Lecture 4: RESTful APIs

Olivier Liechti
AMT

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

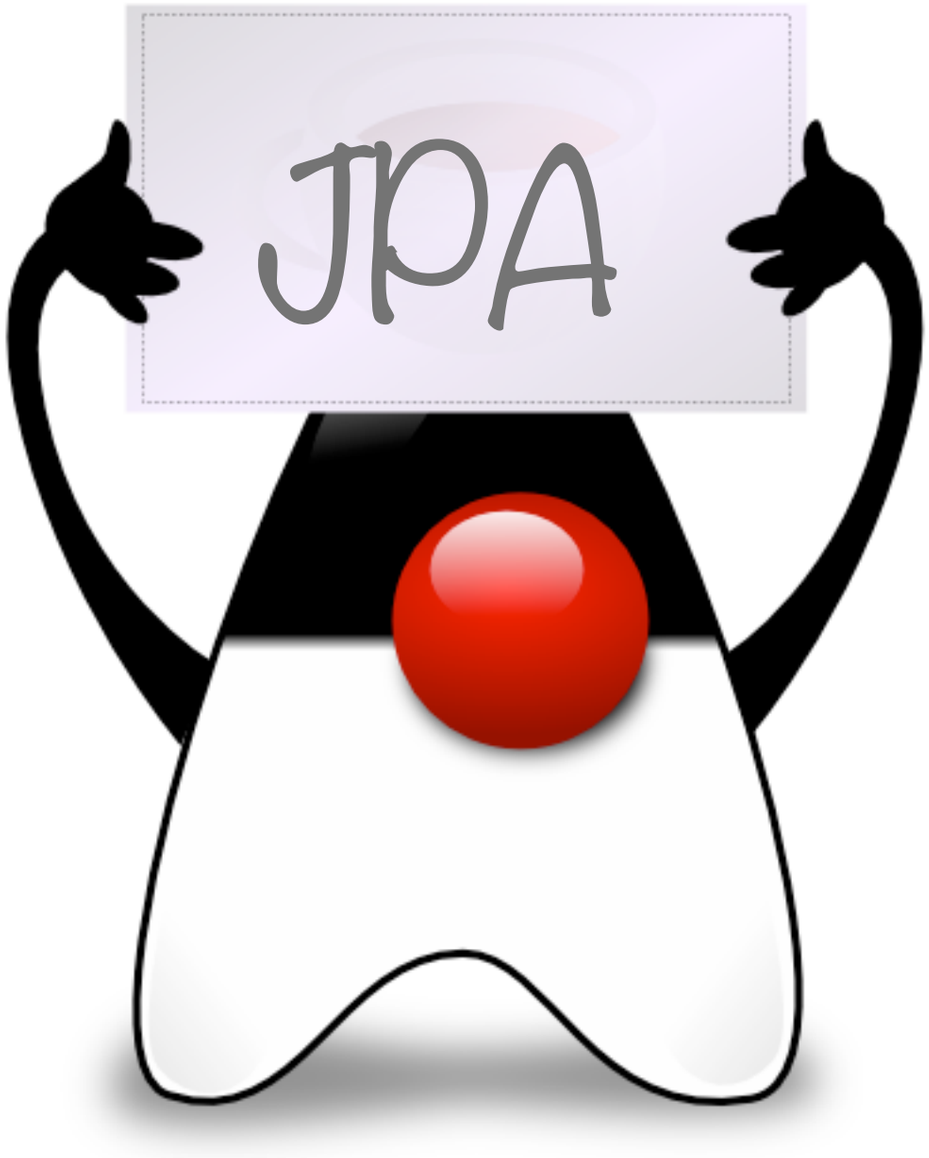
heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Today's agenda

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

14h00 - 15h00	60'	**Lecture** Back to JPA Java Reflection & JavaBeans RESTful APIs with JAX-RS
15h00 - 15h10	*10'*	*Break*
15h10 - 16h25	75'	**Lecture** REST API Design Issues JAX-RS in the MVCDemo project **Demo / Exercise** Testing the REST API with Jersey Client Probe Dock

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

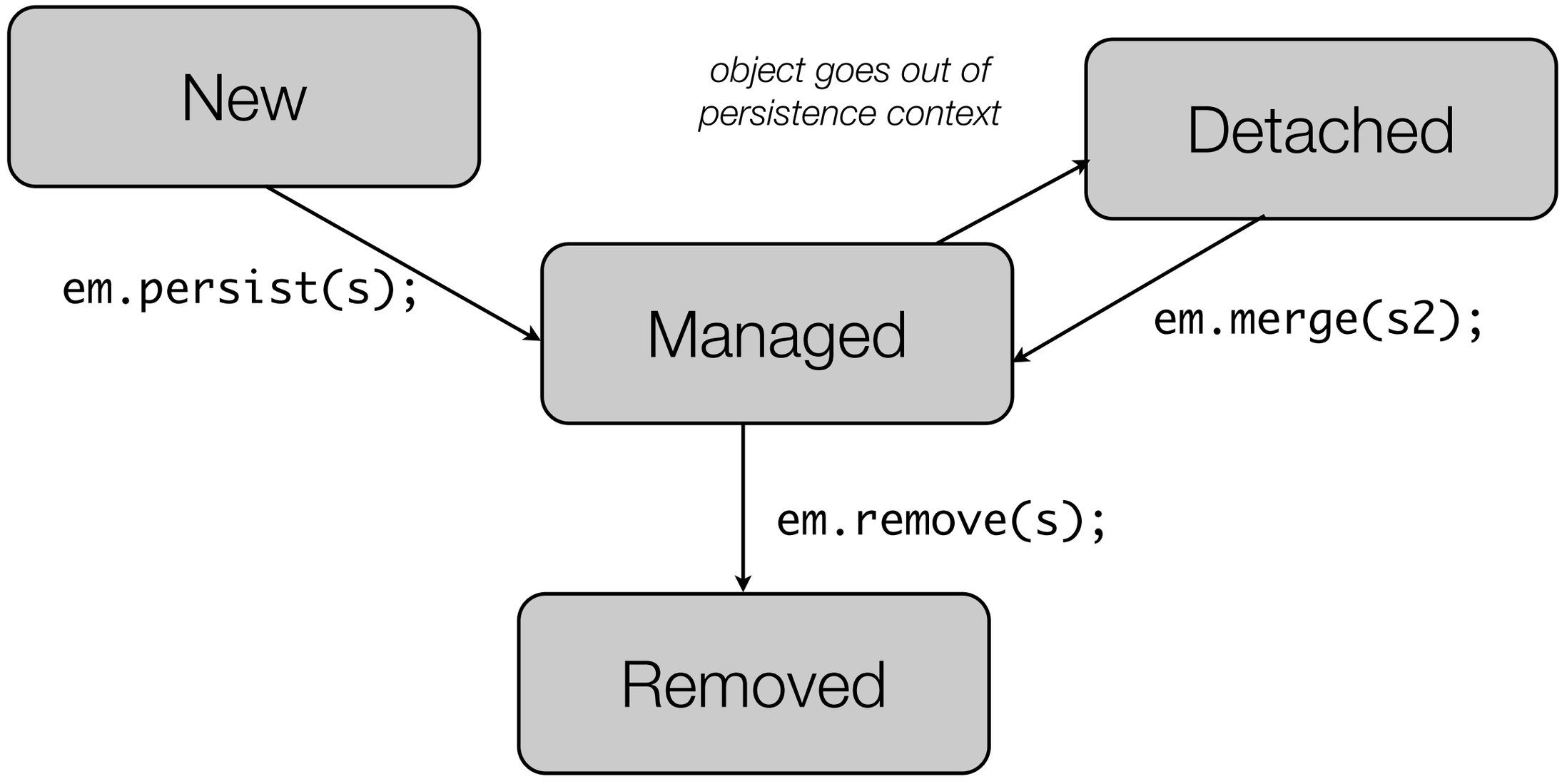


Java Persistence API (JPA)

Life-cycle for JPA Entities

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

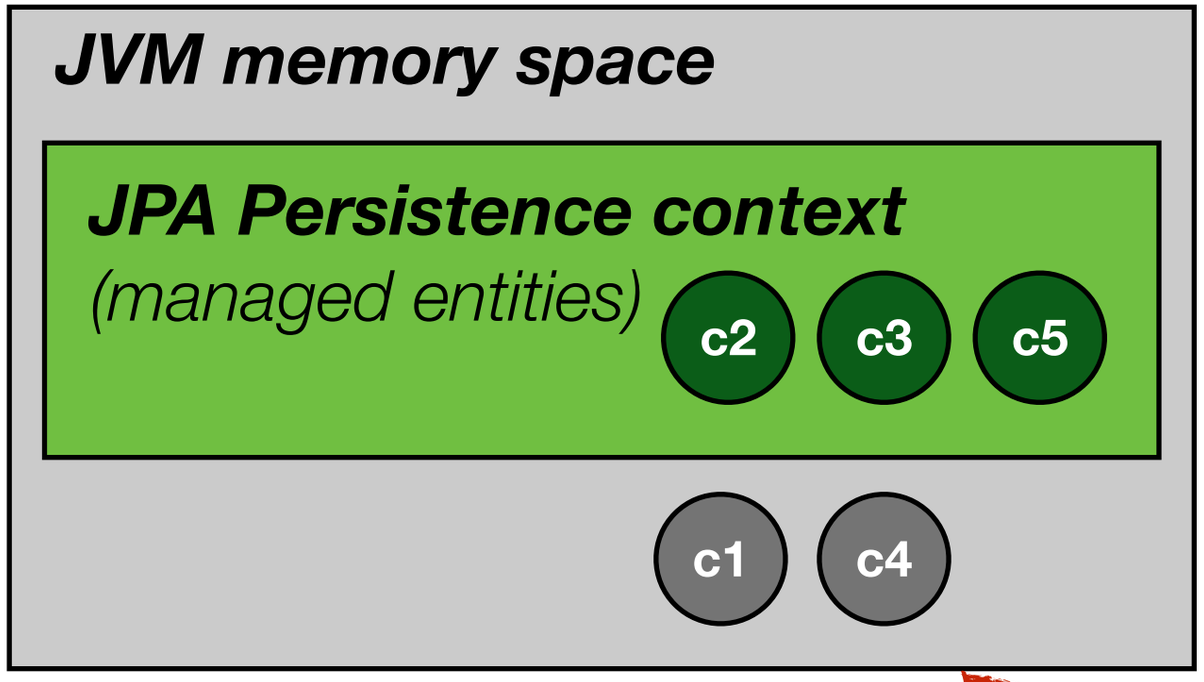
Think what happens when an EJB returns an object to a servlet!



When do objects enter and leave the persistence context?

The persistence context is **created** when **transaction** begins and is **flushed** when transaction commits (or rollbacks).

A **transaction** is started by the **EJB container** whenever a business method is called. It is committed by the container when it returns (or rollbacked if there is an exception).




```
@Stateless
public class Manager {

  @PersistenceContext
  EntityManager em;

  public void businessMethod() {
    Customer c1 = new Customer();

    Customer c2 = new Customer();
    em.persist(c2);

    Customer c3 = em.find(123);

    Customer c4 = new Customer(246, “john”, “doe”);
    Customer c5 = em.merge(c4);
  }
}
```

breakpoint

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

```
Customer c1 = new Customer();
```

Creating a new instance of a JPA entity does not make it a managed object. At this stage, it is **a simple POJO** that is not linked to the DB (*)

```
Customer c2 = new Customer();
// c2 is not in persistence ctx

em.persist(c2);
// c2 is in the persistence ctx
```

Calling **em.persist(c2) brings c2 into the persistence context**. From this point, JPA intercepts all calls made to c2. So, it knows when c2 is modified by a client (i.e. when it becomes "**dirty**").

