# API Slides Companion

# Anatomia API REST

curl -s

https://api.github.com/repos/didattica-forever/Chirpy (address API)

https://api.github.com/repos/didattica-forever/Chirpy (address API)
<filtro_username>/<filtro_reponame> (filtri)

/repos/<filtro_username>/<filtro_reponame> API

/repos/{username}/{reponame} API

result

JSON

# SOAP

```
curl -X POST https://www.example.com/weatherService \
  -H "Content-Type: text/xml" \
  -d '<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetCityWeather xmlns="http://weather.example.com/">
      <CityName>Rome</CityName>
    </GetCityWeather>
  </soap:Body>
</soap:Envelope>'
```

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Body>
  <GetCityWeatherResponse>
   <GetCityWeatherResult>
    <Temperature>18</Temperature>
    <Condition>Sunny</Condition>
   </GetCityWeatherResult>
  </GetCityWeatherResponse>
 </soap:Body>
</soap:Envelope>
```

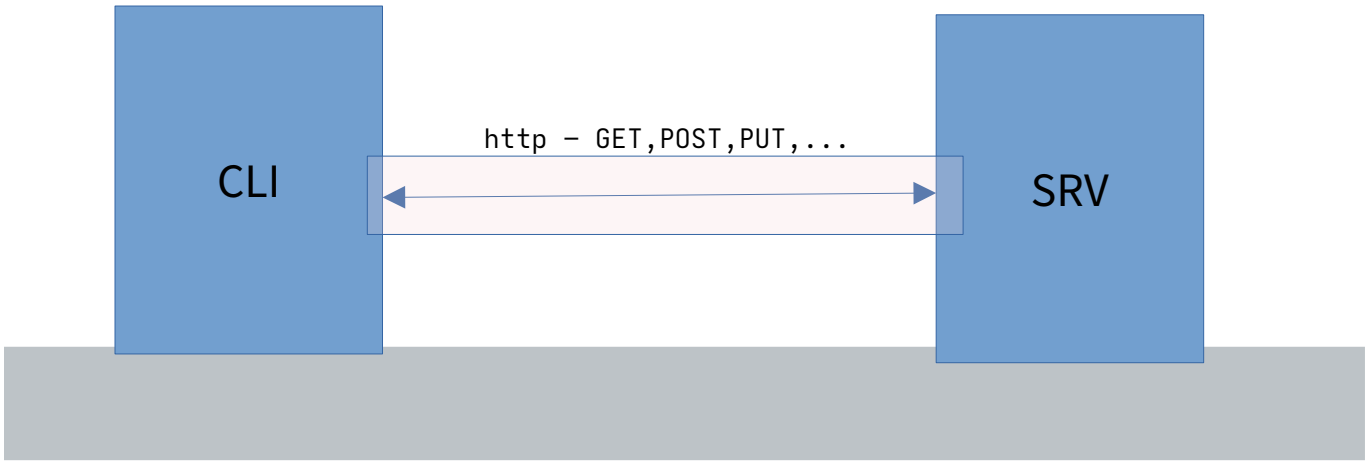
