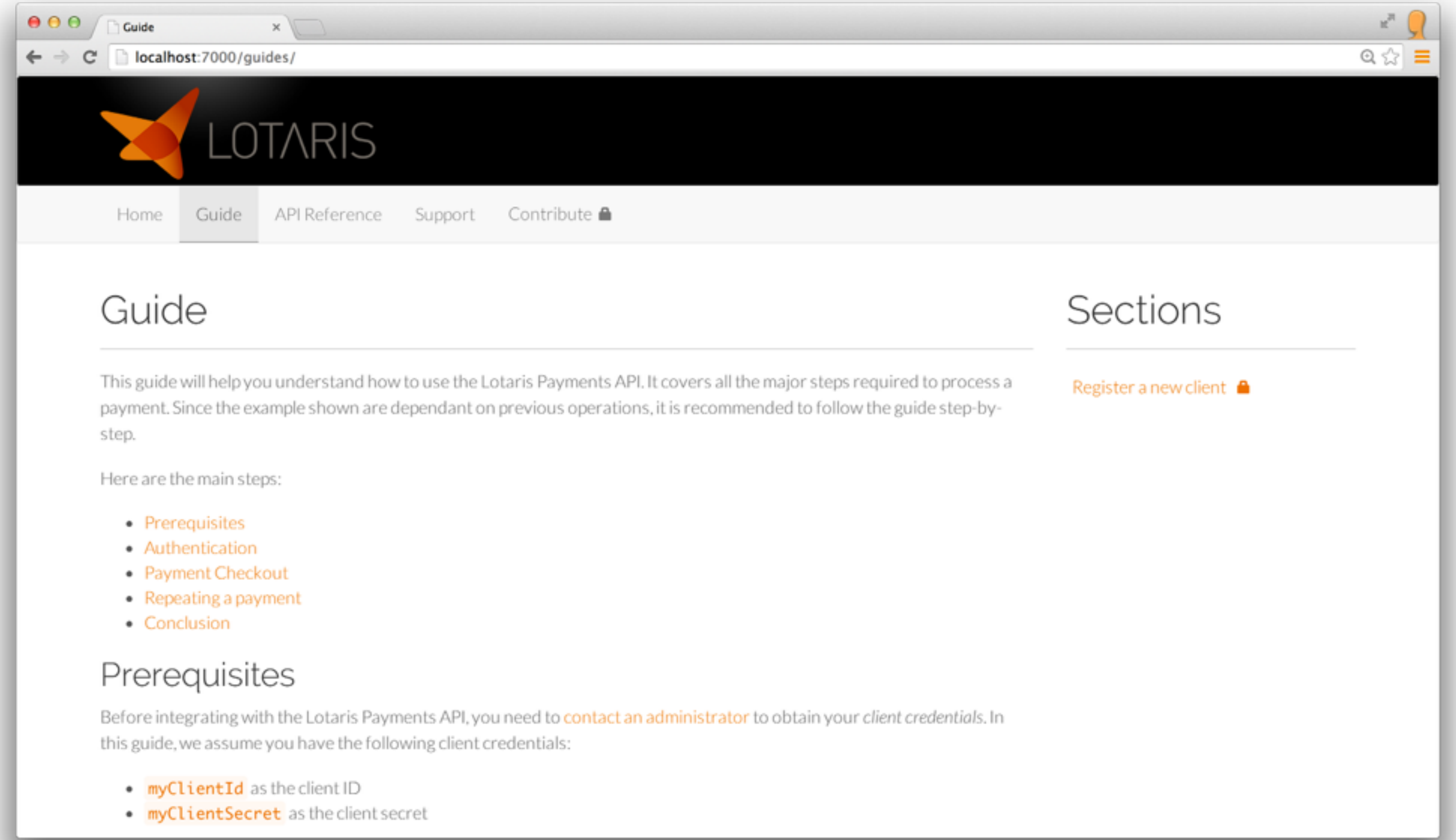
Lotaris Payments is a service providing secure online payment capabilities. All payment operations are exposed through a RESTful API, making it accessible from any online application requiring payments.