Note that at this stage, it is most likely that **nothing has been written to the DB**. SQL statements will only be issued when the transaction commits.

```
Customer c3 = em.find(123);
```

Calling **em.find(123)** issues a SELECT query to the DB. An entity is created with the result and is **brought into the persistence context**.

```
Customer c4 = new
  Customer(246, "john", "doe");
Customer c5 = em.merge(c4);
```

c4 is a simple POJO. When **em.merge(c4)** is invoked, a SELECT statement will be issued to retrieve a row where the primary key is equal to 246. A new entity is created and its properties are copied from c4 (to update the DB later on). **WARNING: c5 is in the persistence context, c4 is not!!**

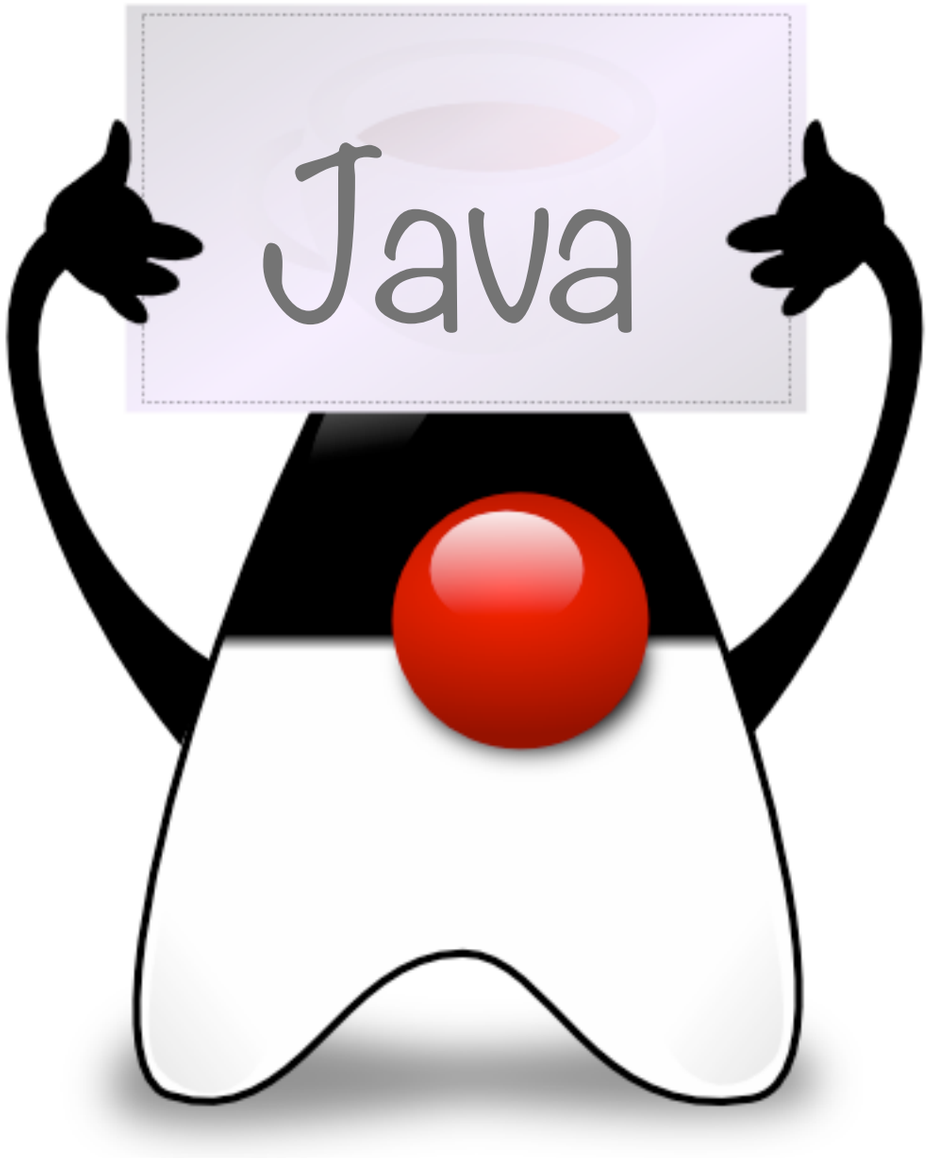
(*) Sort of... see a description of how the magic can happen: http://struberg.wordpress.com/2012/01/08/jpa-enhancement-done-right/

Persistence Context Types

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

- In Java EE, we typically use a **transaction-scoped persistence context**:
  - The client invokes a method on a **Stateless Session Bean**
  - The container intercepts a call and **starts a transaction**
  - The Stateless Session Bean uses JPA, a persistence context is created
    - Entities are loaded into the **persistence context**, modified, added, etc.
    - The method returns, the container **commits** the transaction
    - At this stage, entities in the persistence context are **sent back** to the DB.
- JPA also defines **extended persistence context**:
  - Entities remain managed as long as the Entity Manager lives
  - The JBoss SEAM framework uses extended persistence contexts: a persistence context lives during a whole “conversation”.

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



Java Reflection & JavaBeans

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

2 related questions / observations

Question 1: why do we write this **static block** and isn't it a **dirty hack**?

Client

```
Class.forName("ch.heigdb.HeigDbDriver");
DriverManager.getConnection("jdbc:heigdb://localhost:2205");
```

JDBC Service
`java.sql.DriverManager`

JDBC HeigDB driver

```
public class HeigDbDriver implements java.sql.Driver {

    static {
        DriverManager.registerDriver(new SomeDriver());
    }
    public boolean acceptsURL(String url) {};
    public Connection connect(String url, Properties p) {};
}
```

Why do we do that?

Why don't we write something like:

```
HeigDbDriver driver = new HeigDbDriver();
driver.init();
```

```
public class HeigDbDriver implements java.sql.Driver {

  public void init() {
    DriverManager.registerDriver(new SomeDriver());
  }
  public boolean acceptsURL(String url) {};
  public Connection connect(String url, Properties p) {};
}
```

Question 2: JDBC is pretty straightforward, but… isn't it **verbose and repetitive**?

```
@Stateless
public class SensorJdbcDAO implements SensorDAOLocal {

  @Resource(lookup = "jdbc/AMTDatabase")
  private DataSource dataSource;

 public List<Sensor> findAll() {
    List<Sensor> result = new LinkedList<>();
    try {
      Connection con = dataSource.getConnection();

    PreparedStatement ps = con.prepareStatement("SELECT * FROM Sensors");
    ResultSet rs = ps.executeQuery();

    while (rs.next()) {
      Sensor sensor = new Sensor();
      sensor.setId(rs.getLong("ID"));
      sensor.setDescription(rs.getString("DESCRIPTION"));
      sensor.setType(rs.getString("TYPE"));
      result.add(sensor);
    }

    ps.close();
    con.close();

  } catch (SQLException ex) {
    Logger.getLogger(SensorJdbcDAO.class.getName()).log(Level.SEVERE, null, ex);
  }
  return result;
}
```

When I implement the UserDAO, the RoleDAO, the LocationDAO, will I need to **repeat** all the code around those statements (**boilerplate**)?

Will I need to manually replace the **table and column names** in each DAO?

And when I **maintain** my application, what happens when a new property is added? Do I have to **update my DAO**?

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

As a matter of fact, these 2 questions are closely related!

Answering the 1st question will give you a solution for the 2nd!

Let's compare two options carefully:

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

Client

```
Class.forName("ch.heigdb.HeigDbDriver");
DriverManager.getConnection("jdbc:heigdb://localhost:2205");
```

`ch.heigdb.HeigDbDriver` is a **string**

This means that I can **dynamically load** JDBC drivers, **without changing the code** of the client.

I only need to have the drivers **.jar files** in my class path and to **configure** my client.

Why don't we write something like:

```
HeigDbDriver driver = new HeigDbDriver();
driver.init();
```

`HeigDbDriver` is a hard-coded **Java identifier**

This means that if I want to use another JDBC driver, then I need to **change the client code** and **recompile**.

`Class.forName(String name)` is part of the **Reflection API**, which allow us to write dynamic code in Java.

Edit JDBC Connection Pool

localhost:4848/common/index.jsf

Home | About... | Help

User: anonymous | **Domain:** domainAMT | **Server:** localhost

GlassFish™ Server Open Source Edition

Tree

- server (Admin Server)
- Clusters
- Standalone Instances
- Nodes
- Applications
- Lifecycle Modules
- Monitoring Data
- Resources
  - Concurrent Resources
  - Connectors
  - JDBC
    - JDBC Resources
      - jdbc/__TimerPool
      - jdbc/__default
      - jdbc/myDataSource
      - jdbc/sample
    - JDBC Connection Pools
      - DerbyPool
      - SamplePool
      - __TimerPool
      - mysql_mysql_rootPool
  - JMS Resources
  - JNDI
  - JavaMail Sessions
  - Resource Adapter Configs
- Configurations

General | **Advanced** | **Additional Properties**

Edit JDBC Connection Pool Properties

Save Cancel

Modify properties of an existing JDBC connection pool.

Pool Name: mysql_mysql_rootPool

Additional Properties (7)

Add Property | Delete Properties

Select	Name	Value	Description
	URL	jdbc:mysql://localhost:3306/mysql?zeroDateTin	
	driverClass	com.mysql.jdbc.Driver	
	Password	akAUKLJdf!!882_2	
	portNumber	3306	
	databaseName	mysql	
	User	technicalAccount	
	serverName	localhost	

Through this interface, I am **configuring** Glassfish and asking him to do a:

```
Class.forName("com.mysql.jdbc.Driver");
```

I only need to make sure that the mysql .jar files is in Glassfish's **classpath**

```
@Stateless
public class SensorJdbcDAO implements SensorDAOLocal {

  @Resource(lookup = "jdbc/AMTDatabase")
  private DataSource dataSource;

 public List<Sensor> findAll() {
    List<Sensor> result = new LinkedList<>();
    try {
      Connection con = dataSource.getConnection();

      PreparedStatement ps = con.prepareStatement("SELECT * FROM Sensors");
      ResultSet rs = ps.executeQuery();

      while (rs.next()) {
        Sensor sensor = new Sensor();
        sensor.setId(rs.getLong("ID"));
        sensor.setDescription(rs.getString("DESCRIPTION"));
        sensor.setType(rs.getString("TYPE"));
        result.add(sensor);
      }

      ps.close();
      con.close();

    } catch (SQLException ex) {
      Logger.getLogger(SensorJdbcDAO.class.getName()).log(Level.SEVERE, null, ex);
    }
    return result;
  }
```

Reflection sounds cool. Can't we use it to deal with JDBC in **more generic** ways?

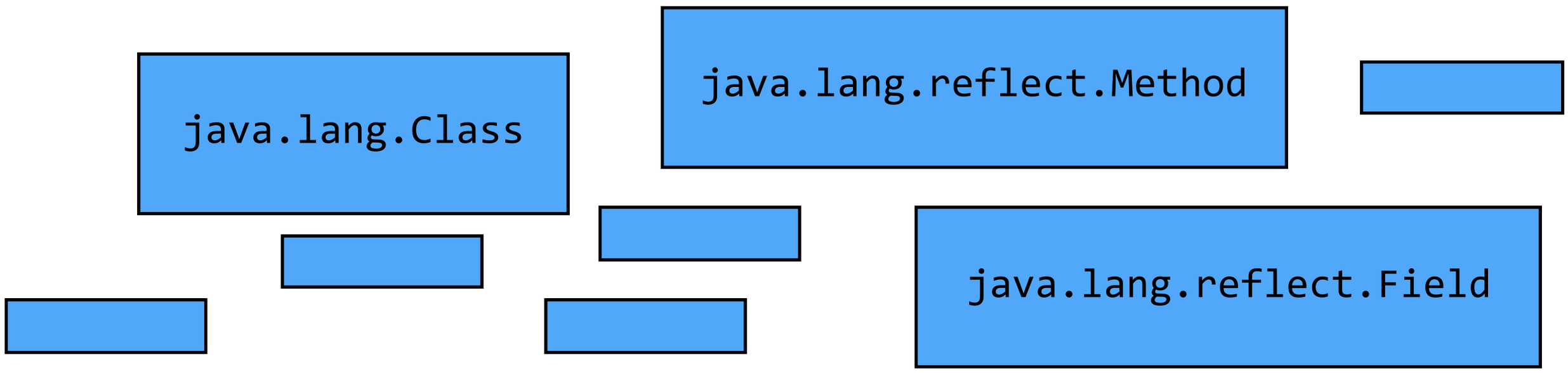
JDBC gives me **metadata** about the DB schema.