# REST

REST trasferimento stato (Es: da server a client)

1) POST /customers — crea nuovo cliente (payload necessario)

2) GET /customers/{id} — profilo cliente

3) elenco di tutti i clienti? ==> GET /customers



# CRUD & HTTP

| CRUD | SQL | HTTP | DAO/REPO |
|---|---|---|---|
| **C**reate | insert | POST | create |
| **R**ead | select | GET | find<br>findAll<br>findBy+sort |
| **U**pdate | update | PUT (100%)<br>PATCH | update<br>merge |
| **D**elete | delete | DELETE | delete<br>remove |

# Idempotenza

PUT /devices/12 { "mode": "AUTO" } - IDEM

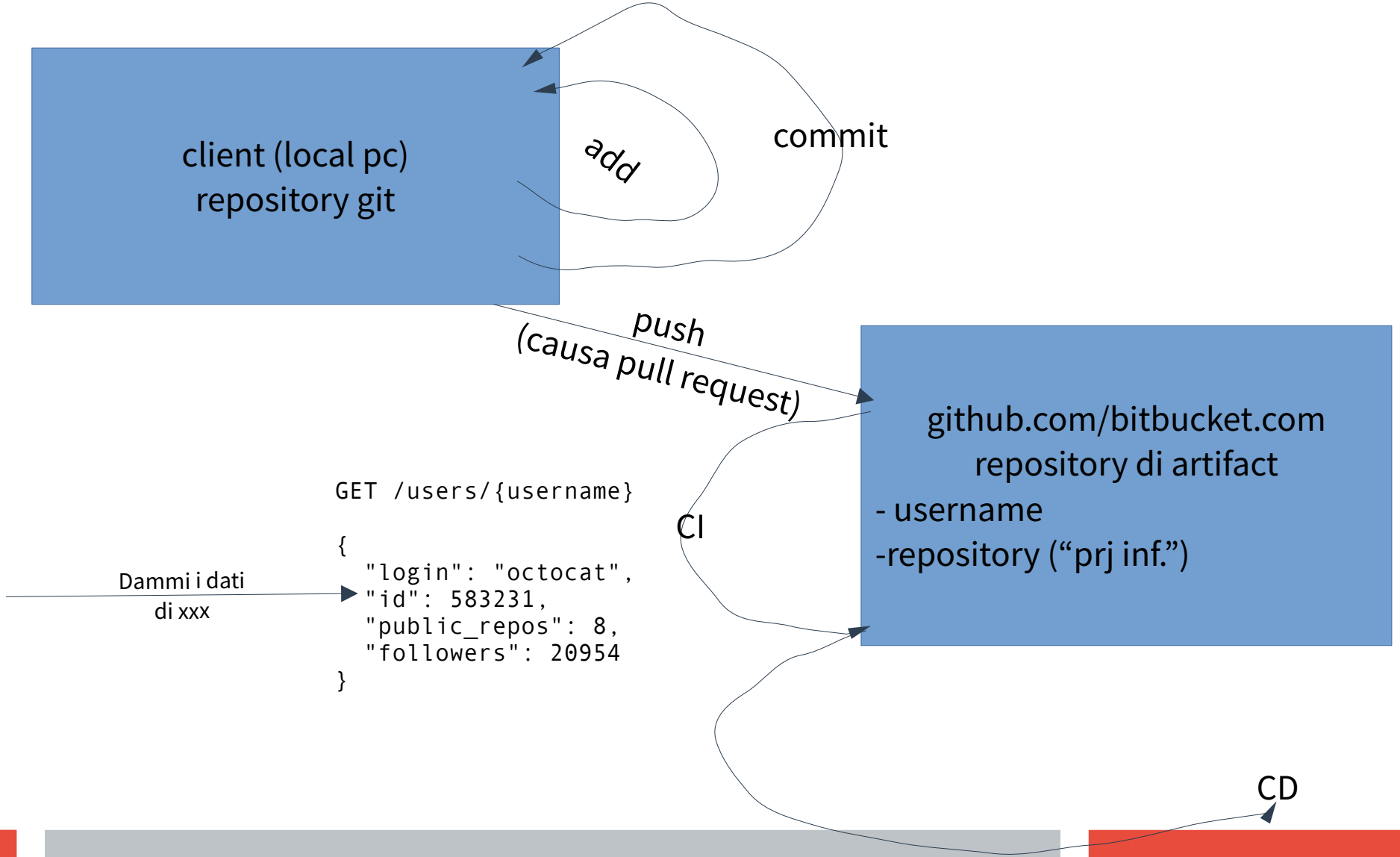
POST /logs { "event": "login" } - NO

DELETE /sessions/99 - IDEM

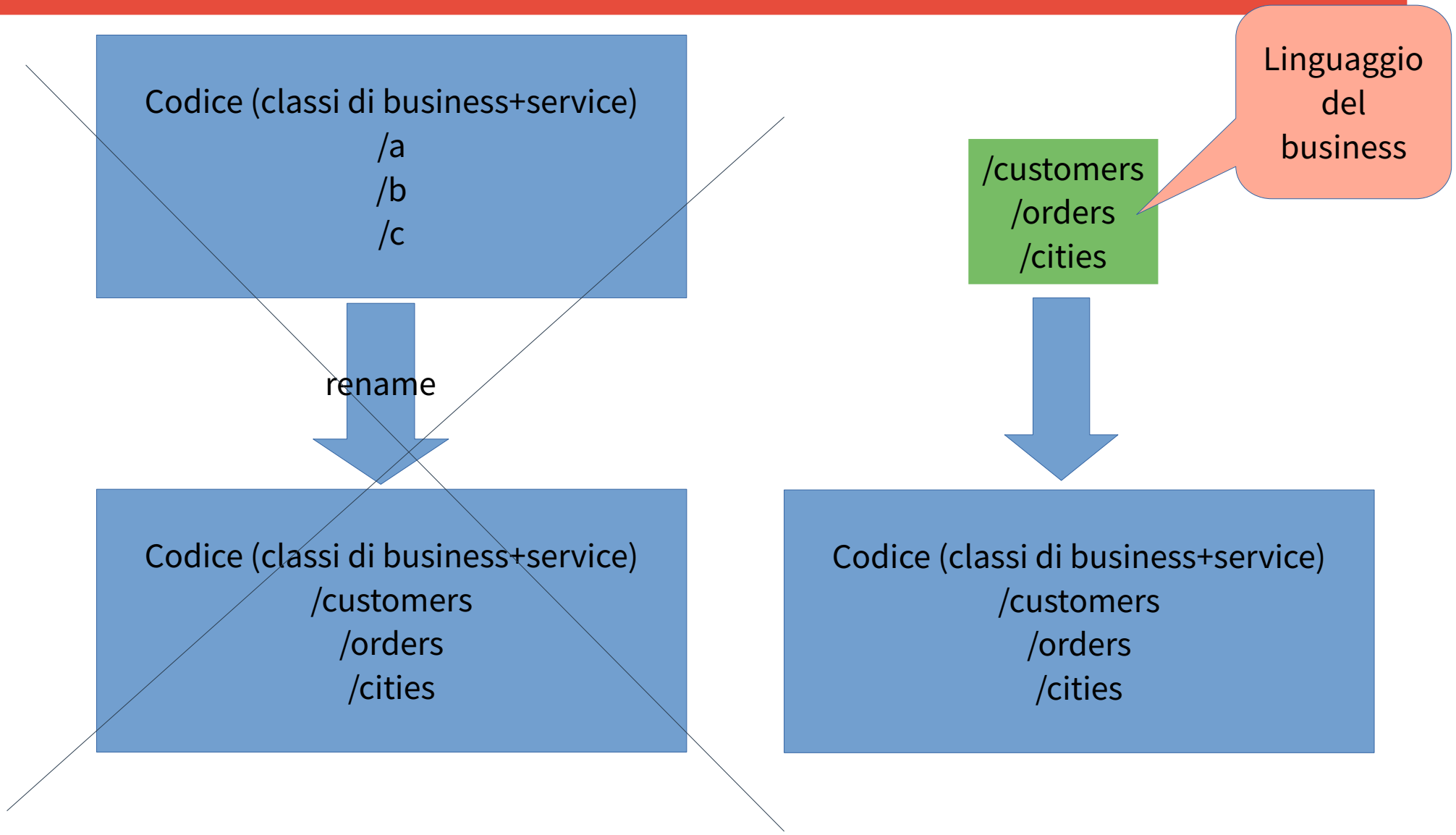
UPDATE accounts SET score = score + 1 – NO

UPDATE accounts SET saldo = credit + debit - NO

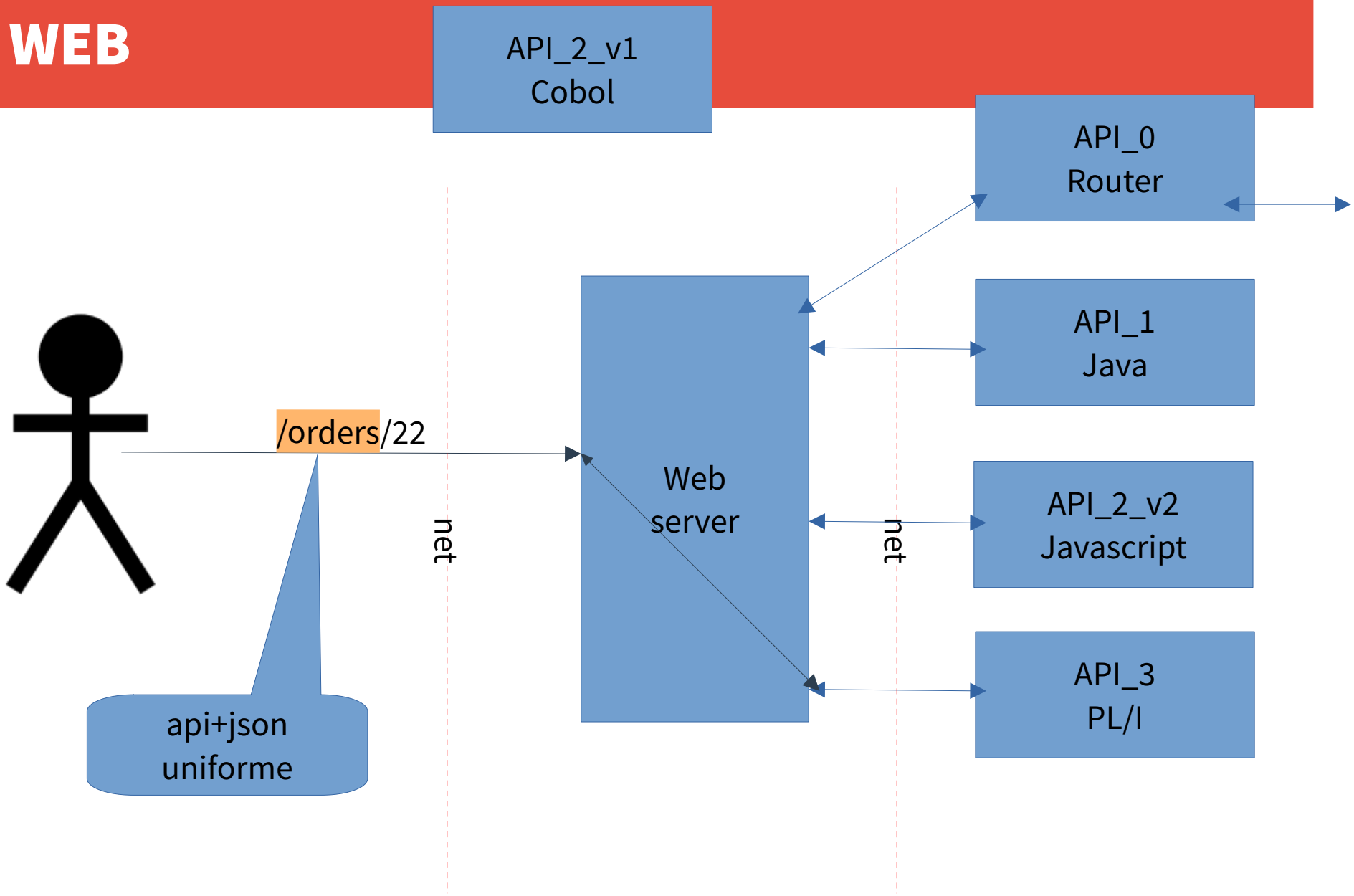
# Git & Github


client (local pc)
repository git
add
commit
push
(causa pull request)