This documentation portal is written for a technical-savvy audience. It aims to help developers integrate their applications with the Lotaris Payments API. It focuses mostly on the RESTful API and is organized around an End-to-end Integration Guide, a Reference API Documentation and a Support Page.

Name: API Documentation for Lotaris Payments
Version: 1.0 | Created: November 13, 2014 8:59 AM

apidoc-seed

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



Guide

This guide will help you understand how to use the Lotaris Payments API. It covers all the major steps required to process a payment. Since the example shown are dependant on previous operations, it is recommended to follow the guide step-by-step.

Here are the main steps:

- Prerequisites
- Authentication
- Payment Checkout
- Repeating a payment
- Conclusion

Prerequisites

Before integrating with the Lotaris Payments API, you need to contact an administrator to obtain your *client credentials*. In this guide, we assume you have the following client credentials:

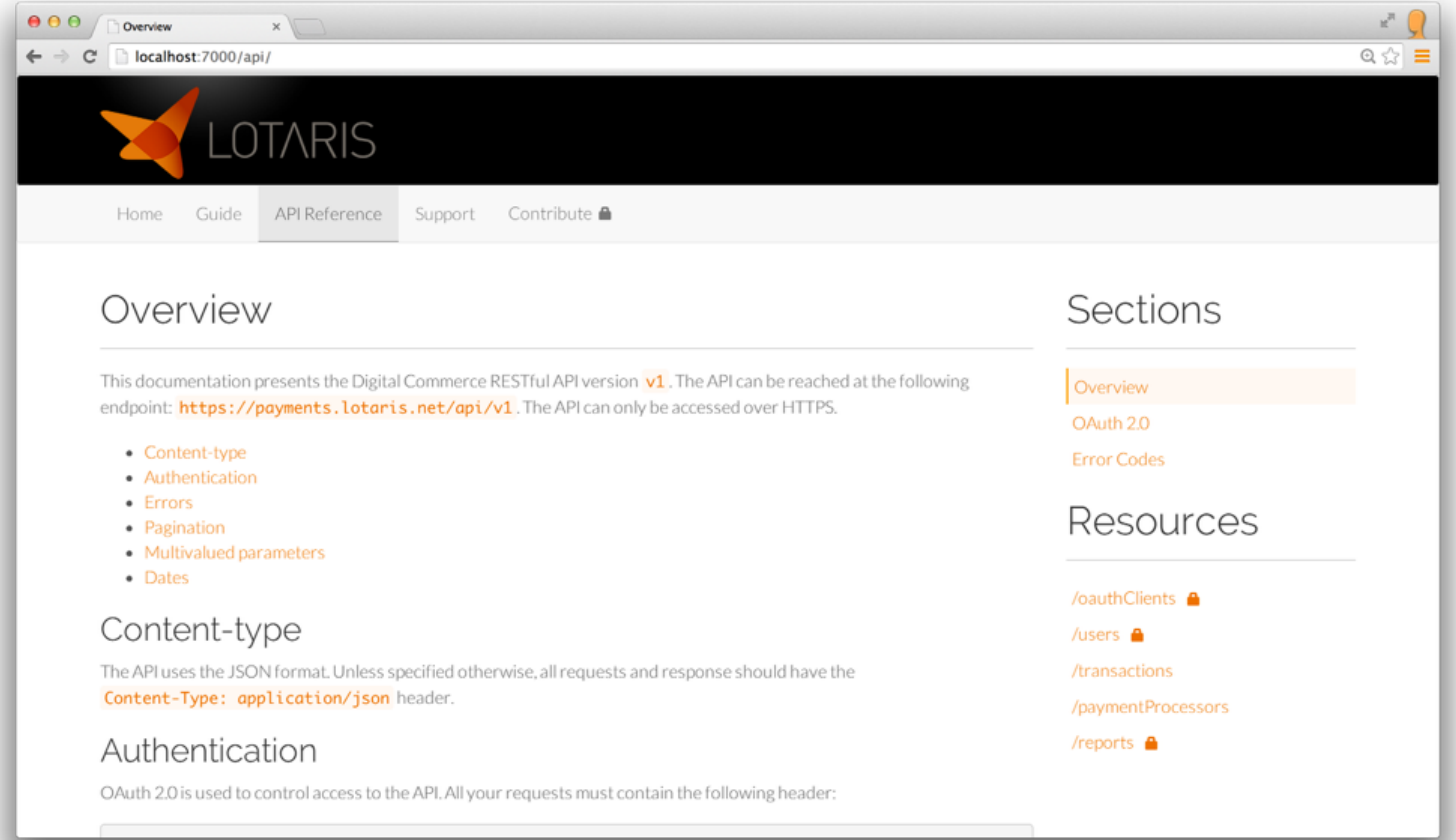
- `myClientId` as the client ID
- `myClientSecret` as the client secret