Reflection gives me ways to dynamically find and **invoke methods** on Java objects.

Can we combine these features to make this code better?

What is the Java **Reflection** API?

- Reflection is a mechanism, through which a program can **inspect and manipulate** its structure and behavior **at runtime.**
- In Java, this means that a program can **get information about classes, their fields, their methods, etc.**
- In Java, this also means that a program can **create instances of classes dynamically** (based on their names, as in the example of JDBC drivers), **invoke methods**, etc.



Can you give me an example of **reflective code**?

- We can **load class definitions** and **create instances**, without hard-coding class names into Java identifiers:

```
Class dynamicManagerClass = Class.forName("ch.heigvd.amt.reflection.services.SensorsManager");
Object dynamicManager = dynamicManagerClass.newInstance();
```

- For a class, we can **get the list of methods** and **their signature**:

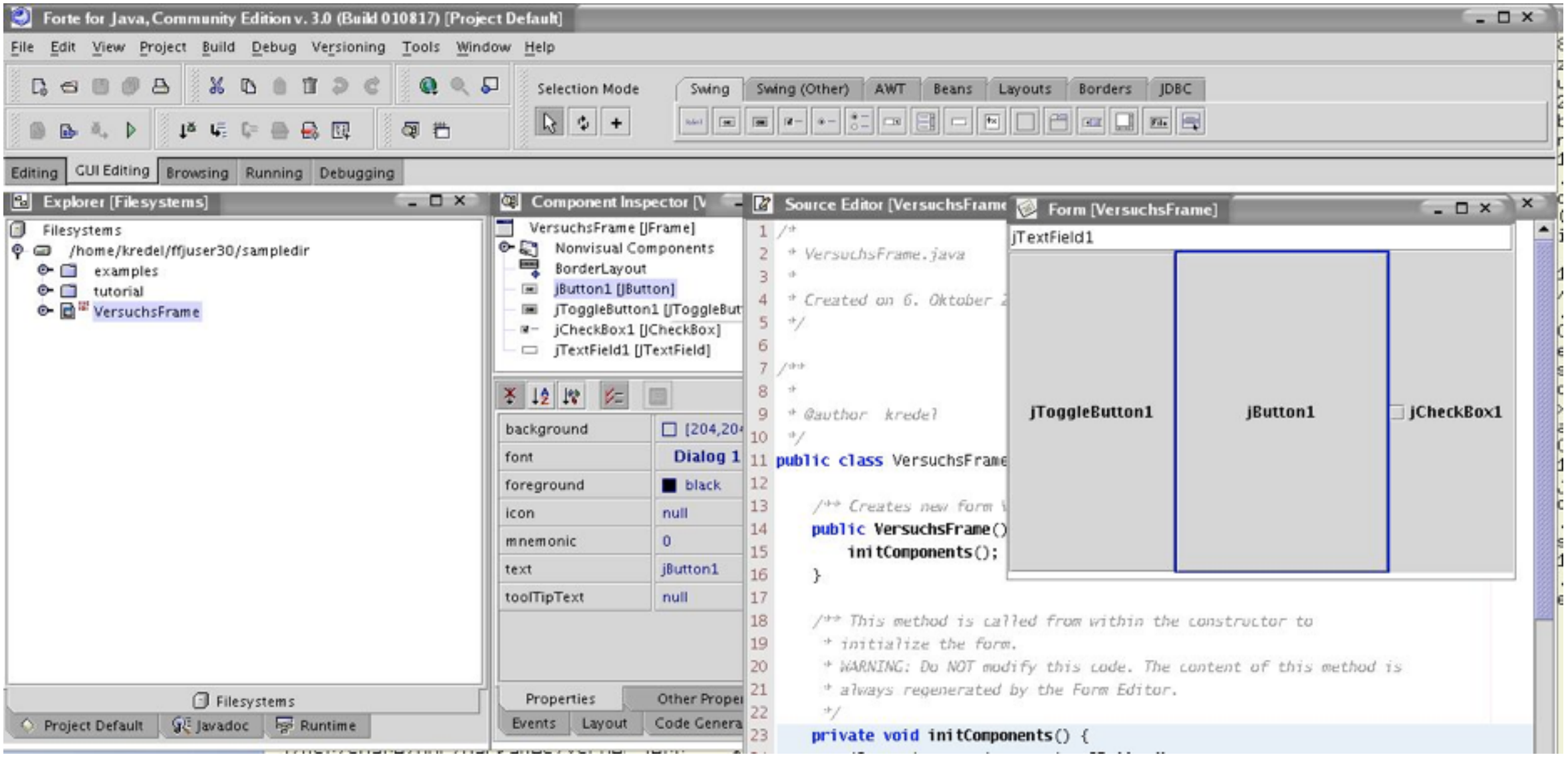
```
Method[] methods = dynamicManagerClass.getMethods();

for (Method method : methods) {
  LOG.log(Level.INFO, "Method name: " + method.getName());

  Parameter[] parameters = method.getParameters();
  for (Parameter p : parameters) {
    LOG.log(Level.INFO, "p.getName()+ ":" + p.getType().getCanonicalName());
  }
}
```

- We can dynamically **invoke a method** on an object:

```
Method method = dynamicManagerClass.getMethod("generateSensors", int.class, String.class);
Object result = method.invoke(dynamicManager, 5, "hello");
```

What are **JavaBeans**?

- First of all, JavaBeans are **NOT** Enterprise Java Beans.
- The JavaBeans specification was proposed a very long time ago (1997) to enable the creation of **reusable components in Java**.
- One of the first use cases was to support the creation of **WYSIWYG development tools**. The programmer could drag and drop a GUI widget from a palette onto a window and edit its properties in a visual editor (think Visual Basic for Java).
- In this scenario, the GUI widgets would be packaged as JavaBeans by **third-party vendors**. The development tool would recognize them as such and would **dynamically extend the palette** of available components.



Forte for Java (aka Netbeans grand-father)

What are **JavaBeans**?

- Since then, JavaBeans have become **pervasive** in the Java Platform and are **used in many other scenarios**.
- This is particularly true in the Java EE Platform. Actually, **you have already implemented** JavaBeans without realizing it.
- While there are other aspects in the specification, the key elements are **coding conventions** that JavaBeans creators should respect:

  1. A JavaBean should have a **public no-args constructor**.
  2. A JavaBean should expose its properties via **getter** and **setter methods** with **well-defined names**.
  3. A JavaBean should be **serializable**.

```java
public class Customer implements Serializable {

  public Customer() {}

  private String firstName;
  private String lastName;
  private boolean goodCustomer;

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public boolean isGoodCustomer() {
    return goodCustomer;
  }

  public void setGoodCustomer(boolean goodCustomer) {
    this.goodCustomer = goodCustomer;
  }

}
```

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

There is a **specific convention** for writing getter methods for **boolean properties**.

What are **JavaBeans**?

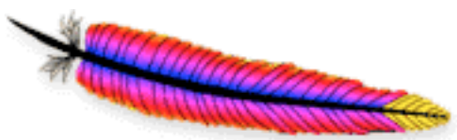
- These **coding and naming conventions** make it easier to **benefit from reflection** in **Java frameworks**:

    1. The framework can use the **public no-args constructor** to **create instances** with `Class.newInstance()`.
    2. The framework can **easily find out which methods it should call** (via reflection), based on a textual name. For instance, when a JSP page includes the string `${sensor.type}`, the runtime knows that it must invoke a method named `"get" + "Type"`.
    3. The **state of a JavaBean** can travel over the wire (for instance when it moves from a remote EJB container to a web container).

What should I be know if I plan **to implement a framework** with JavaBeans?

- With the naming conventions defined in the JavaBeans specification, combined with Java reflection, **you can do pretty much everything yourself**.
- Have a look at the `java.beans` package and at the `Introspector` class. You will have easy access to properties, getters and setters.
- You should be aware of the **Apache Commons BeanUtils** library that will make your life easier.

*"The Java language provides **Reflection** and **Introspection** APIs (see the java.lang.reflect and java.beans packages in the JDK Javadocs). However, **these APIs can be quite complex** to understand and utilize. The BeanUtils component provides **easy-to-use wrappers** around these capabilities."*



http://commons.apache.org/proper/commons-beanutils/

commons beanutils™

Back to the original question... How can I use reflection to **make my JDBC code generic**?

```
@Stateless
public class SensorJdbcDAO implements SensorDAOLocal {

  @Resource(lookup = "jdbc/AMTDatabase")
  private DataSource dataSource;

 public List<Sensor> findAll() {
    List<Sensor> result = new LinkedList<>();
    try {
      Connection con = dataSource.getConnection();

      PreparedStatement ps = con.prepareStatement("SELECT * FROM Sensors");
      ResultSet rs = ps.executeQuery();

      while (rs.next()) {
        Sensor sensor = new Sensor();
        sensor.setId(rs.getLong("ID"));
        sensor.setDescription(rs.getString("DESCRIPTION"));
        sensor.setType(rs.getString("TYPE"));
        result.add(sensor);
      }

      ps.close();
      con.close();

    } catch (SQLException ex) {
      Logger.getLogger(SensorJdbcDAO.class.getName()).log(Level.SEVERE, null, ex);
    }
    return result;
  }
```

Reflection sounds cool. Can't we use it to deal with JDBC in **more generic** ways?

JDBC gives me **metadata** about the DB schema.

Reflection gives me ways to dynamically find and **invoke methods** on Java objects.

Can we combine these features to make this code better?

Back to the original question... How can I use reflection to **make my JDBC code generic**?

```
Sensor sensor = new Sensor();
sensor.setId(rs.getLong("ID"));
sensor.setDescription(rs.getString("DESCRIPTION"));
sensor.setType(rs.getString("TYPE"));
result.add(sensor);
```

Object-Relational Mapping in this example:
Table name = Class name + “s”
Column name = property name

Class names, property names, table names and column names do not have to be hard-coded.

What we need is a **mapping**. We can either rely on **conventions** or define it **explicitly**.

```
String entityName = "Sensor";
String className = "ch.heigvd.amt.lab1.model." + entityName;
String tableName = entityName + "s";
PreparedStatement ps = con.prepareStatement("SELECT * FROM " + tableName);
ResultSet rs = ps.executeQuery();
Class entityClass = Class.forName(className);
PropertyDescriptor[] properties =
Introspector.getBeanInfo(entityClass).getPropertyDescriptors();

while (rs.next()) {
  Object entity;
  entity = entityClass.newInstance();
  for (PropertyDescriptor property : properties) {
    Method method = property.getWriteMethod();
    String columnName = property.getName();
    try {
      method.invoke(entity, rs.getObject(columnName));
    } catch (SQLException e) {
      LOG.warning("Could not retrieve value for property " + property.getName()
          + " in result set. " + e.getMessage());
    }
  }
  result.add(entity);
}
```

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

These mechanisms are used by people who build Object Relational Mapping (ORM) frameworks (like JPA)

heig-vd

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

RESTful APIs

The REST Architectural Style

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

- 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.

Principles of a REST Architecture

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

- 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

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

Resource vs. Representation

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

- 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?

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

- 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
  - **PATCH**: used for partial updates

How should I design and specify my REST API?