github.com/bitbucket.com
repository di artifact
- username
-repository (“prj inf.”)
CI
CD
Dammi i dati
di xxx
GET /users/{username}
{
"login": "octocat",
"id": 583231,
"public_repos": 8,
"followers": 20954
}

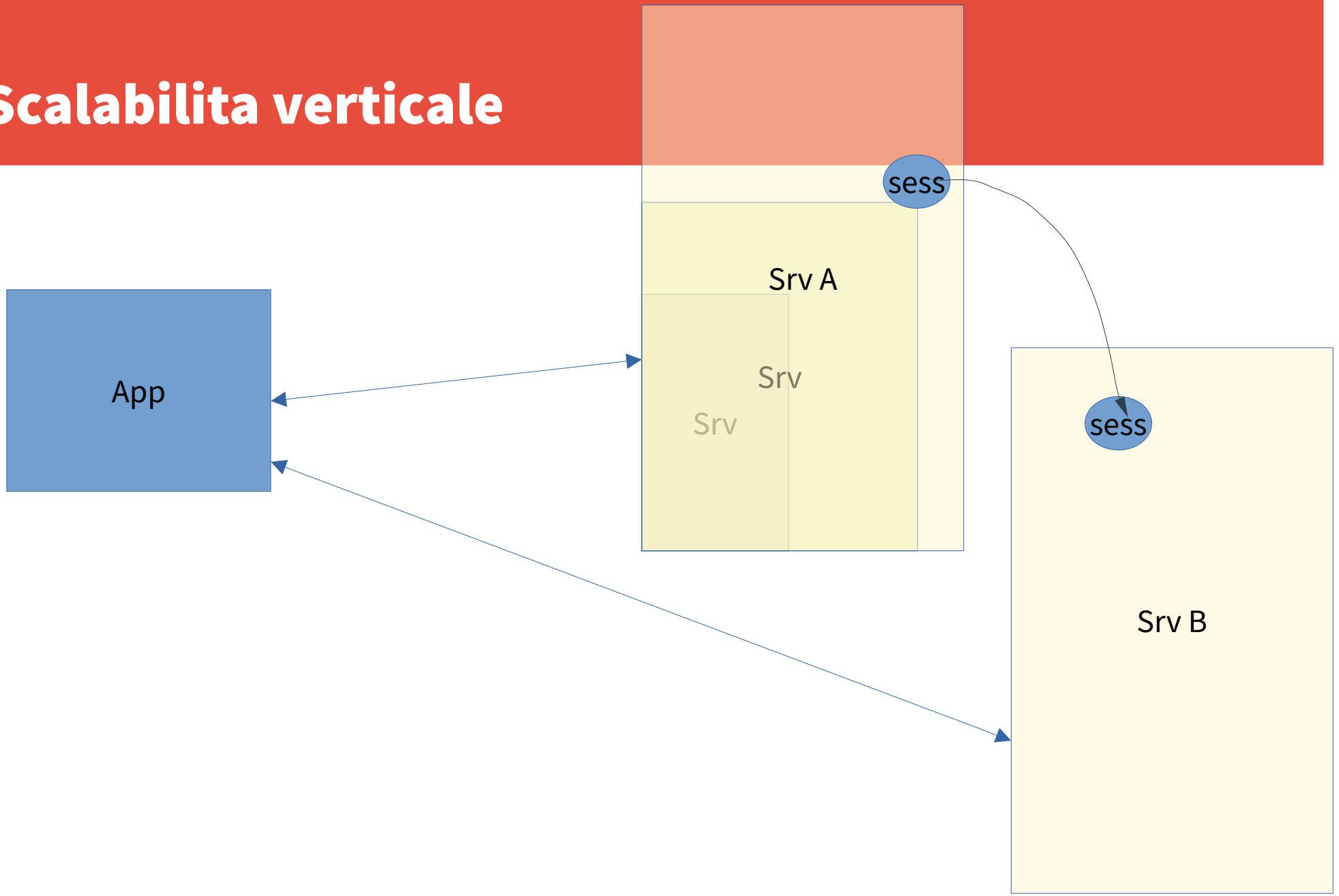
# realizzazione API REST



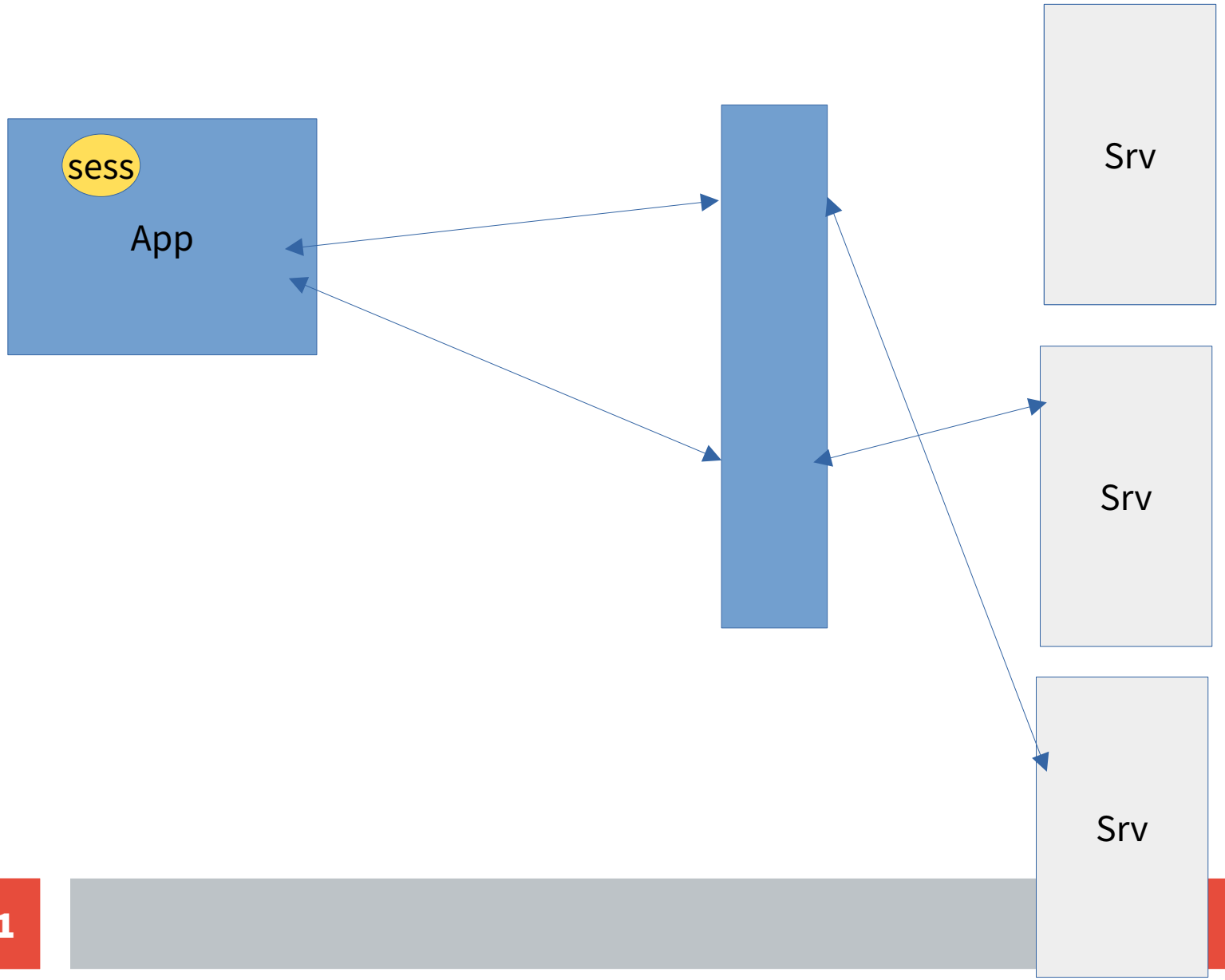
# API WEB


API_2_v1
Cobol
API_0
Router
API_1
Java
API_2_v2
Javascript
API_3
PL/I
Web
server
/orders/22
api+json
uniforme
net
net

# Scalabilita verticale



# Scalabilita orizzontale



# Cache

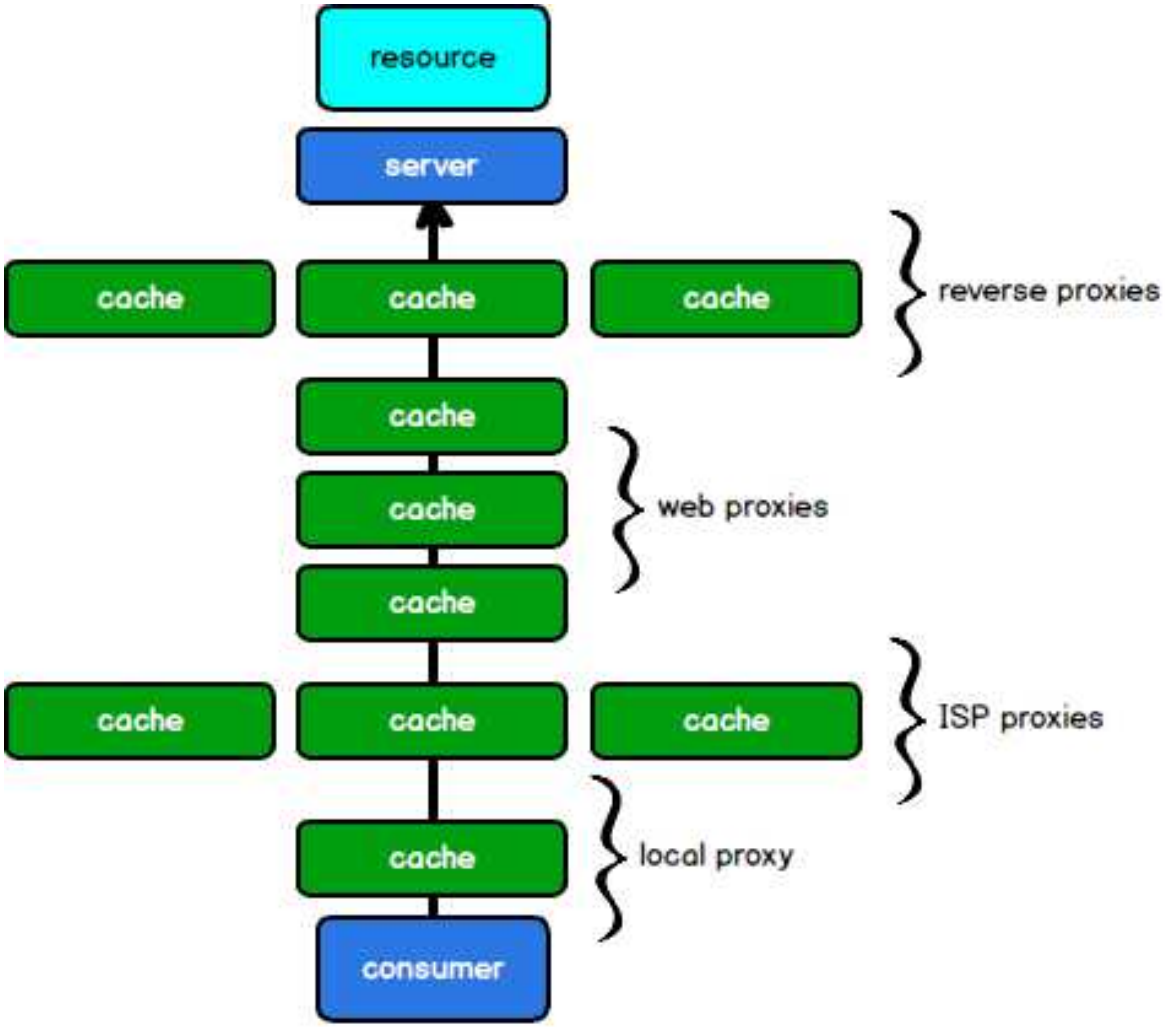
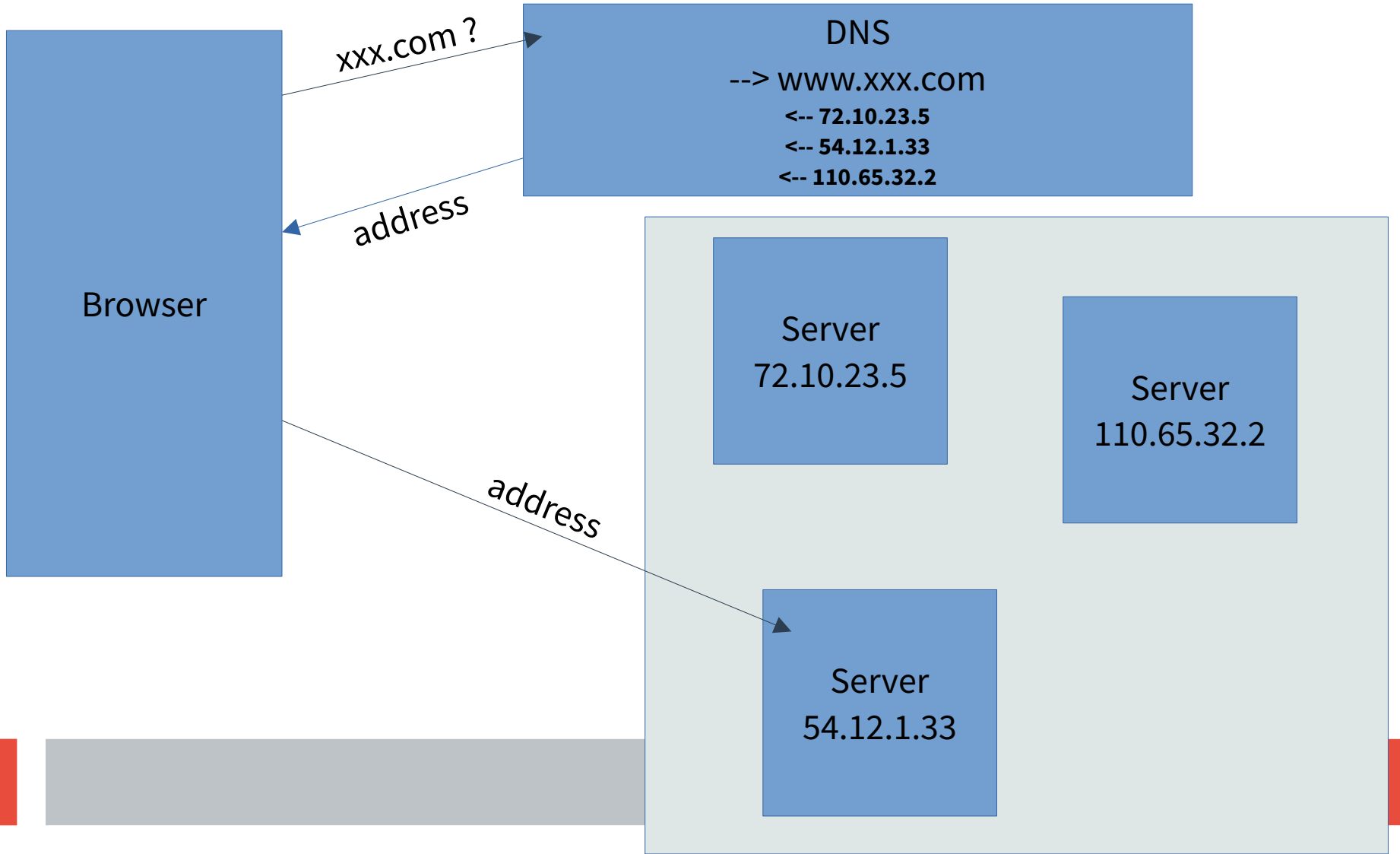
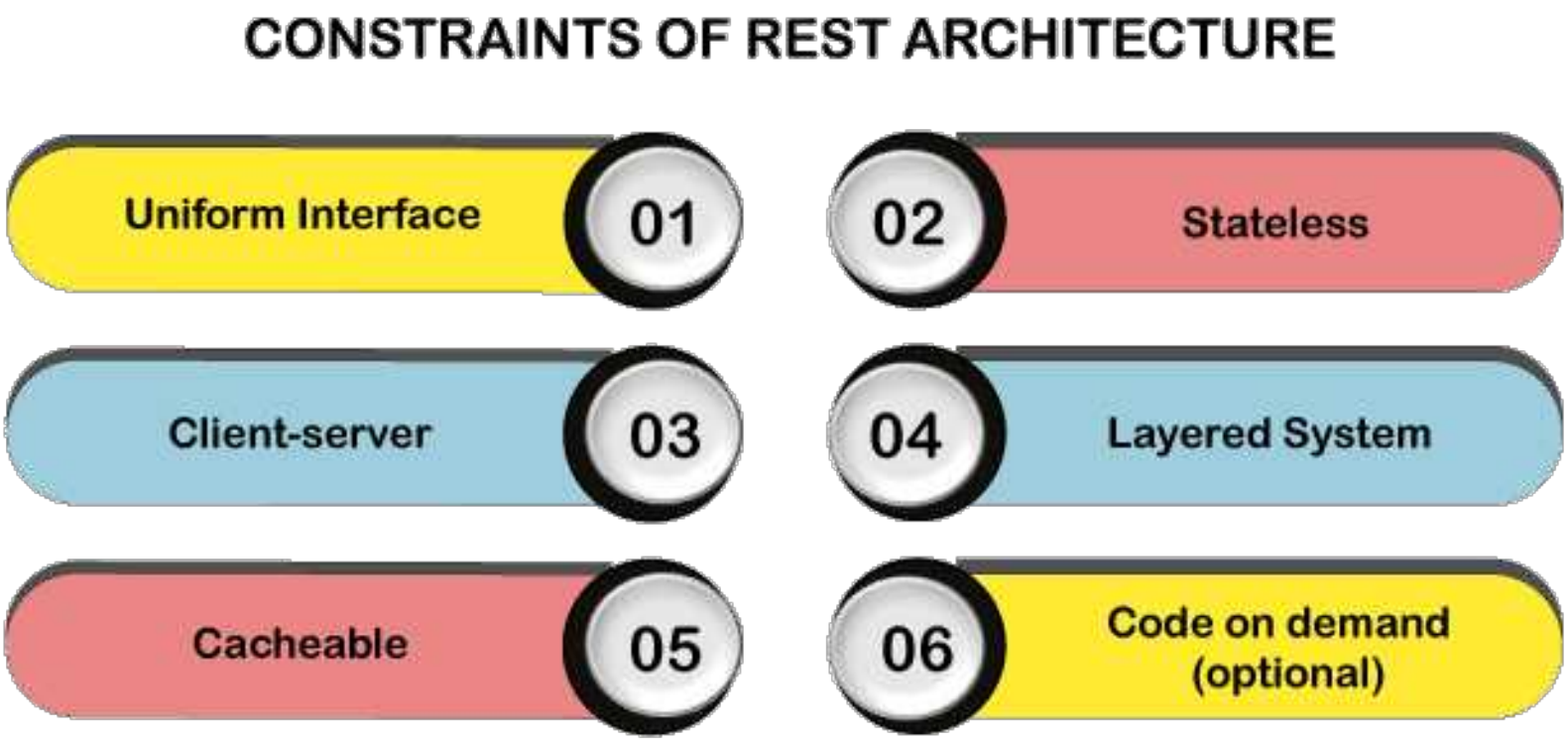



Figure 1. Web caches. REST In Practice, 2010.

# DNS


Browser
xxx.com ?
address
DNS
--> www.xxx.com
<-- 72.10.23.5
<-- 54.12.1.33
<-- 110.65.32.2
address
Server
72.10.23.5
Server
110.65.32.2
Server
54.12.1.33

# Constraint Stile Architetturale REST



# BitBucket

## Lista Commit di un repository