Sections

Register a new client

apidoc-seed

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



Overview

This documentation presents the Digital Commerce RESTful API version `v1`. The API can be reached at the following endpoint: `https://payments.lotaris.net/api/v1`. The API can only be accessed over HTTPS.

- Content-type
- Authentication
- Errors
- Pagination
- Multivalued parameters
- Dates

Content-type

The API uses the JSON format. Unless specified otherwise, all requests and response should have the `Content-Type: application/json` header.

Authentication

OAuth 2.0 is used to control access to the API. All your requests must contain the following header:

Sections

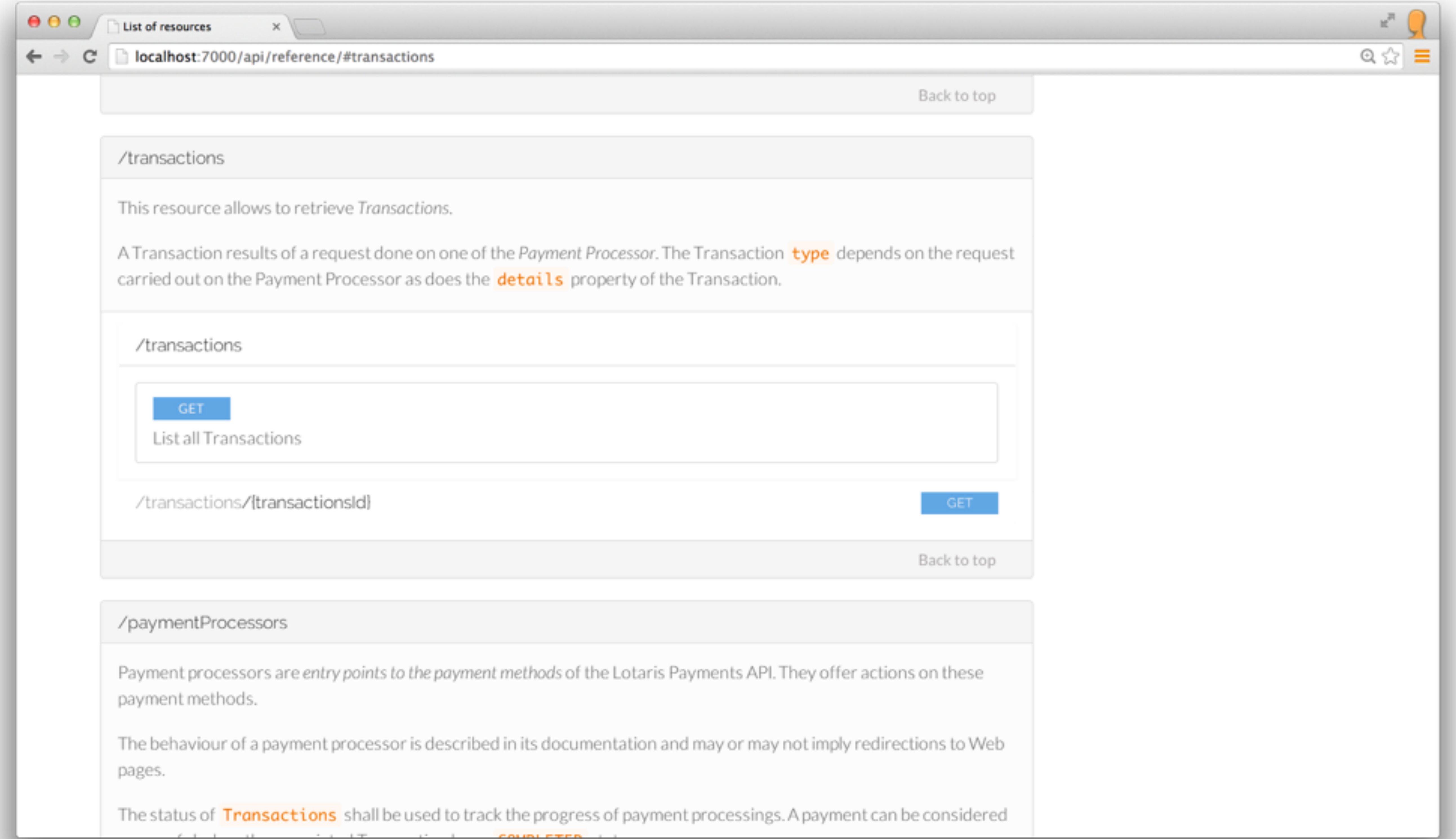
- Overview
- OAuth 2.0
- Error Codes

Resources

- /oauthClients
- /users
- /transactions
- /paymentProcessors
- /reports

apidoc-seed

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



Back to top

/transactions

This resource allows to retrieve *Transactions*.

A Transaction results of a request done on one of the *Payment Processor*. The Transaction `type` depends on the request carried out on the Payment Processor as does the `details` property of the Transaction.