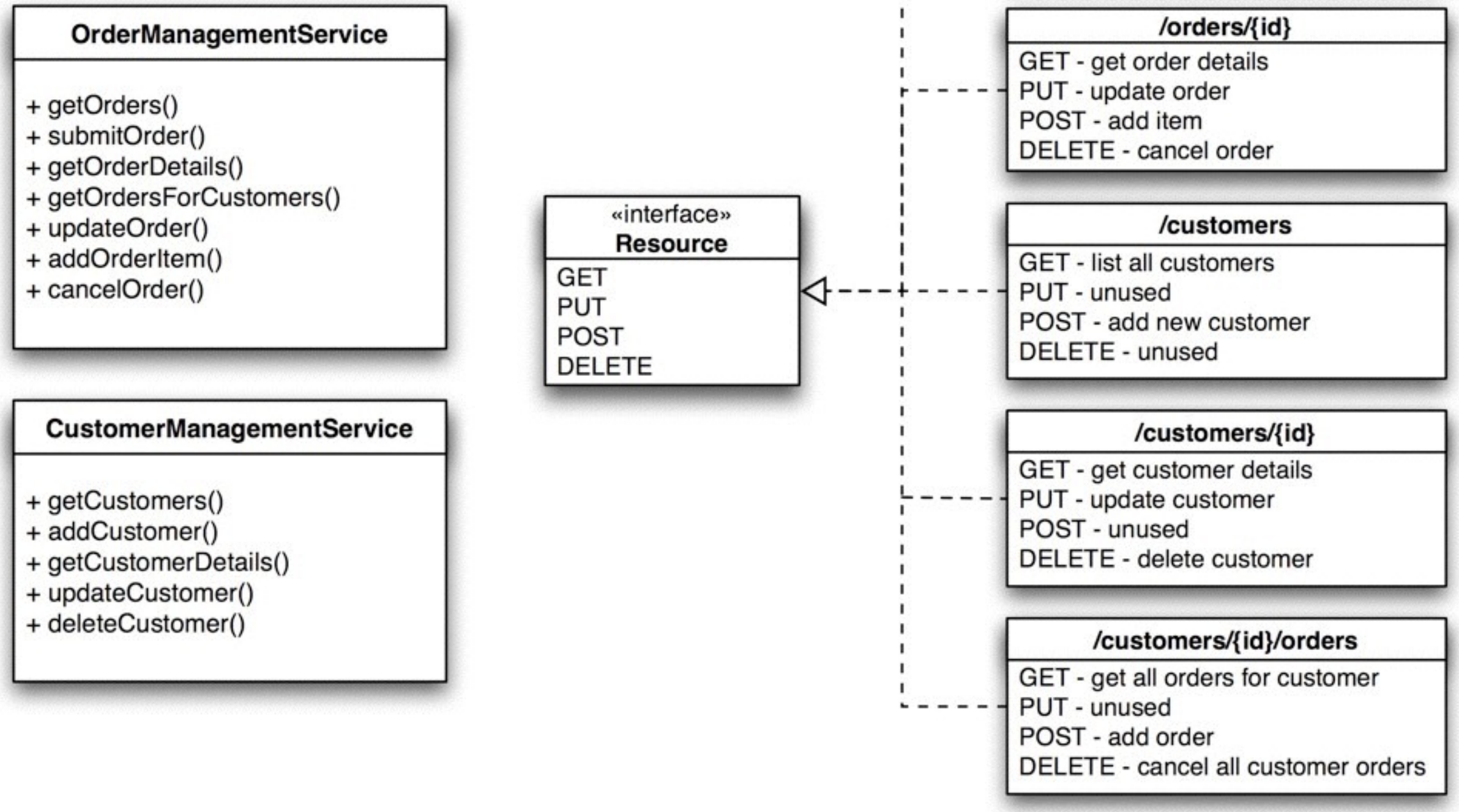
Design a RESTful API

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

- 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

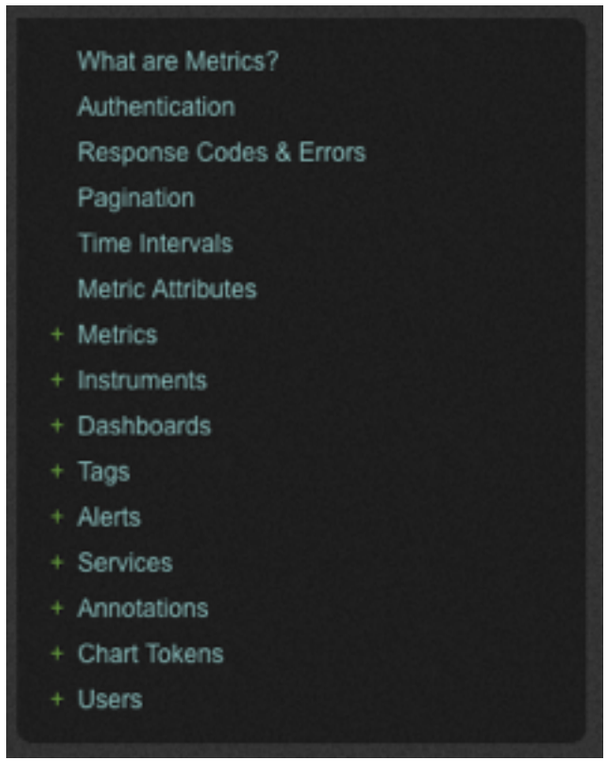

OrderManagementService
+ getOrders()
+ submitOrder()
+ getOrderDetails()
+ getOrdersForCustomers()
+ updateOrder()
+ addOrderItem()
+ cancelOrder()
CustomerManagementService
+ getCustomers()
+ addCustomer()
+ getCustomerDetails()
+ updateCustomer()
+ deleteCustomer()
«interface»
Resource
GET
PUT
POST
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

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

If you have a verb in your URI, you are probably doing something wrong!

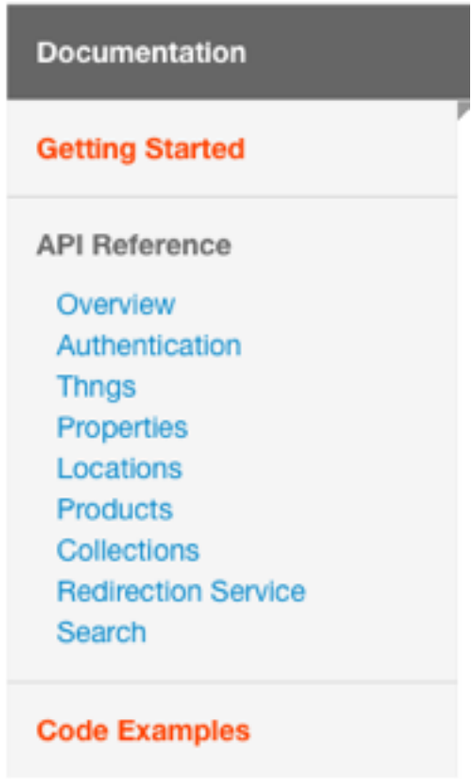
/api/stundents/create	POST /api/stundents/ HTTP/1.1
/api/stundents/22/delete	DELETE /api/stundents/22 HTTP/1.1

Look at Some Examples



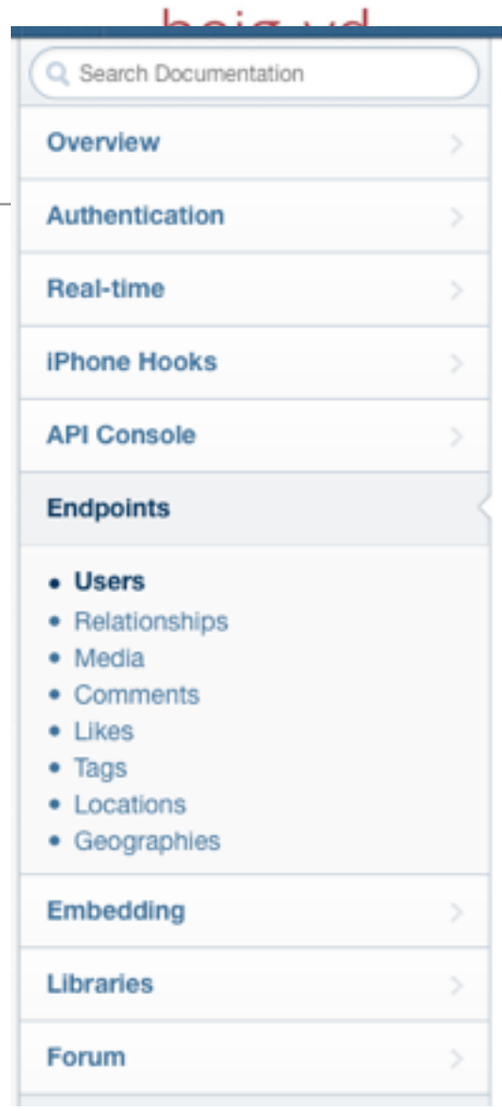
http://dev.librato.com/v1

librato



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

EVRYTHNG™



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

Instagram

librato

Short description of the resource (domain model)

Examples & payload structure

CRUD method description

navigator

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

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.
": null,
ts_long": "Fahrenheit",
cked": true

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

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.

- What are Metrics?
- Authentication
- Response Codes & Errors
- Pagination
- Time Intervals
- Metric Attributes
- Metrics
  - GET /metrics
  - POST /metrics
  - DELETE /metrics
  - GET /metrics/:name
  - PUT /metrics/:name
  - DELETE /metrics/:name
- + Instruments
- + Dashboards
- + Tags
- + Alerts
- + Services
- + Annotations
- + Chart Tokens
- + Users

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

Short description of the whole domain model

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.

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

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>, ...]
}
```

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

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

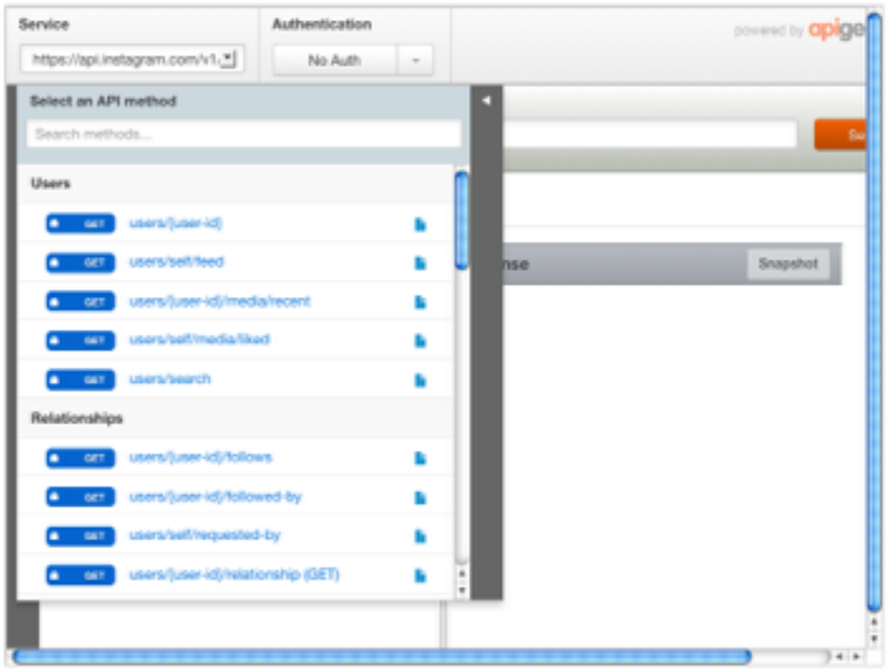
Access to our API is done via HTTPS requests to the `https://api.evrythng.com` domain. Unencrypted HTTP requests are accepted (`http://api.evrythng.com` for low-power device without SSL support), but we **strongly** suggest to use only HTTPS if you store any valuable data in our engine. Every request 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 →



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

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

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 RESPONSE


```
{
    "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.