```
GET /rest/api/1.0/projects/{projectKey}/repos/{repoSlug}/commits
```

## Lista commenti Pull Request filtrando quelli con action=="COMMENTED"

```
GET /rest/api/latest/projects/{project}/repos/{repo}/pull-requests/{prId}/
activities
```

## Elencare permessi repository (cloud)

```
GET /repositories/{workspace}/{repo_slug}/permissions-config/users
```

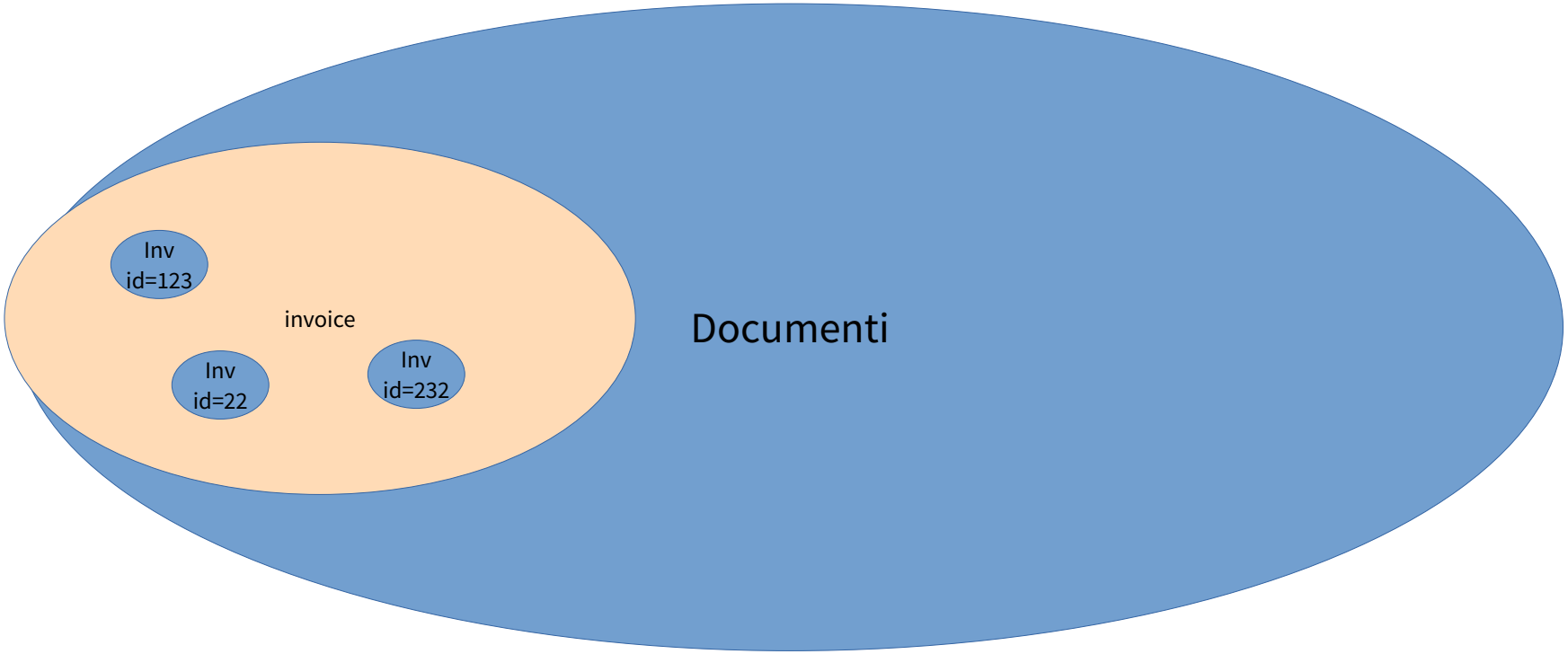
# BitBucket

## Lista Commit di un repository (ritorna un JSON)

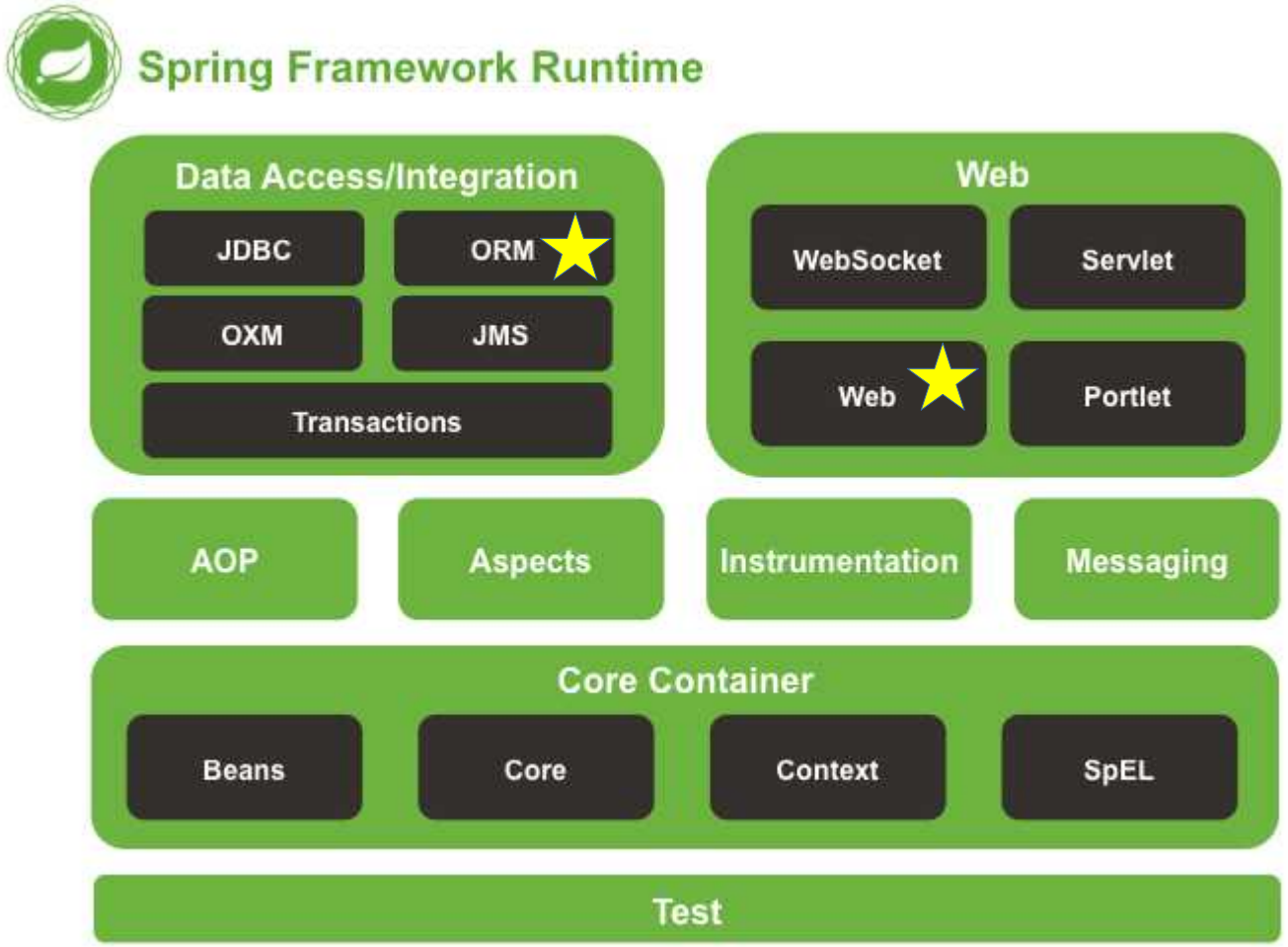
```
curl -u USER:PASS \
  "https://your-bitbucket-server/rest/api/1.0/projects/PROJ/repos/my-repo/commits?until=master"
```


```
{
  "size": 2,
  "limit": 25,
  "isLastPage": true,
  "values": [
    {
      "id": "01f9c86...",
      "displayId": "01f9c86",
      "author": { "name": "Alice", "emailAddress": "alice@example.com" },
      "message": "Fix bug XYZ",
      "parents": [ { "id": "abcdef123" } ]
    },
    { /* altro commit */ }
  ],
  "start": 0,
  "nextPageStart": 2
}
```

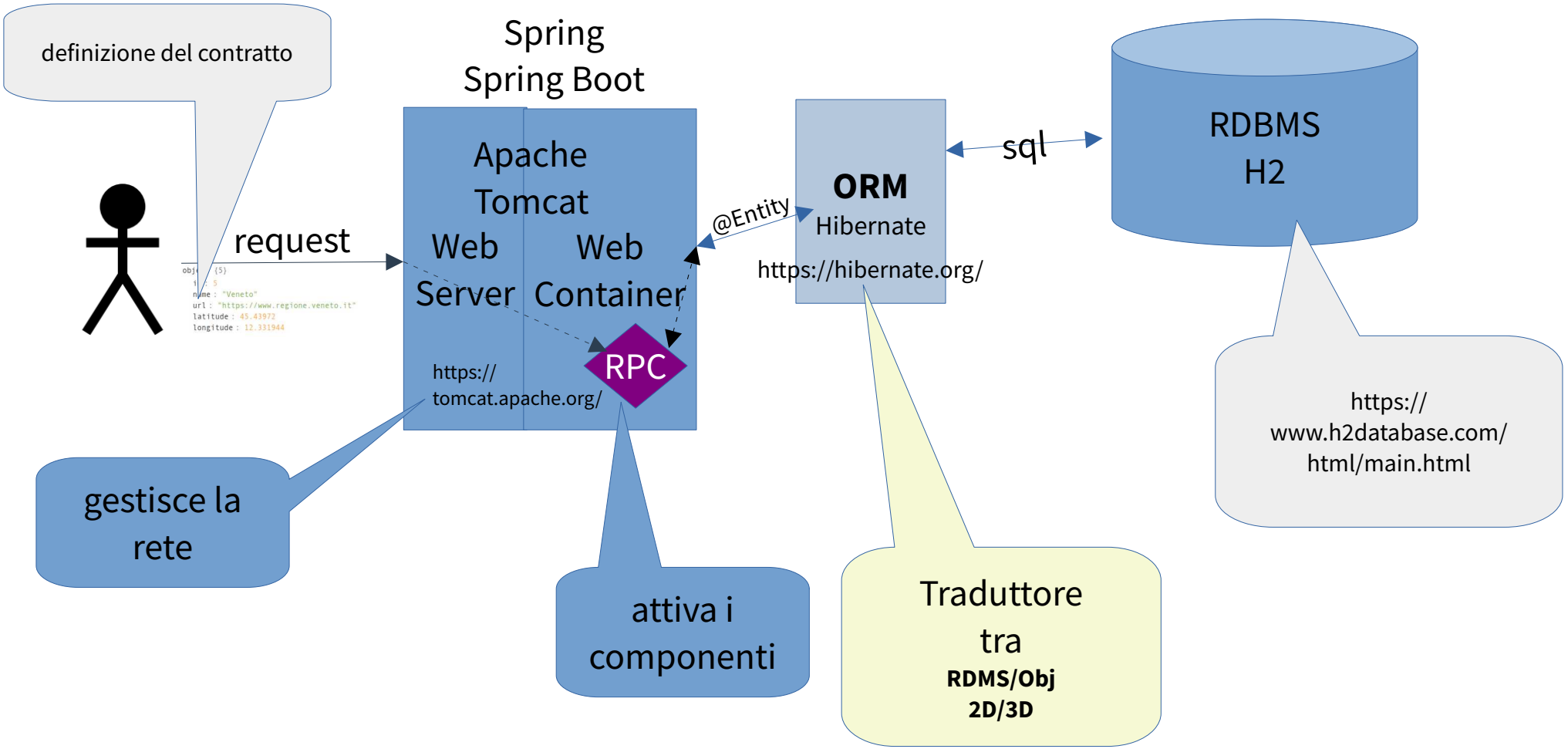
# Classes o tipi & valori



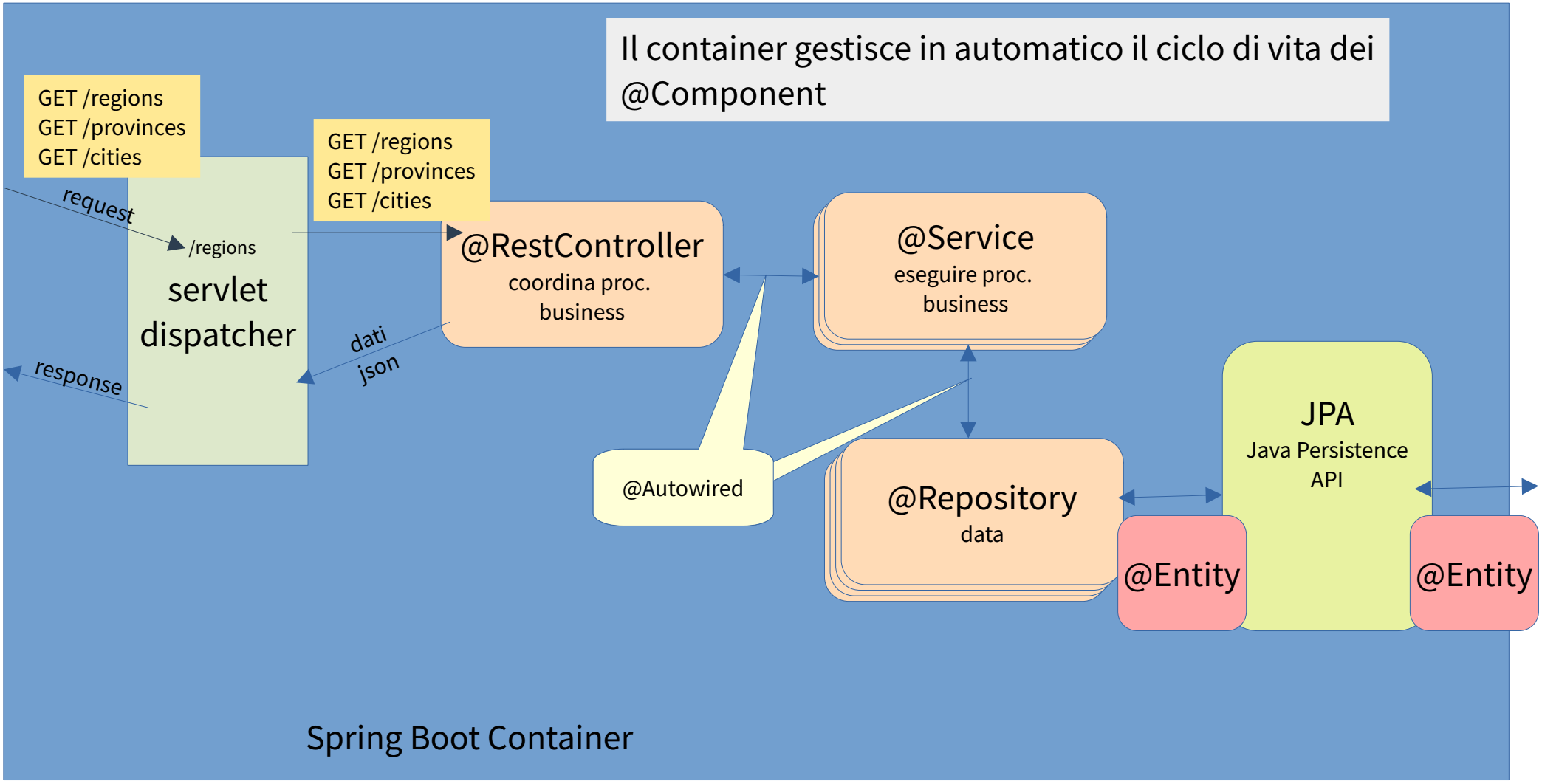
# Server REST Backend Java+Spring(boot)



# Spring Boot Application



# Application in SpringBoot



# RDBMS VS Objects