/transactions

GET
List all Transactions

/transactions/{transactionsId} GET

Back to top

/paymentProcessors

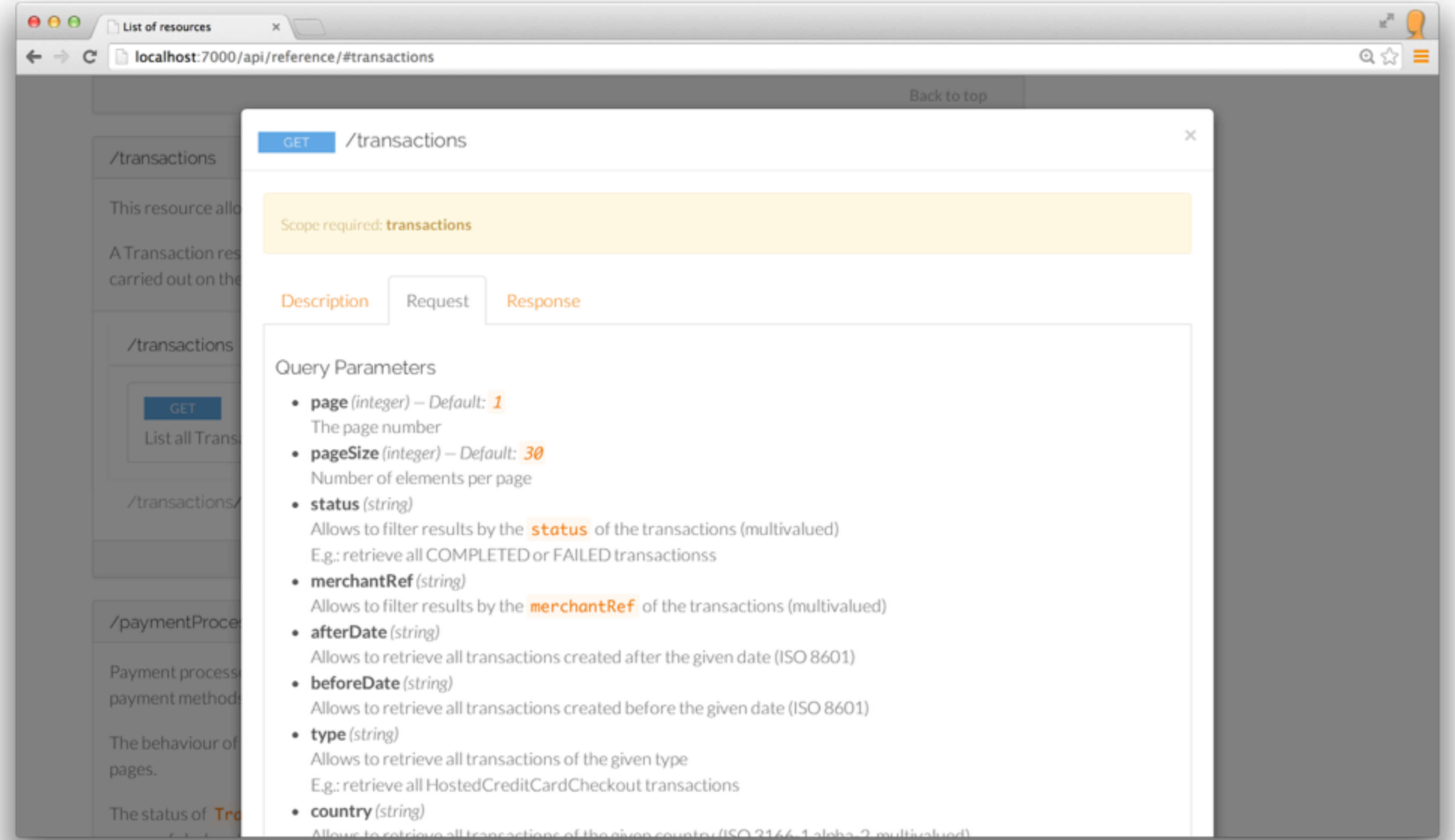
Payment processors are *entry points to the payment methods* of the Lotaris Payments API. They offer actions on these payment methods.

The behaviour of a payment processor is described in its documentation and may or may not imply redirections to Web pages.

The status of `Transactions` shall be used to track the progress of payment processings. A payment can be considered

apidoc-seed

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



GET /transactions

Scope required: **transactions**

Description | Request | Response

Query Parameters

- **page** *(integer) — Default:* *1*
  The page number
- **pageSize** *(integer) — Default:* *30*
  Number of elements per page
- **status** *(string)*
  Allows to filter results by the `status` of the transactions (multivalued)
  E.g.: retrieve all COMPLETED or FAILED transactionss
- **merchantRef** *(string)*
  Allows to filter results by the `merchantRef` of the transactions (multivalued)
- **afterDate** *(string)*
  Allows to retrieve all transactions created after the given date (ISO 8601)
- **beforeDate** *(string)*
  Allows to retrieve all transactions created before the given date (ISO 8601)
- **type** *(string)*
  Allows to retrieve all transactions of the given type
  E.g.: retrieve all HostedCreditCardCheckout transactions
- **country** *(string)*
  Allows to retrieve all transactions of the given country (ISO 3166-1 alpha-2, multivalued)

Individual work

- Clone the **apidoc-seed repo**, build and test it in your browser.
- Follow the instructions in the **RAML 100 tutorial** (http://raml.org/docs.html), but do it in the `/src/api/raml/index.raml` file (keep a copy of the original file).
- Modify the **Welcome page** in the documentation site (when you deliver your project, I expect to find a description of your platform here).
- Get rid of the **Blog** menu item (and related pages).
- Customize the **page footer**.
- Duplicate the API Reference menu element (and related pages). In our project, we will describe **several REST APIs**, exposed to different types of clients.