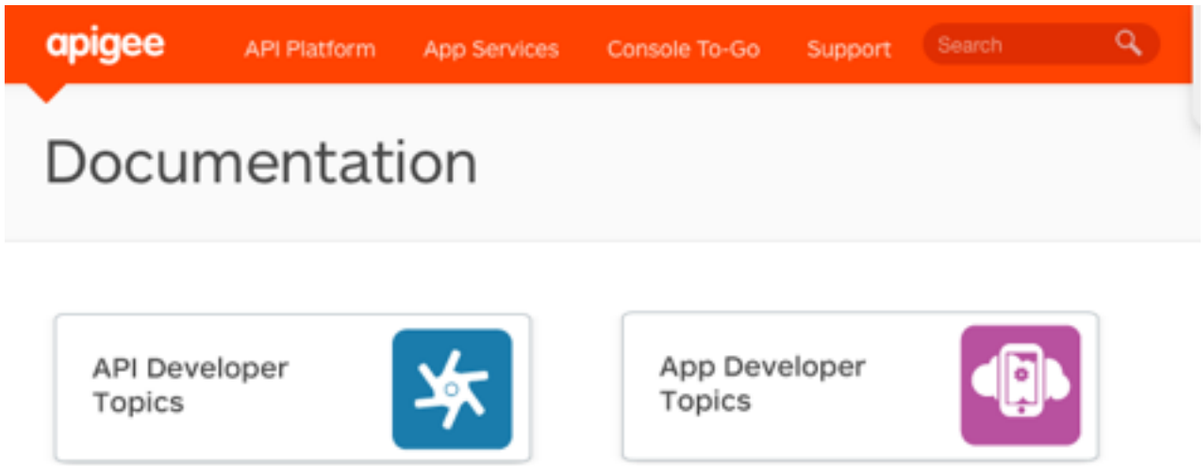
The Envelope

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

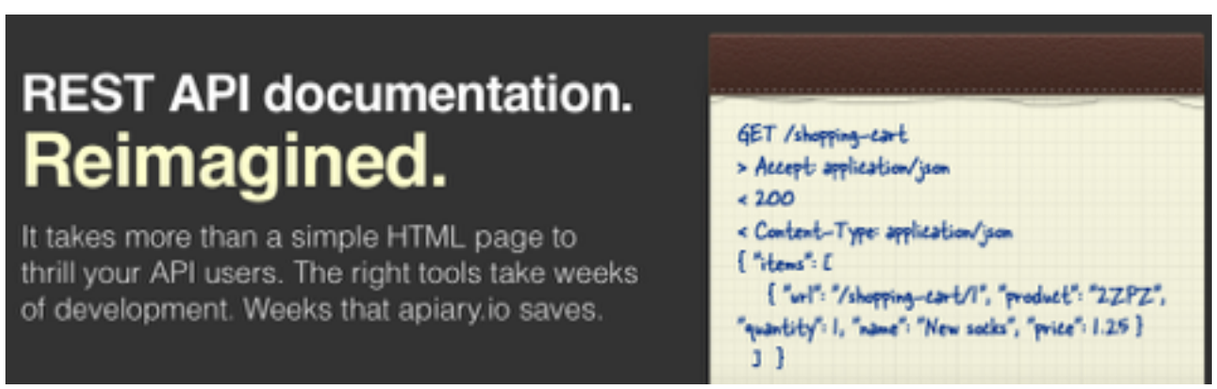

```
{
    "meta": {
        "code": 200
    },
    "data": {
        ...
    },
    "pagination": {
        "next_url": "...",
        "next_max_id": "13872296"
    }
}
```

Some Tools that Might Help/Inspire You

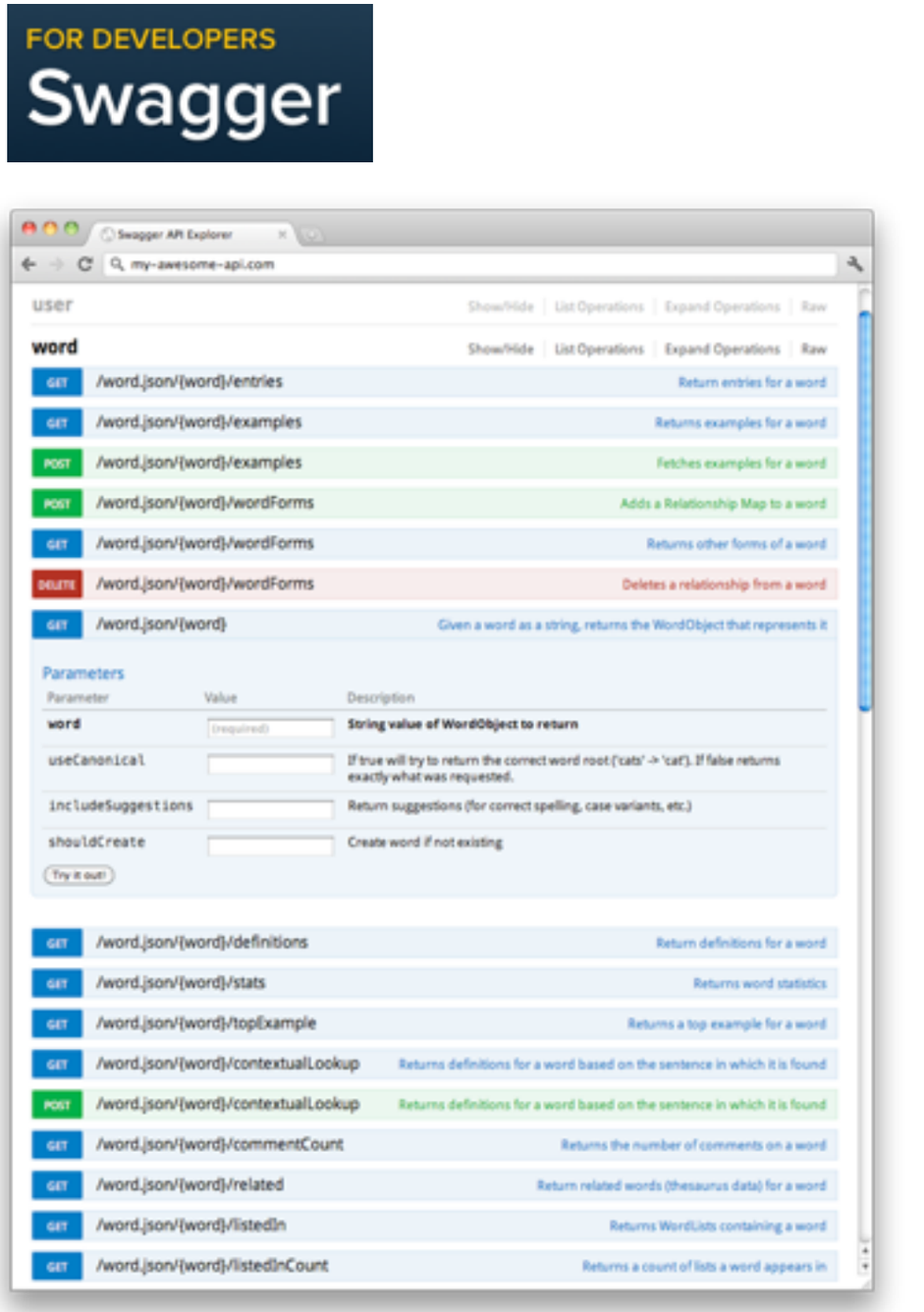
heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud



http://apigee.com/docs/



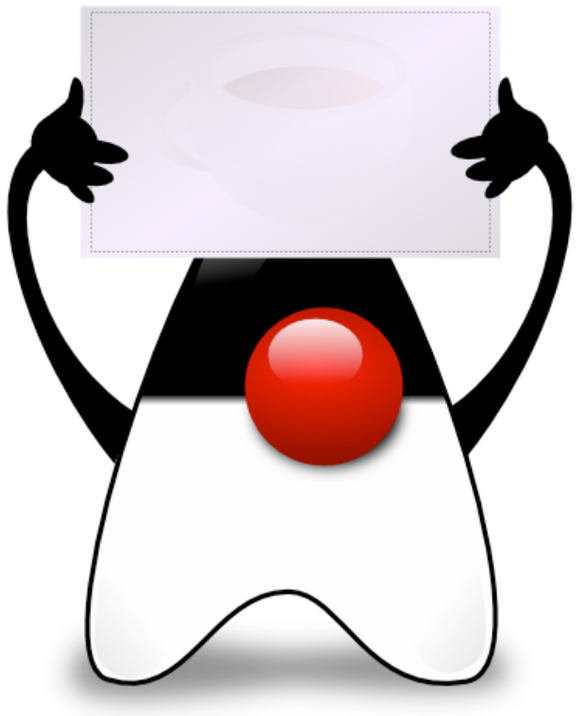
http://apiary.io/



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

heig-vd

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



RESTful APIs with JAX-RS

How to implement a REST API?

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

- 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, JSR 339).
  - Oracle provides the reference implementation, in the **Jersey project** (open source). This is the implementation that you get with **Glassfish**.

Java EE, JSRs and implementations

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

IBM WebSphere

JBoss by Red Hat

PostgreSQL *driver*

OpenJPA

Apache CXF

WildFly

ORACLE *driver*

HIBERNATE

RESTEasy

Glassfish

MySQL *driver*

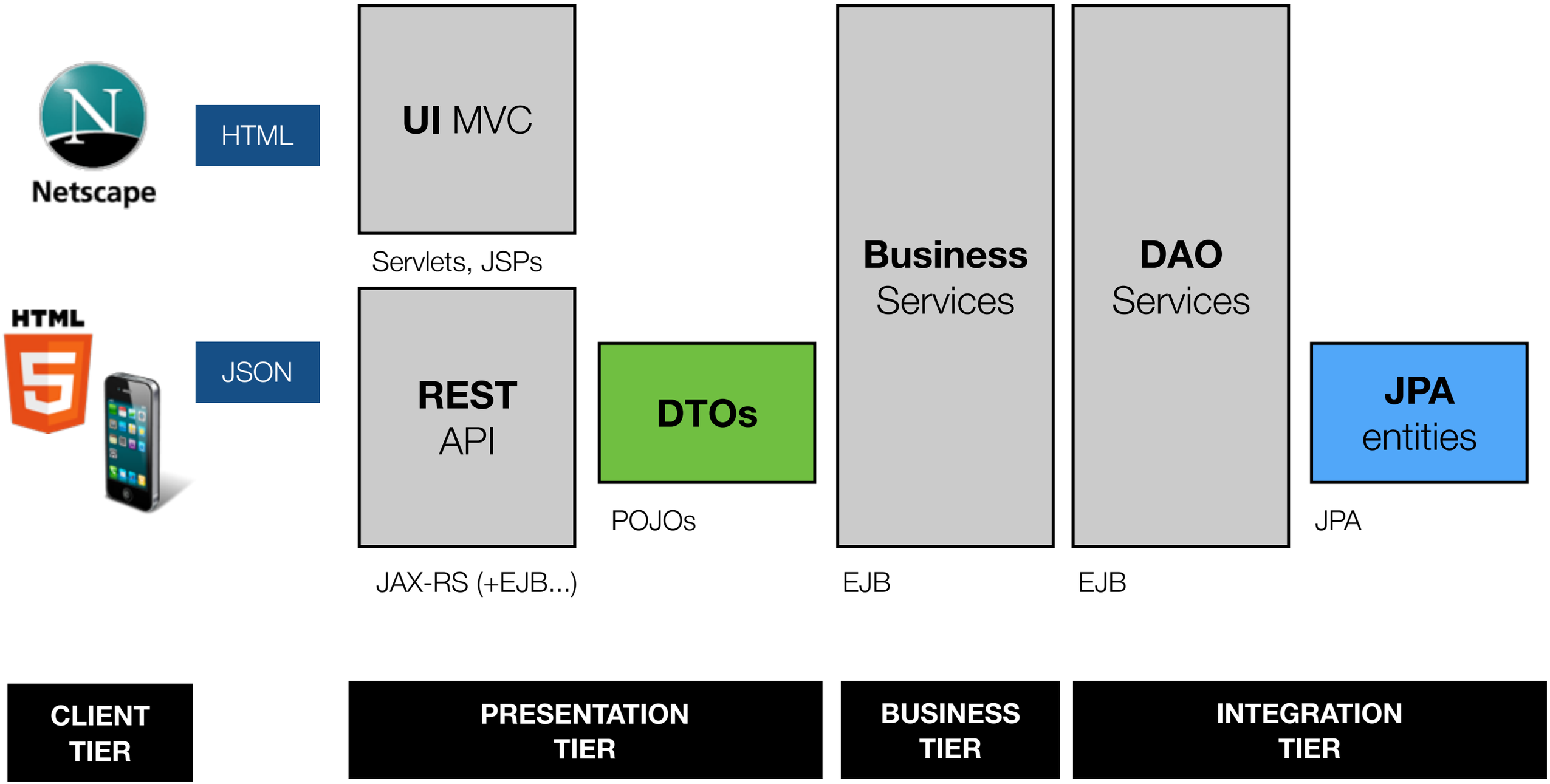
eclipselink

Jersey
RESTful Web Services in Java.