```
@Column(name = "codice_citta_metropolitana") // visione lato rdbms
```

```
private String codiceCittaMetropolitana; // visione lato OOP
```

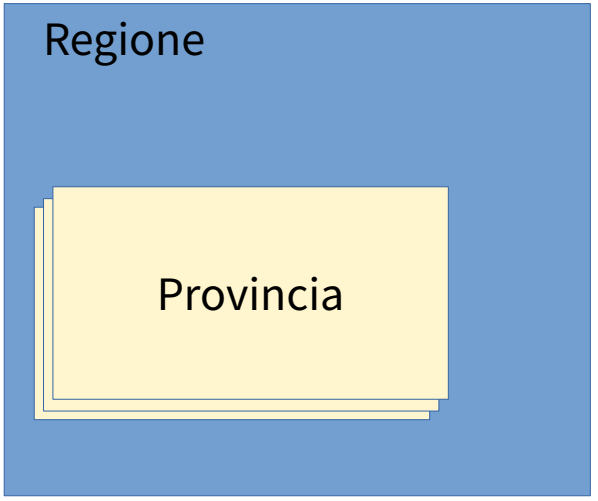
SELECT * FROM province;

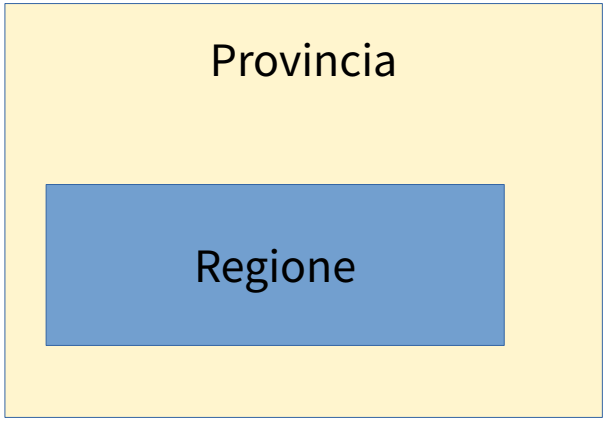
| ID | ID_REGIONE | CODICE_CITTA_METROPOLITANA | NOME | SIGLA_AUTOMOBILISTICA | LATITUDINE | LONGITUDINE |
|---|---|---|---|---|---|---|
| 1 | 1 | 201 | Torino | TO | 45.063299 | 7.669289 |
| 2 | 1 | *null* | Vercelli | VC | 45.320220 | 8.418508 |
| 3 | 1 | *null* | Novara | NO | 45.548513 | 8.515079 |
| 4 | 1 | *null* | Cuneo | CN | 44.597031 | 7.611422 |
| 5 | 1 | *null* | Asti | AT | 44.900765 | 8.206432 |
| 6 | 1 | *null* | Alessandria | AL | 44.817559 | 8.704663 |

SELECT * FROM REGIONI;

| ID | NOME | URL | LATITUDINE | LONGITUDINE |
|---|---|---|---|---|
| 1 | Piemonte | https://www.regione.piemonte.it | 45.066666 | 7.7 |
| 2 | Valle d'Aosta/Vallée d'Aoste | https://www.regione.vda.it | 45.73722 | 7.320556 |
| 3 | Lombardia | https://www.regione.lombardia.it | 45.46416 | 9.190336 |
| 4 | Trentino-Alto Adige/Südtirol | https://www.regione.taa.it | 46.066666 | 11.116667 |
| 5 | Veneto | https://www.regione.veneto.it | 45.43972 | 12.331944 |
| 6 | Friuli Venezia Giulia | https://www.regione.fvg.it | 45.636112 | 13.804167 |

2D




3D

# Al boot dell'applicazioni

https://start.spring.io/


**HTTP Methods**

GET => `@GetMapping`

POST => `@PostMapping`

PUT => `@PutMapping`

DELETE => `@DeleteMapping`

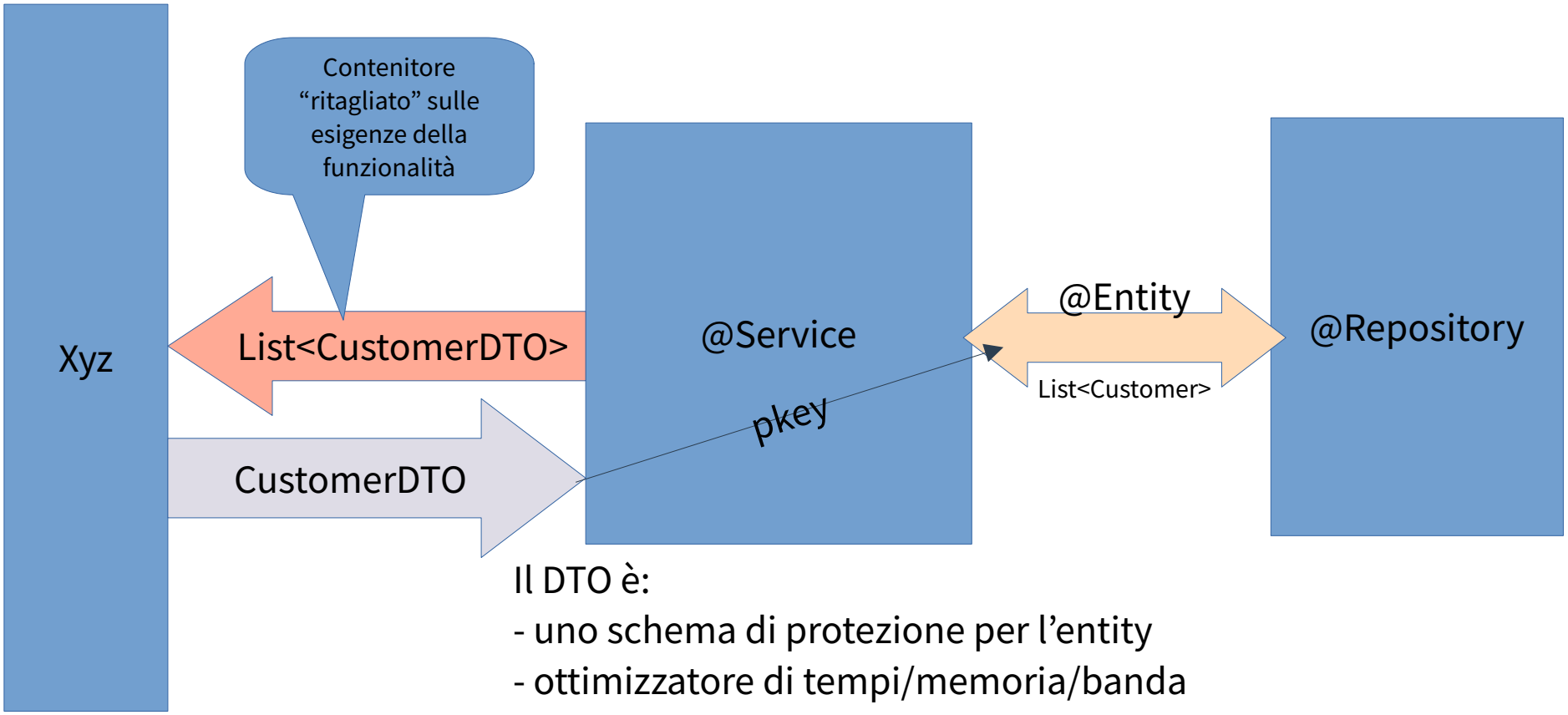
PATCH => `@PatchMapping`

1) match tra urlpath e RequestMapping
onMatch=>attiva
onNoMatch=>404

2) match tra Http Method e @XyzMapping
onMatch=>attiva
onNoMatch=>404

3) match tra Http Method Url e URL di @XyzMapping
onMatch=>attiva
onNoMatch=>404

# DTO pattern (Data Transfer Object)




Il DTO è:
- uno schema di protezione per l'entity
- ottimizzatore di tempi/memoria/banda
- foundation per il JSON di scambio

Il DTO è associato ad un mapper che converte da e verso l'entity

# News in V2

- new ApiError

- repo method findByNomeContainingIgnoreCase(String nome) di RegionRepository, ProvinceRepository, CityRepository (query added)

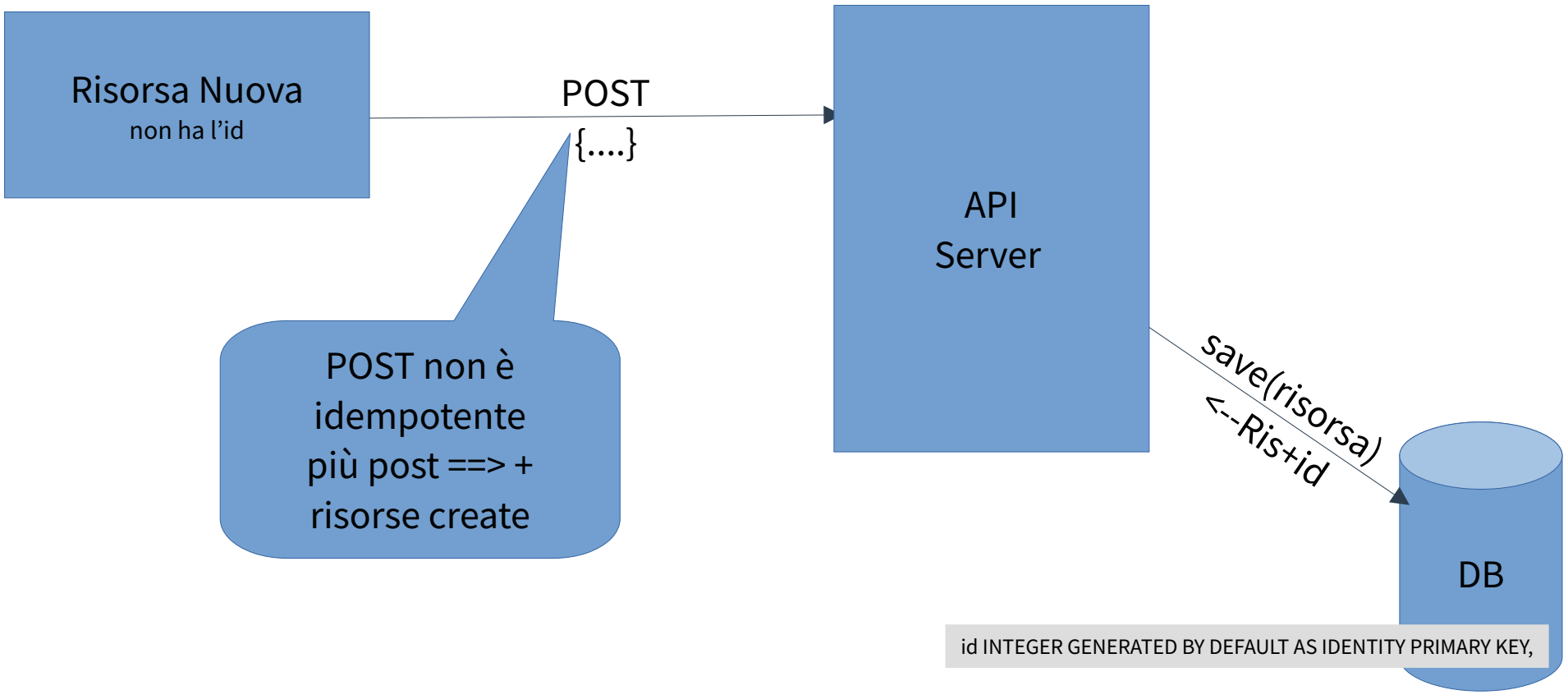
- RegionController, ProvinceController, CityController per aggiunta metodo di filtraggio e “query string”