Java EE | **JDBC** | **JPA** | **JAX-RS**

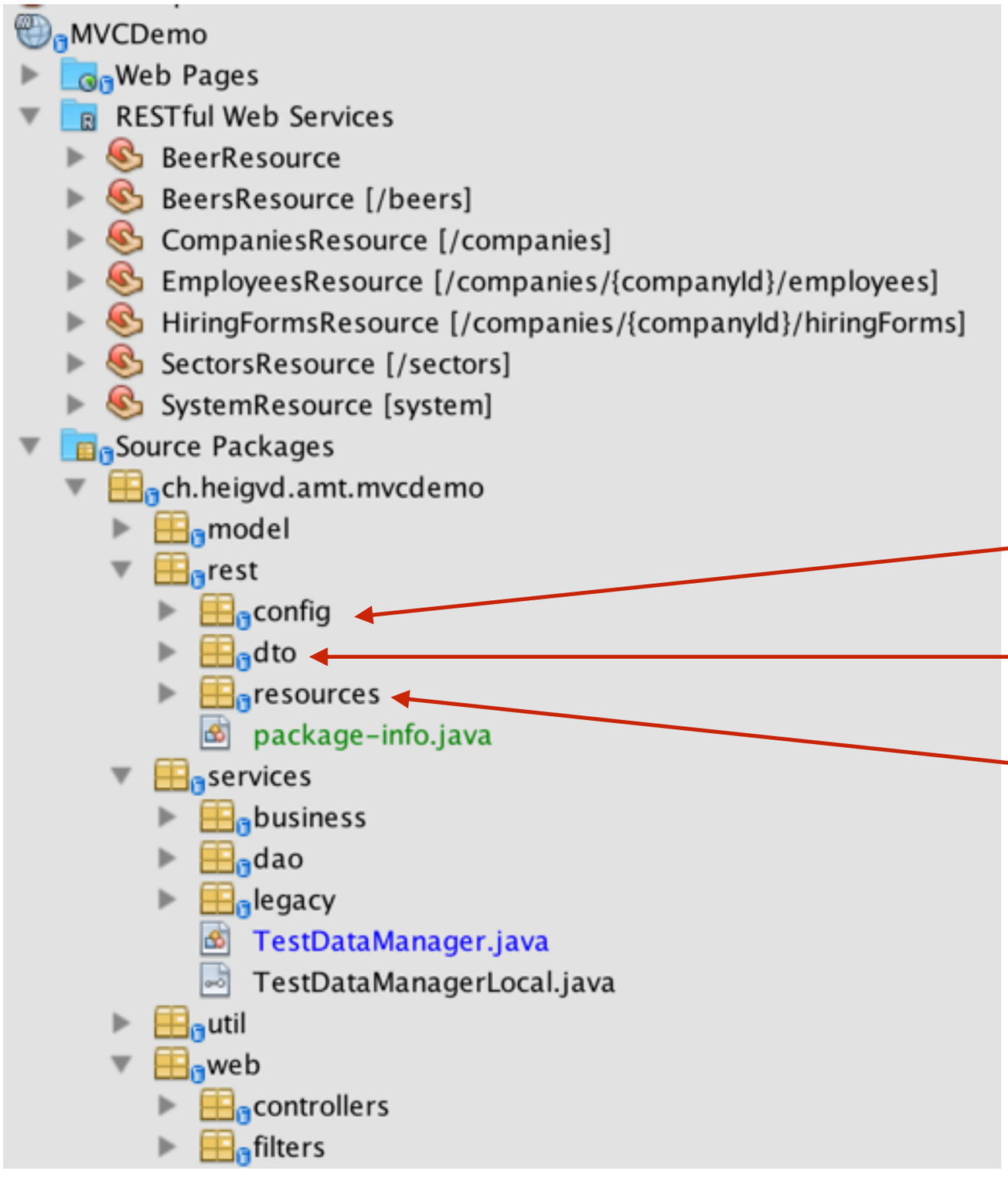
End-to-end Reference Architecture

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



End-to-end Reference Architecture

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



JAX-RS plumbing...

Objects **received from** / **sent to** clients via the REST API (JSON)

JAX-RS resource classes

JAX-RS 101

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

- JAX-RS is another example of applying **Inversion of Control** (IoC) in Java EE.
- As a developer, you create **Resource Classes** for your API endpoints:

`GET /users/ HTTP/1.1`	`UsersResource.java`
`POST /students/23/grades HTTP/1.1`	`StudentsResource.java`

- With various **annotations**, you tell the application server which classes and which methods should be invoked when HTTP requests are sent by clients:
  - Depending on the **URL**
  - Depending on the **HTTP method**
  - Depending on the **Accept** or **Content-type header**
- This is very similar what we have seen with Servlets in the past (with web.xml and @WebServlet)

See https://docs.oracle.com/javaee/7/tutorial/jaxrs002.htm#GILIK

JAX-RS 101

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

```
@Stateless
@Path("/beers")
public class BeersResource {

  @EJB
  BeersManagerLocal beersManager;

  @GET
  @Produces("application/json")
  public List<Beer> getBeers() {
    return beersManager.getAllBeers();
  }

  @POST
  @Consumes("application/json")
  public long addBeer(Beer beer) {
    return beersManager.add(beer);
  }
}
```

Even if we are conceptually in the presentation tier, we declare our JAX-RS resource class as a Stateless Session Bean.

This allows us to inject EJBs in the class. This also facilitates the use of the JPA persistence context (transaction starts when we enter the JAX-RS method)

IN THIS CASE, **DO NOT DEFINE A LOCAL INTERFACE** (it makes things a bit more complicated...)

```
GET /beers/ HTTP/1.1
Host: localhost:8080
Accept: application/json
```

```
POST /beers/ HTTP/1.1
Host: localhost:8080
Content-type: application/json

{
  "name" : "Cardinal",
  "country" : "Switzerland"
}
```

JAX-RS 101

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

- When you implement a REST API, you have to:
  - **serialize** Java objects into JSON (for GET methods)
  - **deserialize** JSON into Java objects (for POST/PUT/PATCH)
  - **JAX-RS** (with the help of **Jackson**, **JAXB** and other friends...) handles the serialization and deserialization for you.

```
@GET
@Produces("application/json")
public List<Beer> getBeers() {}

@POST
@Consumes("application/json")
public long addBeer(Beer beer)
```

See https://jersey.java.net/documentation/latest/user-guide.html#json
https://jersey.java.net/documentation/latest/user-guide.html#json.jackson

JAX-RS 101

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

- JAX-RS provides you with annotations to pass request attributes to your callback methods

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

   @GET
   @Path("/{studentId}/grades")
   @Produces("application/json")
   public List<Grade> getGrades(
      @PathParam(value="studentId") Long studentId,
      @QueryParam("ifLessThan") Double threshold
     ) {
       ...
     }
   }
}
```

```
GET /students/73/grades?ifLessThan=4 HTTP/1.1
Host: localhost:8080
Accept: application/json
```

See https://docs.oracle.com/javaee/7/tutorial/jaxrs002.htm#GIPYW

The Data Transfer Object (DTO) pattern

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

- **Requirement**: we need Java classes that capture the state of our business domain entities (Students, Companies, etc.), so that we can **exchange the related information with HTTP clients**.
- **Situation**: we have already created business entities in the `model` package. These are our JPA entities, which we use to exchange information between the services and the database.
- **Question**: can we simply r**euse these JPA** classes for the communication between the REST API and the clients?
- **Answer: not everybody agrees!**
- **Answer: I personally strongly recommend not to do that.**

The Data Transfer Object (DTO) pattern

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

- **Pattern description**: when you apply the DTO design pattern:
  - You introduce a package, where you define **simple Java Beans** (POJOs) with properties, getters and setters.
  - You create instances of these POJOs in the REST API layer and send them to clients (for GET requests). Typically, your business / DAO services give your JPA entities, **which you transform into DTOs**.
  - In the other direction, JAX-RS deserializes JSON into DTO instances. You use properties of these DTOs when you invoke business / DAO services (**you create the JPA entities and initialize them** with the content of the DTOs).

The Data Transfer Object (DTO) pattern

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

- **Pattern benefits**: there are **at least 4 reasons** for applying the pattern:
  - Control on the **visibility** of your data (**security**, **confidentiality**). In the MVCDemo project, employees have a **salary**. While it is necessary to have this property in the JPA entity (because it has to be stored in the database), it is clearly not something that you want to leak out via your REST API.
  - Have **full control on the data structure** presented to your clients. **Your API will be cleaner and easier to use**.
  - **Reduce the chattiness** and **improve the performance** of your clients applications. If you use JPA entities, then it is likely that REST clients will need to send a lot of HTTP requests to get all components of a page (1 call for the company, n calls for the sectors, etc.). With a DTO layer, you can aggregate multiple small business entities into a coarse-grained object that you send over the network.
  - **Avoid tricky technical issues**. If you try to use JPA entities, you will have to deal with circular references (@XmlTransient) and other issues. Trust me, you will spend a lot of time fighting with the underlying frameworks.

The Data Transfer Object (DTO) pattern

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

```
public class CompanySummaryDTO {

  private URI href;
  private List<String> sectors = new ArrayList<>();
  private String name;
  private long numberOfEmployees;
  private String ceo;

  public URI getHref() {
    return href;
  }

  public void setHref(URI href) {
    this.href = href;
  }

  public List<String> getSectors() {
    return sectors;
  }
...
}
```

```
@Stateless
@Path("/companies")
public class CompaniesResource {

  @GET
  @Produces("application/json")
  public List<CompanySummaryDTO> getCompanies() {
    List<CompanySummaryDTO> result = new ArrayList<>();
    List<Company> companies = companiesDAO.findAll();
    for (Company company : companies) {
      long companyId = company.getId();
      CompanySummaryDTO dto = new CompanySummaryDTO();
      populateSummaryDTOFromEntity(company, dto);
      result.add(dto);
    }
    return result;
  }

}
```

The Data Transfer Object (DTO) pattern

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

```
private CompanySummaryDTO populateSummaryDTOFromEntity(Company company, CompanySummaryDTO dto) {
  long companyId = company.getId();
  URI companyHref = uriInfo
    .getAbsolutePathBuilder()
    .path(CompaniesResource.class, "getCompany")
    .build(companyId);
  dto.setHref(companyHref);
  dto.setName(company.getName());
  dto.setNumberOfEmployees(companiesDAO.countEmployees(company.getId()));
  List<String> sectorsDTO = new ArrayList<>();
  for (Sector sector : company.getSectors()) {
    sectorsDTO.add(sector.getName());
  }
  dto.setSectors(sectorsDTO);

  List<Employee> employees = companiesDAO.findEmployeesByTitle(companyId, "CEO");
  if (employees.size() == 1) {
    Employee ceo = employees.get(0);
    dto.setCeo(ceo.getFirstName() + " " + ceo.getLastName());
  } else if (employees.isEmpty()) {
    dto.setCeo("There is no CEO");
  } else {
    dto.setCeo("There are " + employees.size() + " co-CEOs");
  }

  return dto;