```
@GetMapping
public List<CityDTO> getAllCities(
        @RequestParam(name = "regionId", required = false) Integer regionId,
        @RequestParam(name = "provinceId", required = false) Integer provinceId,
        @RequestParam(name = "name", required = false) String name) {
    return cityService.getCities(regionId, provinceId, name);
}
```

# API: cosa serve? Lista della spesa.

- progettare la semantica (significato delle API), tenendo presente il linguaggio “funzionale” (DDD ubiquitus language), non il linguaggio del DB.
- Progettare il contratto: definire la struttura del JSON+DTO (o dell’XML) da usare nello “scambio dei dati”, ovvero l’interfaccia fornita dai servizi. [il DTO viene serializzato in JSON e deserializzato da JSON]
- Progettare il contratto relativo al messaging: definire la struttura del JSON (o dell’XML) da usare nella segnalazione degli errori (Possibilmente risolto dagli standard aziendali).

# POST e Creazione risorse



# Navigazione con limit

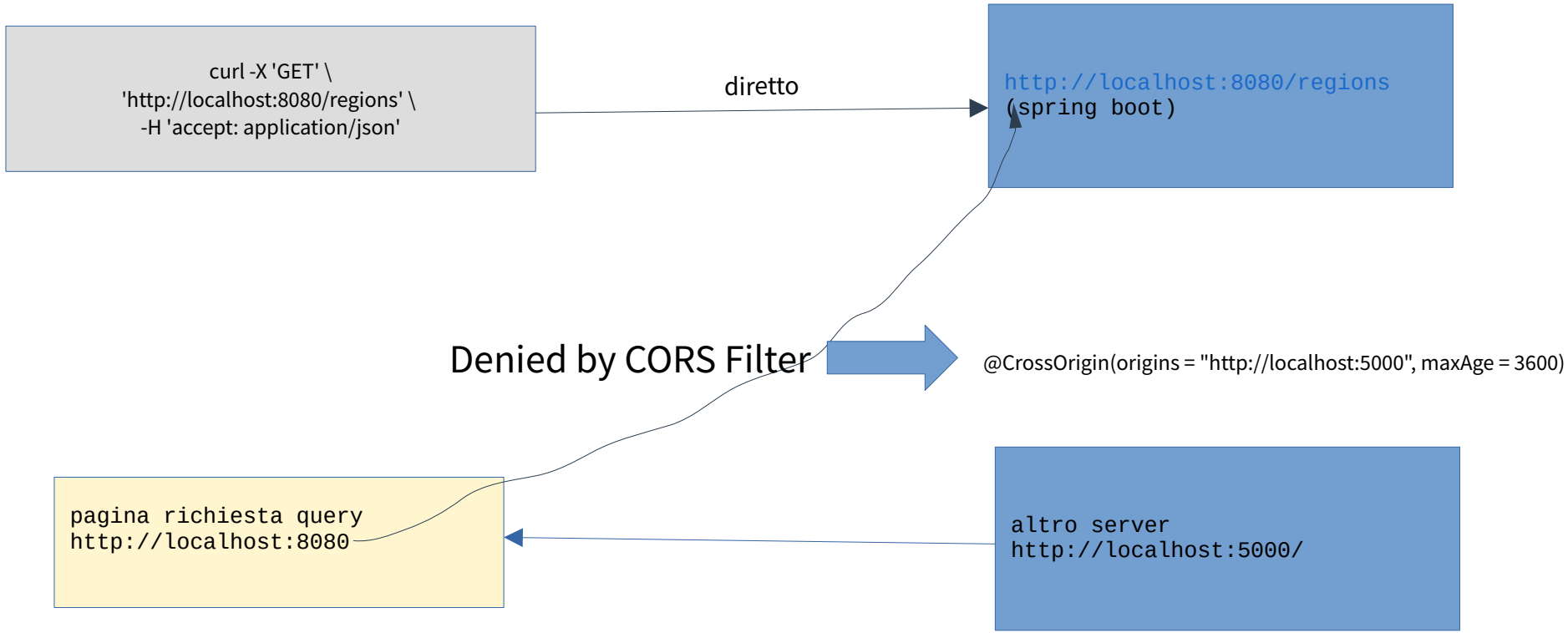
a) prima volta ==> /products?cursor=abc123&limit=50

b) altre volte /products?cursor=abc123&limit=50&productId=xyz
dove productId è l'ultimo id dei prodotti ricevuti:
Il server calcola

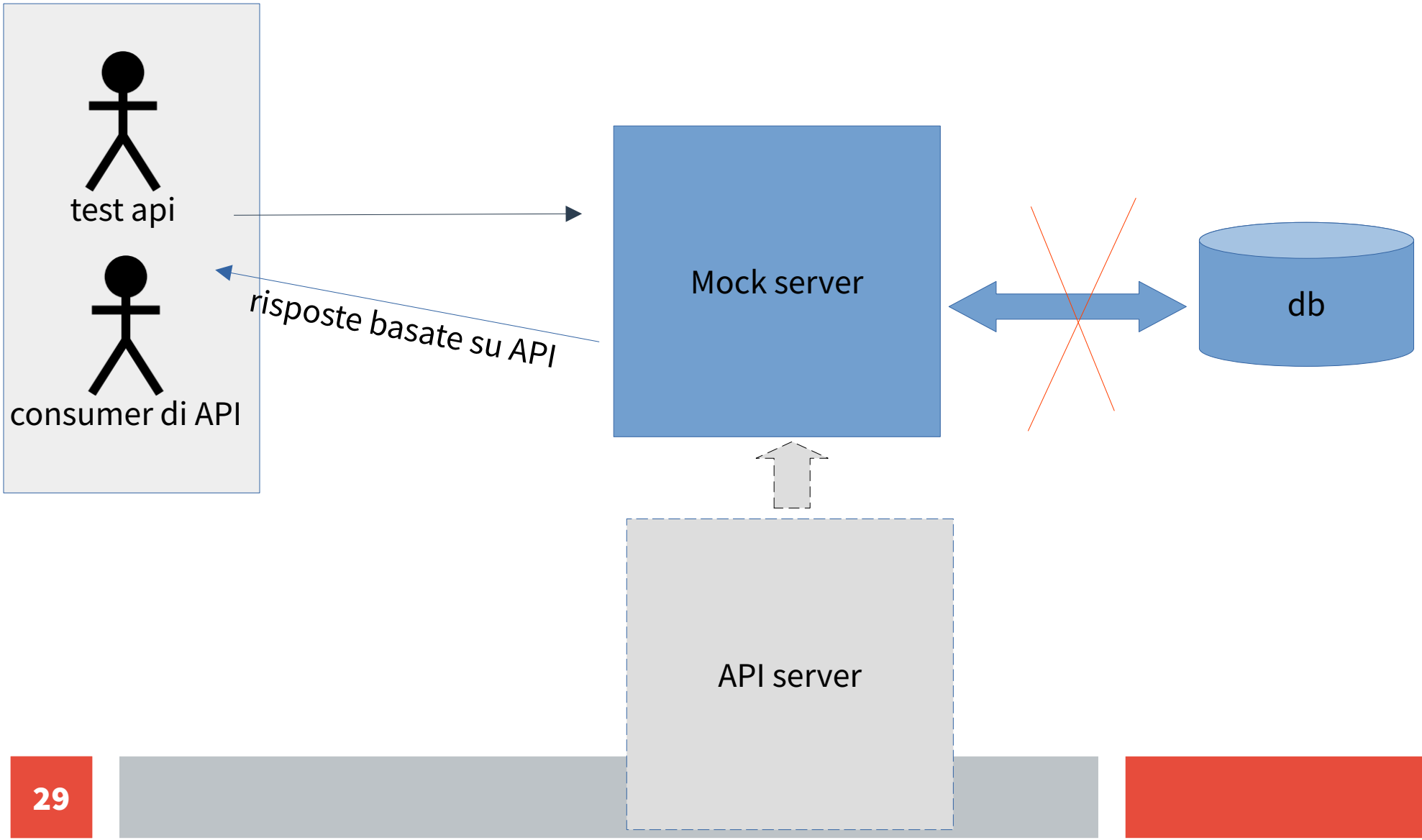
in avanti
select * from products where productId>xyz limit 50

all'indietro
select * from products where productId<=xyz limit 50

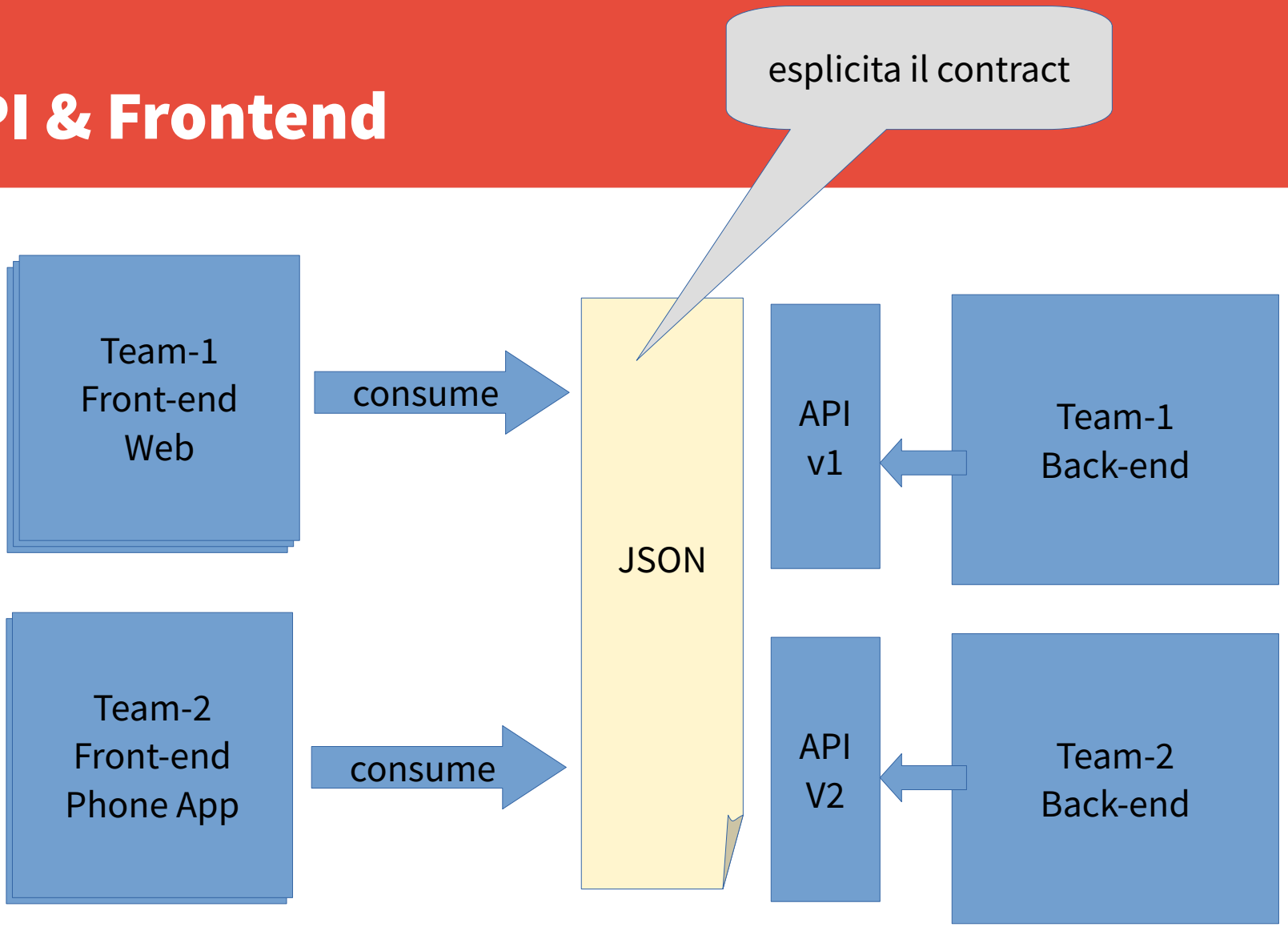
# CORS


curl -X 'GET' \
'http://localhost:8080/regions' \
-H 'accept: application/json'
diretto
http://localhost:8080/regions
(spring boot)
Denied by CORS Filter
@CrossOrigin(origins = "http://localhost:5000", maxAge = 3600)
pagina richiesta query
http://localhost:8080
altro server
http://localhost:5000/

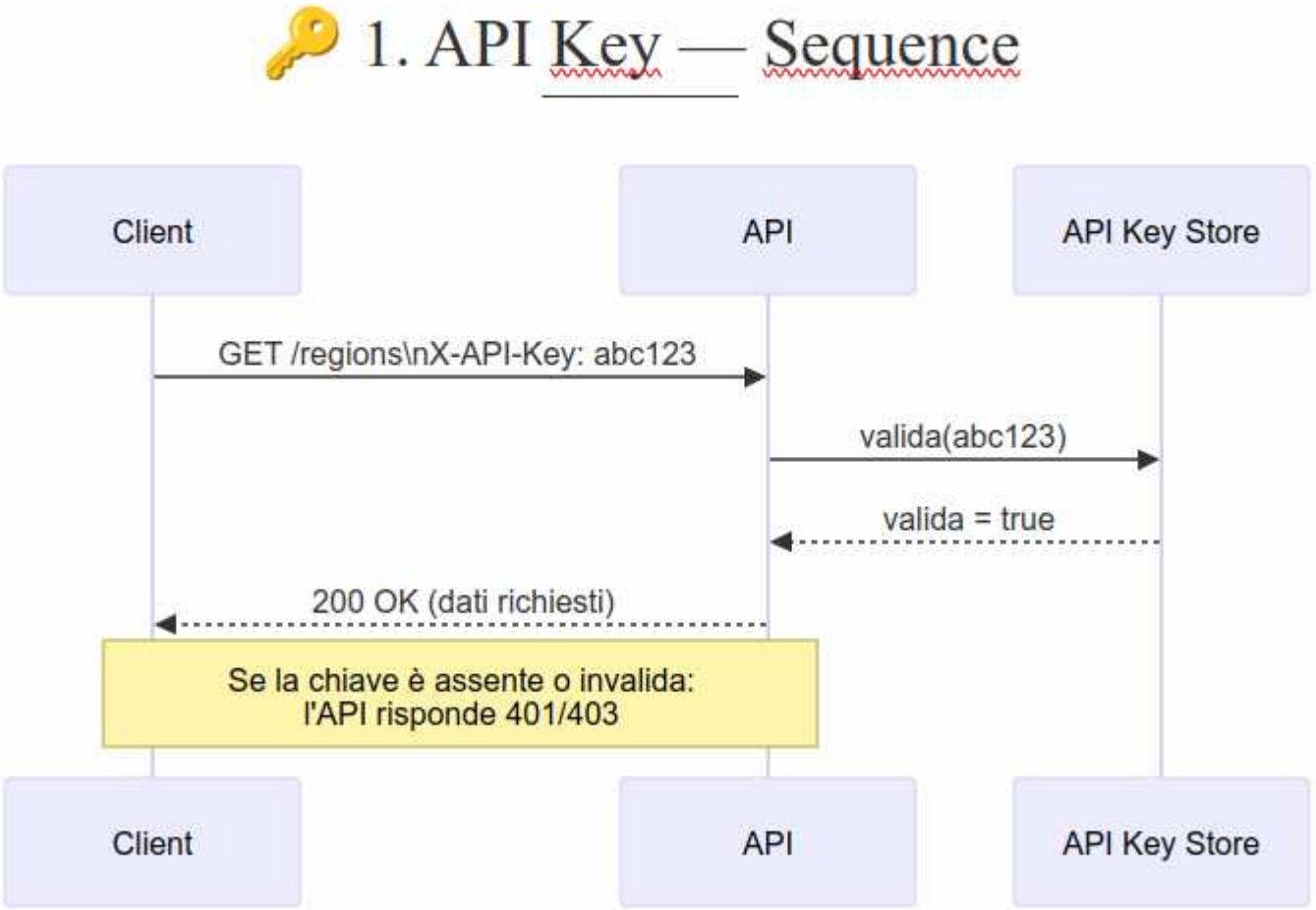
# Mock Server


test api
consumer di API
risposte basate su API
Mock server
db
API server

# API & Frontend



# AUTENTICAZIONE api-key



# API-KEY HTTP headers spec

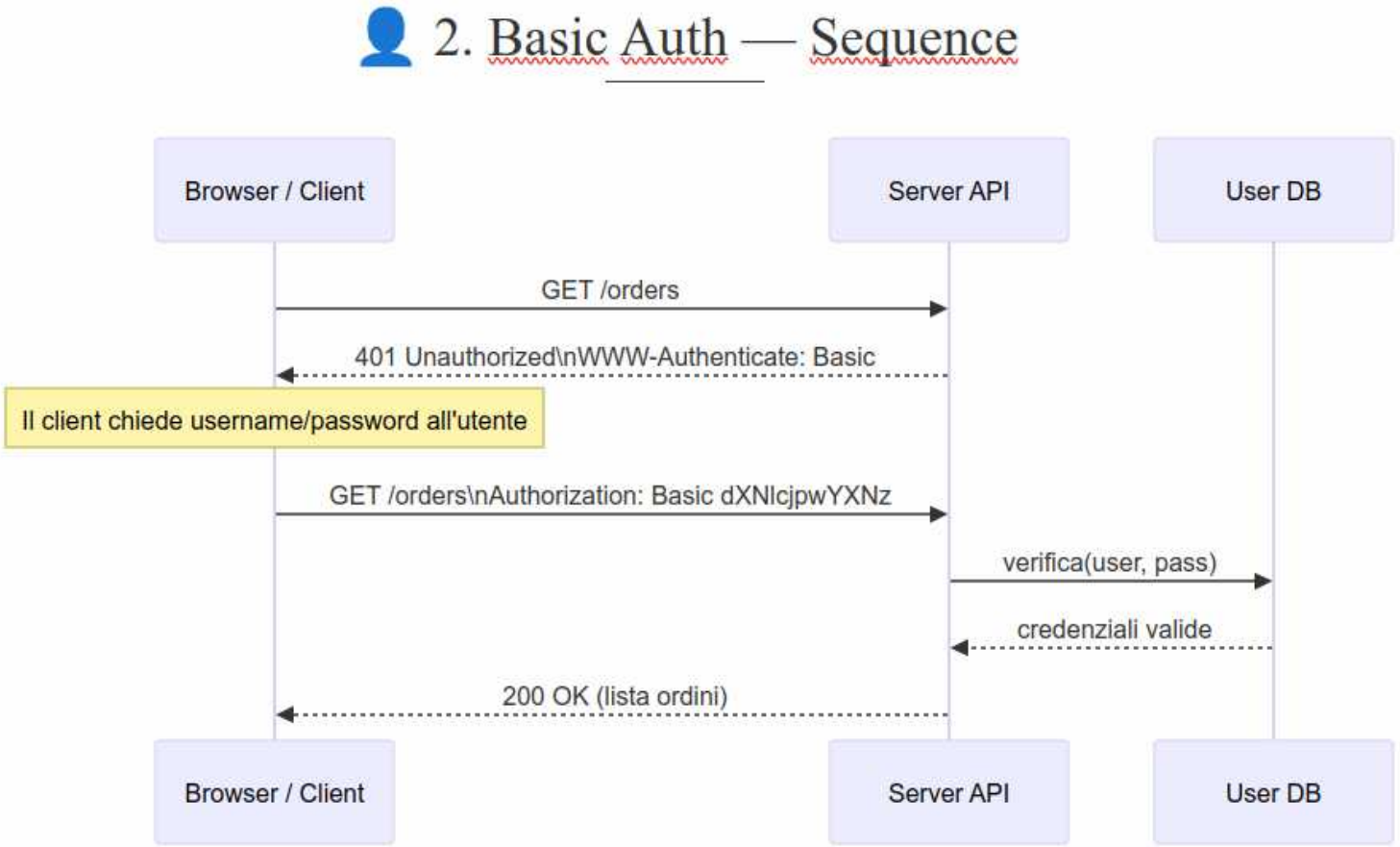
```
curl -v -H "X-API-KEY: demo-key-123" "http://localhost:8080/provinces"

* Host localhost:8080 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:8080...
* Connected to localhost (::1) port 8080
* using HTTP/1.x
> GET /provinces HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.15.0
> Accept: */*
> X-API-KEY: demo-key-123
>
* Request completely sent off
< HTTP/1.1 200
< X-Content-Type-Options: nosniff
< X-XSS-Protection: 0
< Cache-Control: no-cache, no-store, max-age=0, must-revalidate
< Pragma: no-cache
< Expires: 0
< X-Frame-Options: DENY
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Tue, 09 Dec 2025 14:42:00 GMT
<
[{"id":1,"name":"Torino","regionId":1,
```

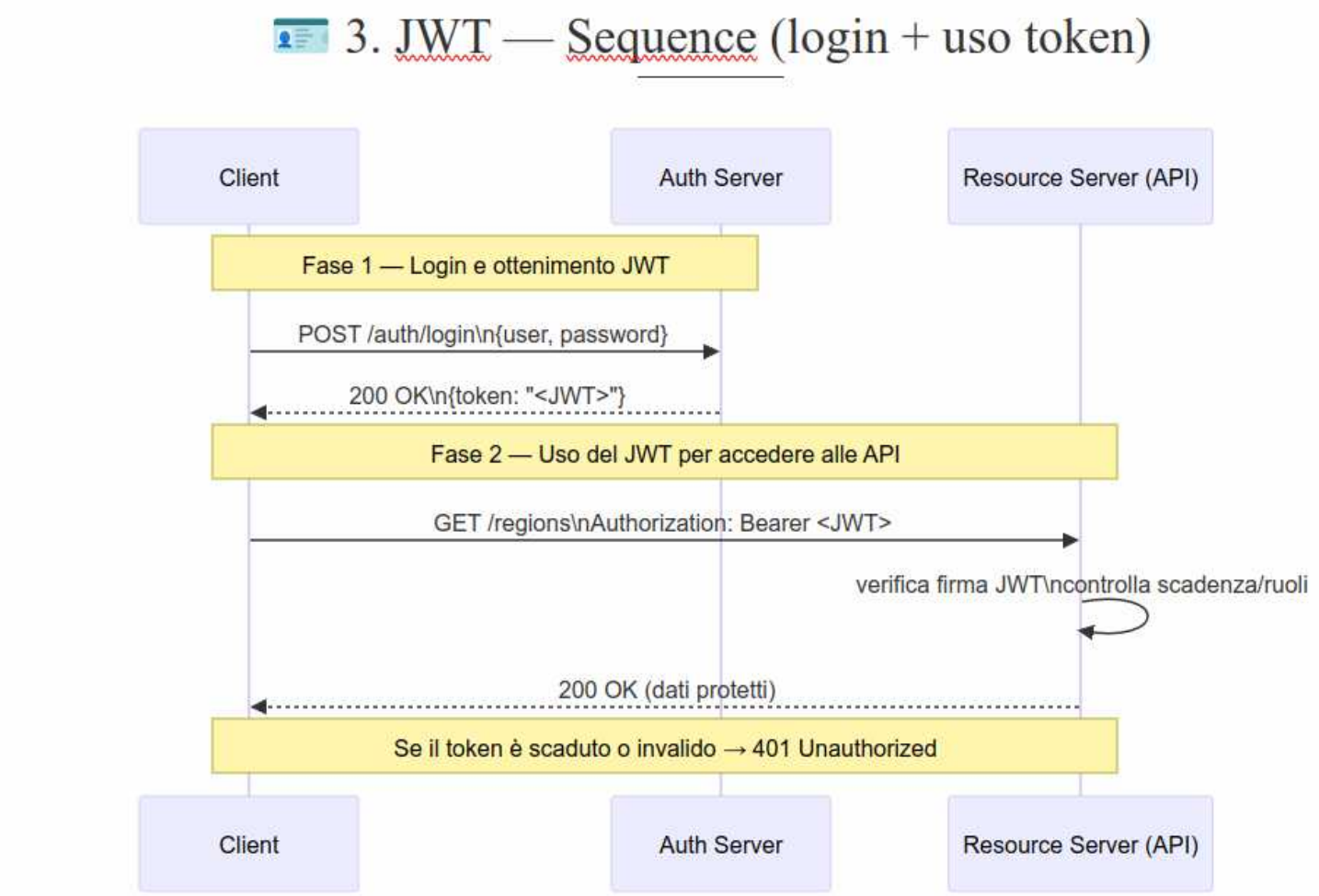
API KEY inviata dalla request

Body
json delle province

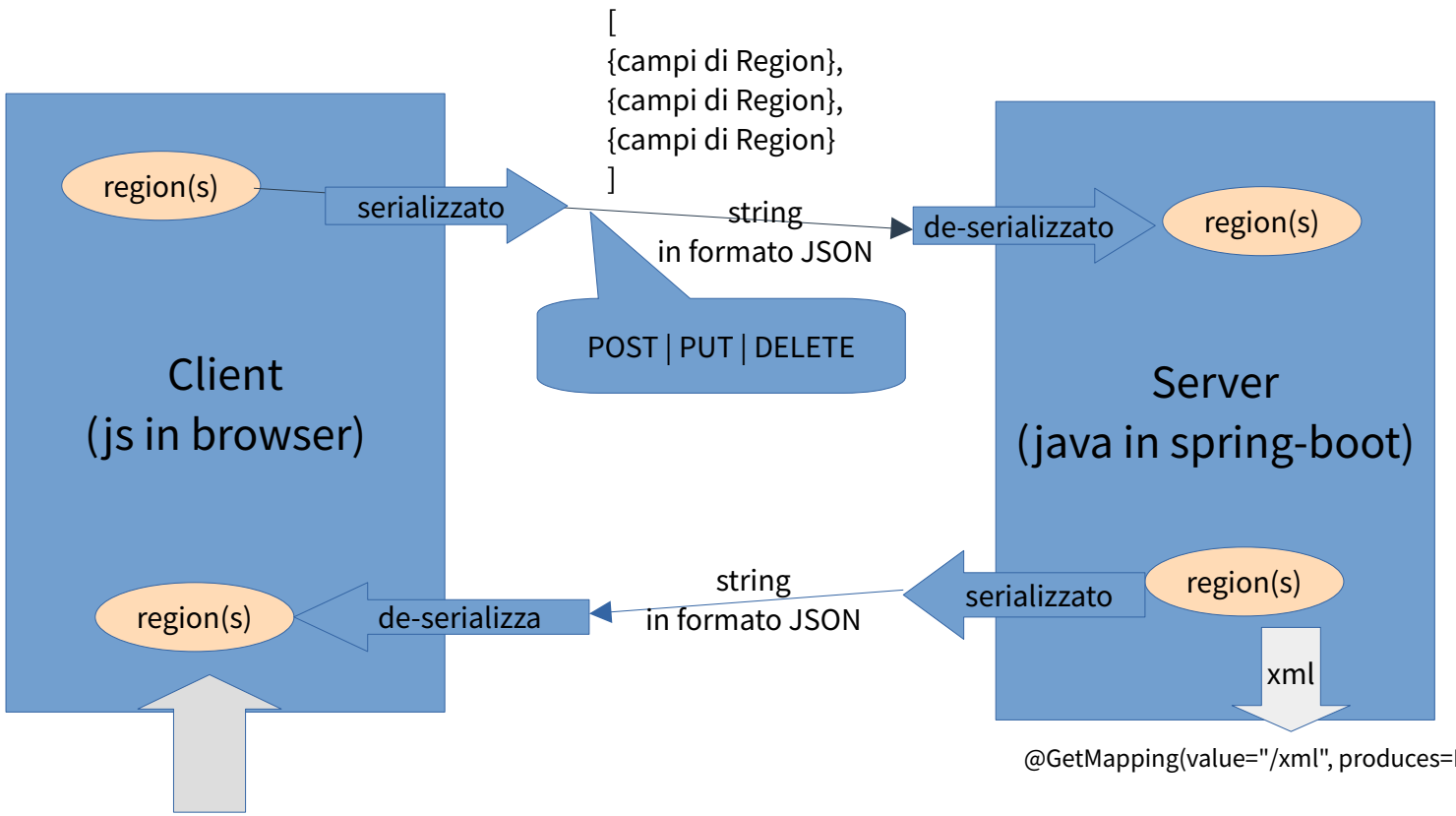
# Autenticazione basic



# Jwt (Json Web Token) autentication



# Serializzazione/Deserializzazione




@GetMapping(value="/xml", produces=MediaType.APPLICATION_XML_VALUE)