}
```

heig-vd

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

Designing RESTful APIs

URL Structure

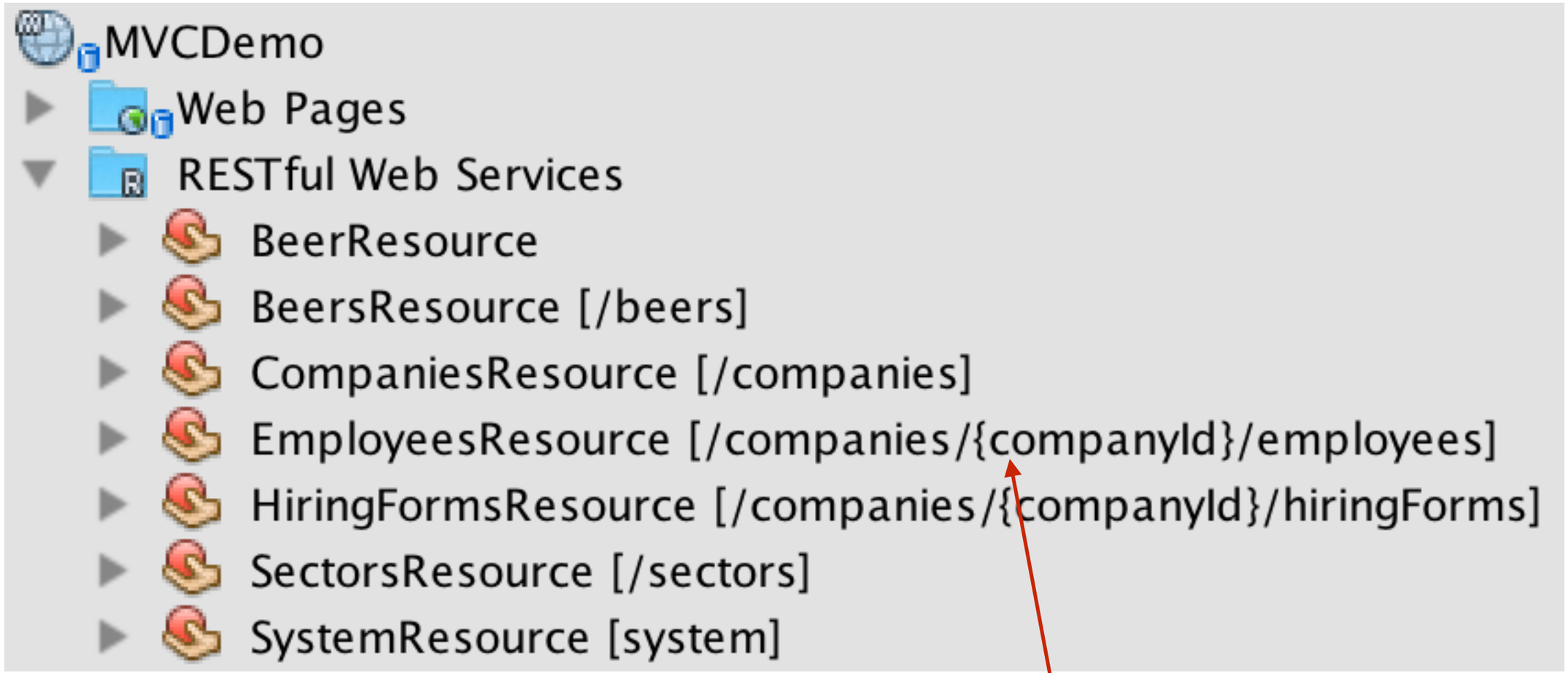
heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- 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

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



Motivations

All employees belong to one company

Accessing all employees via `/employees` could make sense for the platform administrator, but we don't have a use case yet

URL Structure

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

```
@Stateless
@Path("/companies/{companyId}/employees")
public class EmployeesResource {

  @Context UriInfo uriInfo;
  @EJB private EmployeesDAOLocal employeesDAO;
  @EJB private CompaniesDAOLocal companiesDAO;

  @GET
  @Produces("application/json")
  public List<EmployeeSummaryDTO> getEmployees(@PathParam(value="companyId") long companyId) {
    List<EmployeeSummaryDTO> result = new ArrayList<>();
    List<Employee> employees = companiesDAO.findAllEmployeesForCompanyId(companyId);
    for (Employee employee : employees) {
      long employeeId = employee.getId();

      URI href = uriInfo.getAbsolutePathBuilder().path(EmployeesResource.class, "getEmployee").build(employeeId);

      EmployeeSummaryDTO dto = new EmployeeSummaryDTO();
      dto.setHref(href);
      dto.setFirstName(employee.getFirstName()); dto.setLastName(employee.getLastName()); dto.setTitle(employee.getTitle());
      result.add(dto);
    }
    return result;
  }

  @GET
  @Path("/{id}")
  @Produces("application/json")
  public EmployeeSummaryDTO getEmployee(@PathParam(value="id") long id) throws BusinessDomainEntityNotFoundException {
      Employee employee = employeesDAO.findById(id);
      return new EmployeeSummaryDTO(employee.getFirstName(), employee.getLastName(), employee.getTitle());
  }

}
```

Linked resources

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

- In most domain models, you have relationships between domain entities:
  - Example: one-to-many relationship between "Company" and "Employee"
- Imagine that you have the following REST endpoints:
  - `GET /companies/{id}` to retrieve one company by id
- Question: what payload do you expect when invoking this URL?

Linked resources

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

```
{
  "name": "Apple",
  "address" : {},
  "employees" : [
    {
      "firstName" : "Tim",
      "lastName" : "Cook",
      "title" : "CEO"
    },
    {
      "firstName" : "Jony",
      "lastName" : "Ive",
      "title" : "CDO"
    }
  ]
}
```

Embedding
(reduces "chattiness", often good if there are "few" linked resources; company-employee is not a good example)

```
{
  "name": "Apple",
  "address" : {},
  "employeeIds" : [134, 892, 918, 9928]
```

References via IDs
(not recommended: the client must know the URL structure to retrieve an an employee)

```
{
  "name": "Apple",
  "address" : {},
  "employeeURLs" : [
    "/companies/89/employees/134",
    "/companies/89/employees/892",
    "/companies/89/employees/918",
    "/contractors/255/employees/9928",
 ]
}
```

References via URLs
(better: decouples client and server implementation)

Linked resources

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

```
[
    {
        "href": "http://localhost:8080/MVCDemo/api/companies/1",
        "sectors": [
            "IT",
            "Telecommunications",
            "Entertainment"
        ],
        "name": "Apple",
        "numberOfEmployees": 85,
        "ceo": "Tim Cook"
    },
    {
        "href": "http://localhost:8080/MVCDemo/api/companies/9",
        "sectors": [
            "Sector-1444102164062-4",
            "Sector-1444102164062-5"
        ],
        "name": "Company-1444102164062-2",
        "numberOfEmployees": 0,
        "ceo": "There is no CEO"
    },
    {
        "href": "http://localhost:8080/MVCDemo/api/companies/4",
        "sectors": [
            "Financials"
        ],
        "name": "UBS",
        "numberOfEmployees": 0,
        "ceo": "There is no CEO"
    }
]
```

```
@Stateless
@Path("/companies")
public class CompaniesResource {

  @GET
  @Produces("application/json")
  public List<CompanySummaryDTO> getCompanies() {
    List<CompanySummaryDTO> result = new ArrayList<>();
    List<Company> companies = companiesDAO.findAll();
    for (Company company : companies) {
      long companyId = company.getId();
      CompanySummaryDTO dto = new CompanySummaryDTO();
      populateSummaryDTOFromEntity(company, dto);
      result.add(dto);
    }
    return result;
  }

}
```

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

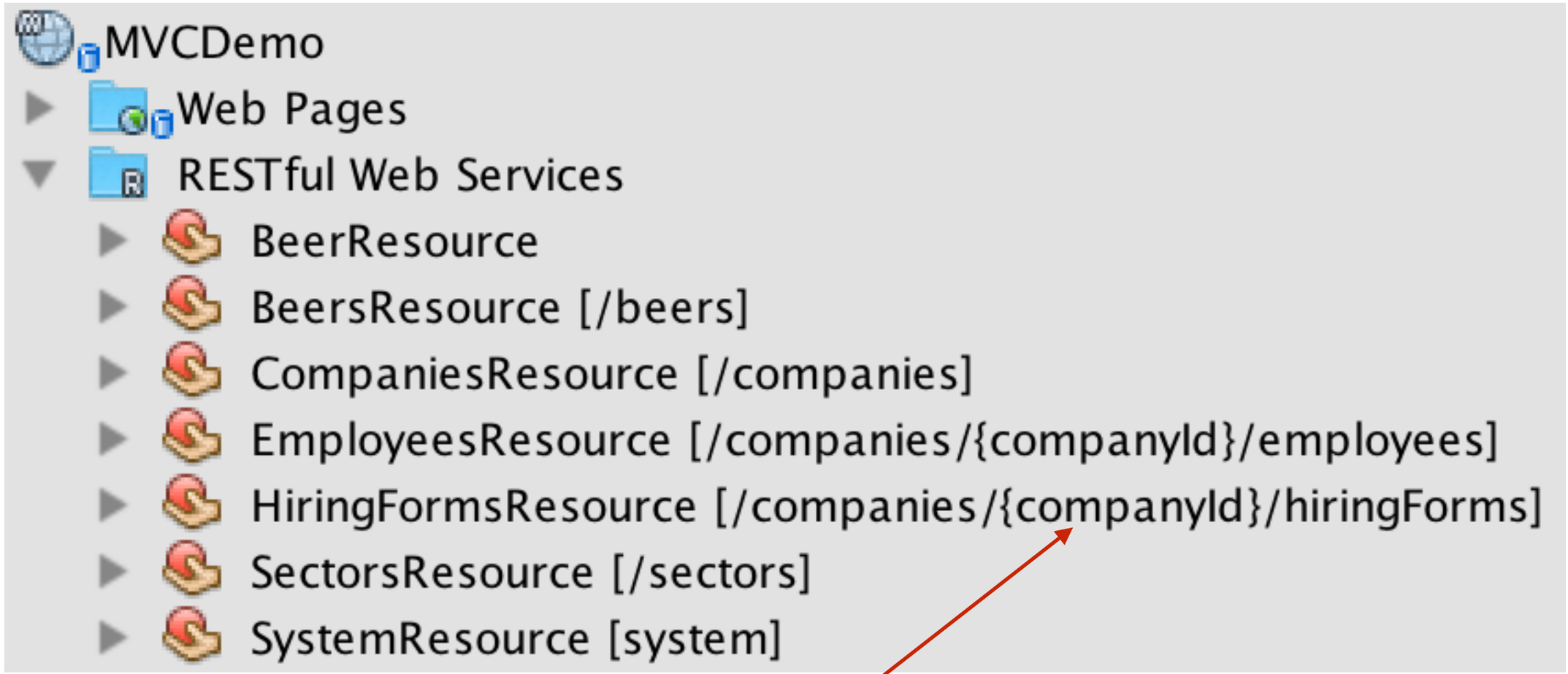
Resources & Actions (2)

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.).

Resources & Actions (3)

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



When we hire an employee, we don't do a POST on /companies/23/employees. We don't do a POST on /employees/.

Instead, we create a HiringForm resource, which contains the employee details and the title. A business service processes this form

Resources & Actions (4)

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

```
@POST
@Consumes("application/json")
public Response submitHiringForm(HiringFormDTO hiringForm, @PathParam("companyId") long companyId)
  throws BusinessDomainEntityNotFoundException {

  Company company = companiesDAO.findById(companyId);
  Employee employee = humanResourcesService.hireEmployee(company, hiringForm);

  URI newHireURI = uriInfo
    .getBaseUriBuilder()
    .path(EmployeesResource.class)
    .path(EmployeesResource.class, "getEmployee")
    .build(company.getId(), employee.getId());

  return Response
    .created(newHireURI)
    .build();

}
```

Resources & Actions (4)

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

```
  @Override
  public Employee hireEmployee(Company company, HiringFormDTO hiringForm) throws
BusinessDomainEntityNotFoundException {

    company = companiesDAO.findById(company.getId());
    Employee newHire = new Employee();
    newHire.setFirstName(hiringForm.getFirstName());
    newHire.setLastName(hiringForm.getLastName());
    newHire.setTitle(hiringForm.getTitle());
    assignStartingSalary(newHire);

    newHire = employeesDAO.createAndReturnManagedEntity(newHire);
    companiesDAO.hire(company, newHire);
    return newHire;
  }

  private void assignStartingSalary(Employee employee) {
    String title = employee.getTitle();
    switch (title) {
      case "CEO":
        employee.setBasicSalary(Chance.randomDouble(1, 200000));
        employee.setBonus(Chance.randomDouble(50000, 5000000));
        break;
      case "software engineer":
        employee.setBasicSalary(80000);
        employee.setBonus(1000);
        break;
      default:
        employee.setBasicSalary(Chance.randomDouble(50000, 150000));
        employee.setBonus(Chance.randomDouble(0, 50000));
    }
```

Pagination (1)

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

- 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)

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

- **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)

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

- **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

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

- 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

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

- 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.)

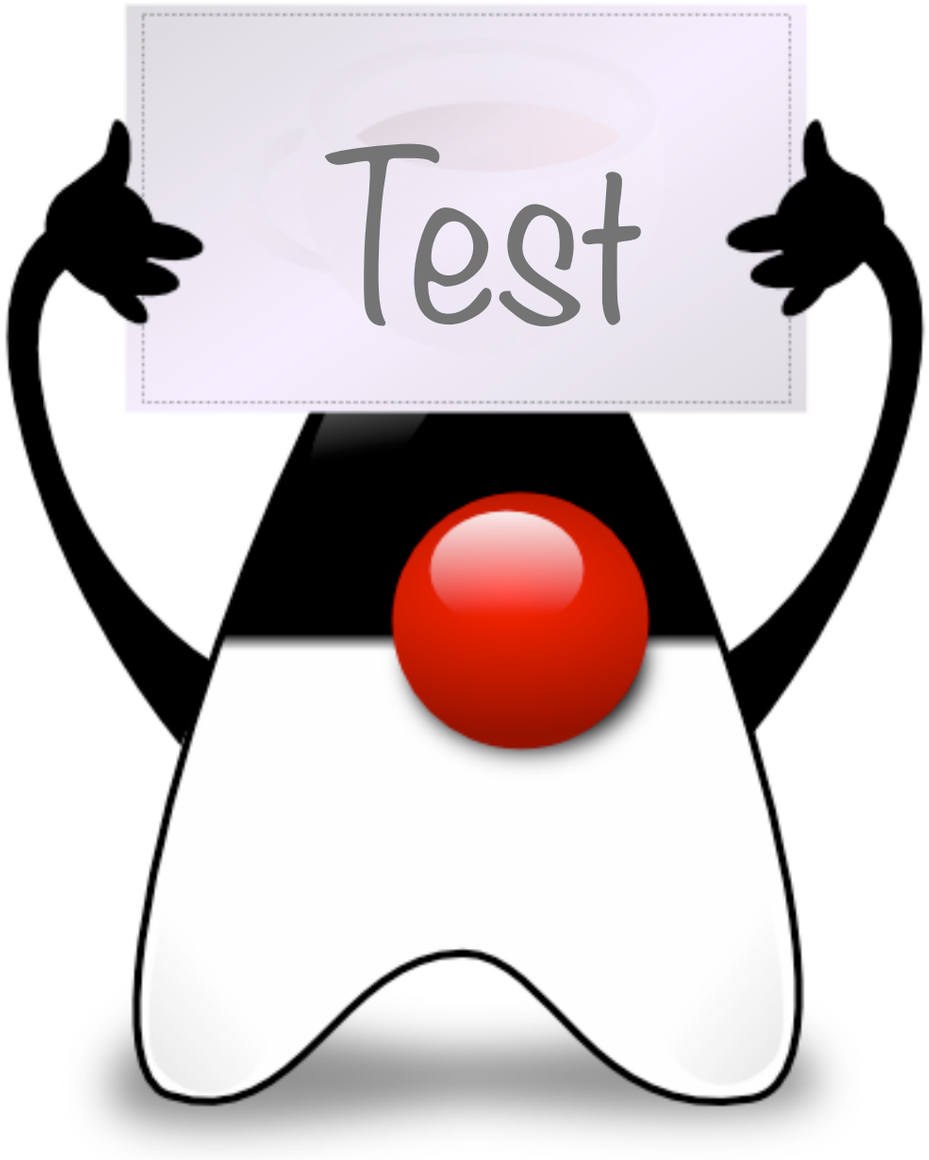
- When you introduce a change in your API (and in the corresponding documentation), you will have a **compatibility issue**. Namely, you will have to support **some clients that still use the old version** of the API and **others that use the new version of the API**.

- 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



Testing REST API

Testing the REST API

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

- So far, we have already seen **different types tests**:
  - **Non functional tests** (performance, scalability, etc.) with **JMeter**
  - User acceptance tests (controlled browser) with **Selenium** & **WebDriver**
  - You already knew about JUnit tests

- There are different strategies, tools and frameworks to test the REST API. We have used one approach in the **MVCDemoUserAcceptanceTests** project:
  - We use the **Jersey Client** framework
  - This provides us with a **fluent API** that makes it easier to prepare HTTP requests and to inspect HTTP responses

Testing the REST API

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

Send an HTTP request

```
@Test
public void itShouldBePossibleToListCompanies() throws IOException {
  WebTarget target = client.target("http://localhost:8080/MVCDemo/api").path("companies");
  Response response = target.request().get();

  assertThat(response.getStatus()).isEqualTo(Response.Status.OK.getStatusCode());

  String jsonPayload = response.readEntity(String.class);
  assertThat(jsonPayload).isNotNull();
  assertThat(jsonPayload).isNotEmpty();

  JsonNode[] asArray = mapper.readValue(jsonPayload, JsonNode[].class);
  assertThat(asArray).isNotNull();
  assertThat(asArray.length).isNotEqualTo(0);

  for (JsonNode company : asArray) {
    assertThat(company.get("ceo")).isNotNull();
    assertThat(company.get("sectors")).isNotNull();
    assertThat(company.get("numberOfEmployees")).isNotNull();
  }
}
```

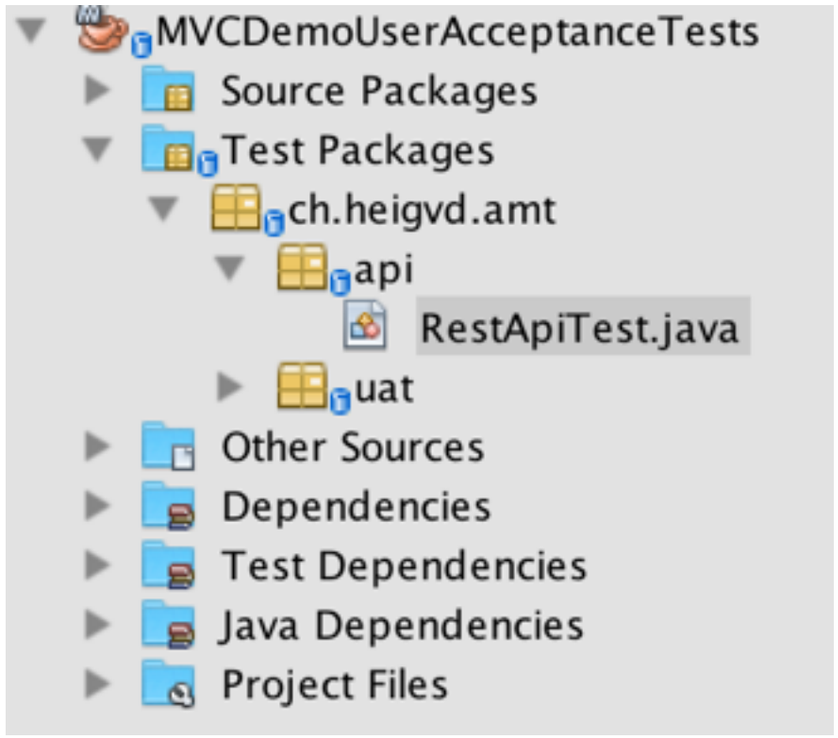
Check that we get a 200

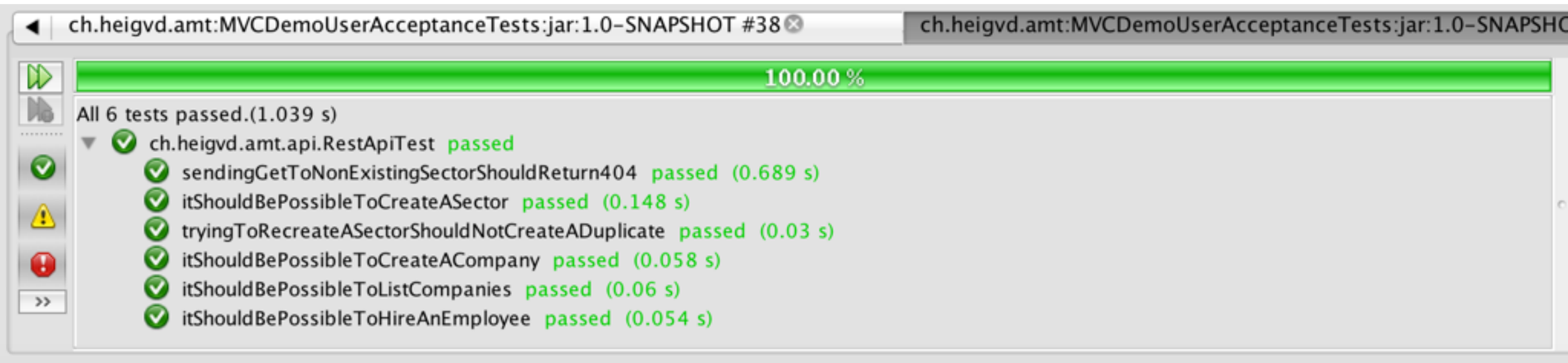
Get the JSON payload as string and parse it

Iterate over the array of companies and validate that JSON properties are there

Testing the REST API

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





Probe Dock

Olivier

https://trial.probedock.io/wasabi-technologies/reports

Probe Dock | Dashboard | Reports | Projects | Help | oliechti

Wasabi Technologies

Latest Reports | 09:18 by oliechti

Results	Runner(s)	Duration	Date	Details
6	oliechti	1s 90ms	Tue, Oct 6, 2015 9:18 AM	MVCDemoJavaEE 1.0.0 unit
6	oliechti	788ms	Tue, Oct 6, 2015 5:32 AM	MVCDemoJavaEE 1.0.0 unit
6	oliechti	785ms	Tue, Oct 6, 2015 5:30 AM	MVCDemoJavaEE 1.0.0 unit
5 1	oliechti	856ms	Tue, Oct 6, 2015 5:29 AM	MVCDemoJavaEE 1.0.0 unit
5 1	oliechti	915ms	Tue, Oct 6, 2015 5:18 AM	MVCDemoJavaEE 1.0.0 unit
5	oliechti	903ms	Tue, Oct 6, 2015 5:02 AM	MVCDemoJavaEE 1.0.0 unit
13	oliechti	37s 378ms	Mon, Oct 5, 2015 9:59 PM	MVCDemoJavaEE 1.0.0 unit
8	oliechti	34s 797ms	Mon, Oct 5, 2015 9:49 PM	MVCDemoJavaEE 1.0.0 unit

Probe Dock v0.1.9

Probe Dock

Olivier

https://trial.probedock.io/wasabi-technologies/reports/mwqd2x4ay9ze

Probe Dock | Dashboard | **Reports** | Projects | Help | oliechti

Wasabi Technologies

Latest Reports | 09:18 by oliechti

Test Run Report - Tue, Oct 6, 2015 9:18 AM

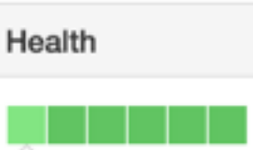
MVCDemoJavaEE 1.0.0 **unit**

Details

- 6 test results
- Run in 1s 90ms

Summary

Health



Rest api test: it should be possible to create a company

Duration: 59ms

Filter by result

Filter by category

All categories

Filter by ticket

All tickets

Filter by name

Any name

Filter by tag

All tags

Probe Dock v0.1.9

Probe Dock

Olivier

https://trial.probedock.io/wasabi-technologies/reports/mwqd2x4ay9ze

Probe Dock | Dashboard | Reports | Projects | Help | oliechti

Rest api test: it should be possible to create a company

MVCDemoJavaEE 1.0.0 unit

Run: Oct 6, 2015 9:18 AM **By:** oliechti **Duration:** 59ms

Rest api test: it should be possible to create a sector

MVCDemoJavaEE 1.0.0 unit

Run: Oct 6, 2015 9:18 AM **By:** oliechti **Duration:** 148ms

Rest api test: it should be possible to hire an employee

MVCDemoJavaEE 1.0.0 unit

Run: Oct 6, 2015 9:18 AM **By:** oliechti **Duration:** 55ms

Rest api test: it should be possible to list companies

MVCDemoJavaEE 1.0.0 unit

Run: Oct 6, 2015 9:18 AM **By:** oliechti **Duration:** 60ms

Rest api test: sending get to non existing sector should return 404

Probe Dock v0.1.9