```
// de-serializza in un oggetto lo stream json
let obj = JSON.parse('{"id":1,"name":"Piemonte","url":"https://www.regione.piemonte.it"}');
print("nome="+obj.name);
```

# Cosa serve per provare

**- ambiente java**

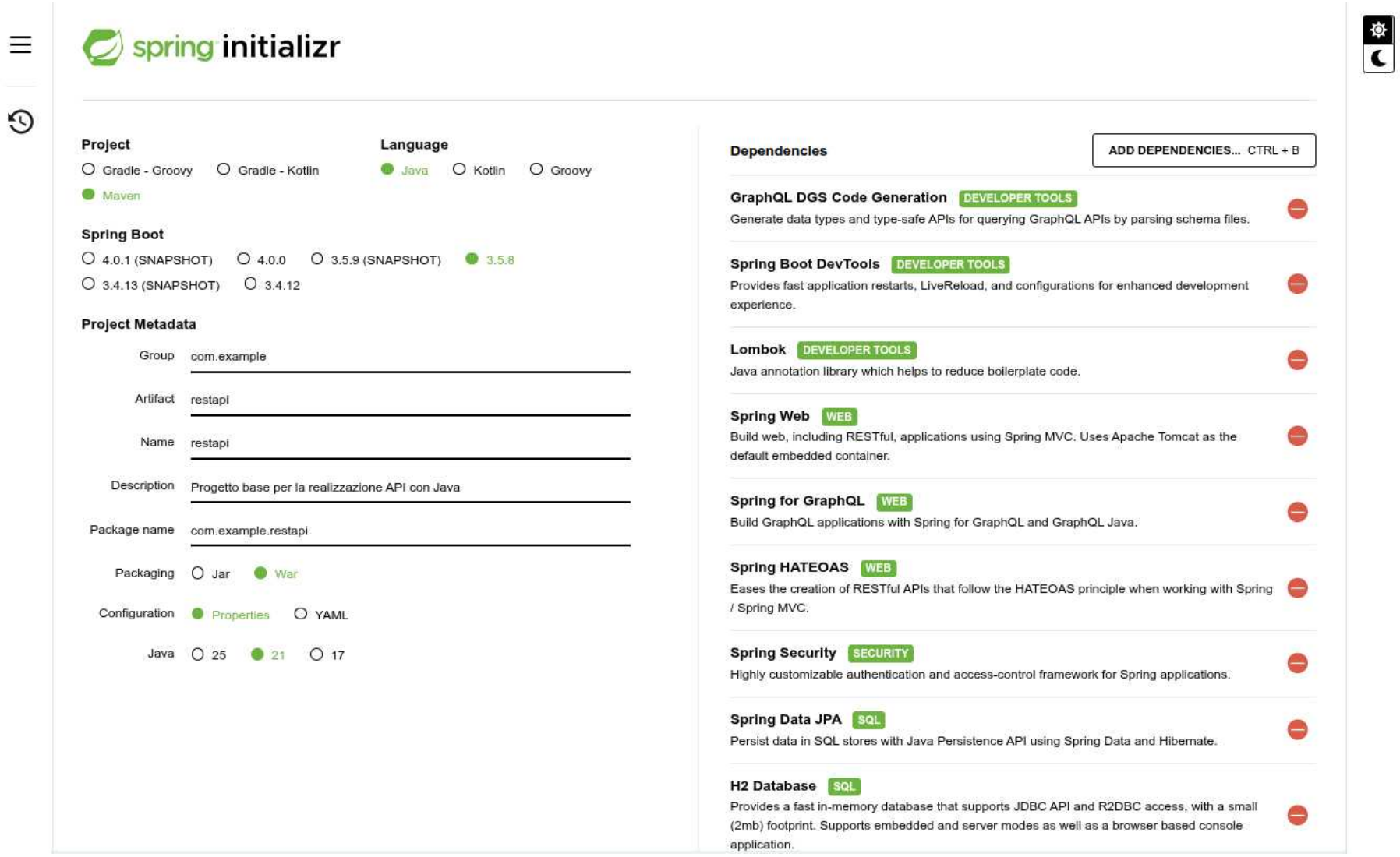
- java jdk (java.oracle.com)
- IDE es: eclipse da www.eclipse.org
- curl (da https://curl.se/windows/)
- browser

**- base per lo sviluppo**

- scheletro di progetto pronto start.spring.io (es: restapi.zip) da importare nell'IDE
- modificare application.properties
- struttura database schema.sql (struttura) data.sql (dati) da posizionare in <progetto>/src/main/resources

```
# Carica schema.sql e data.sql automaticamente
spring.sql.init.mode=always
spring.sql.init.schema-locations=classpath:schema.sql
spring.sql.init.data-locations=classpath:data.sql,.....
```

# start.spring.io



# Roadmap

1 – struttura db

2 – mappare le @Entity

3 – @Entity --> cosa mostrare --> dto

4 – @Repository (accesso ai dati)

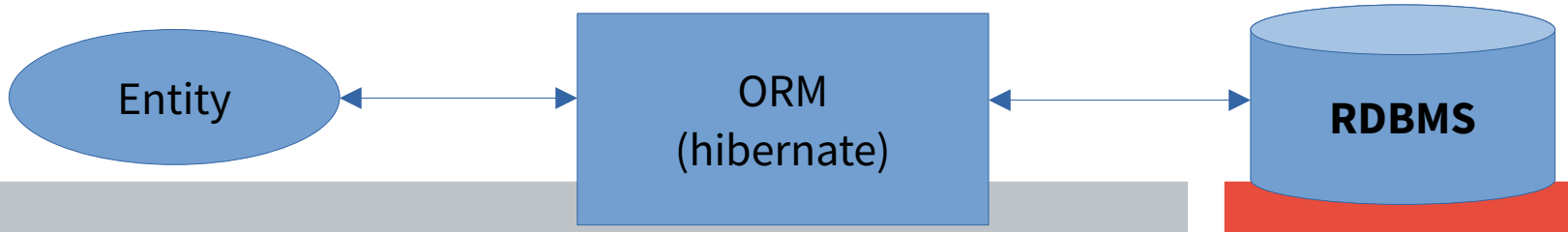
5 – @Service (logica di business)

6(0) – disegno delle API significato di business+path+parametri

7 - @Controller

8 – test

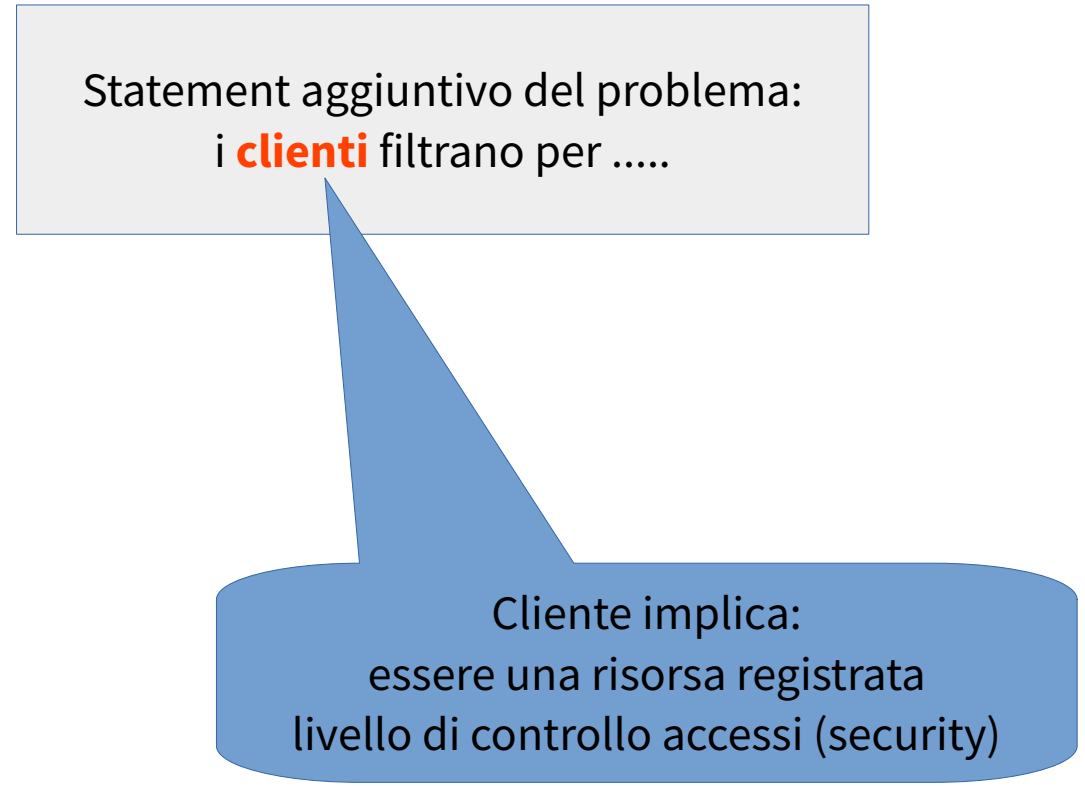
@Entity è il legame stretto tra la ns applicazione e l'ORM (Object Relational Mapper). Non usiamo più un RDBMS, ma un prodotto che da un lato fornisce oggetti e dall'altro si unisce al RDMBS.



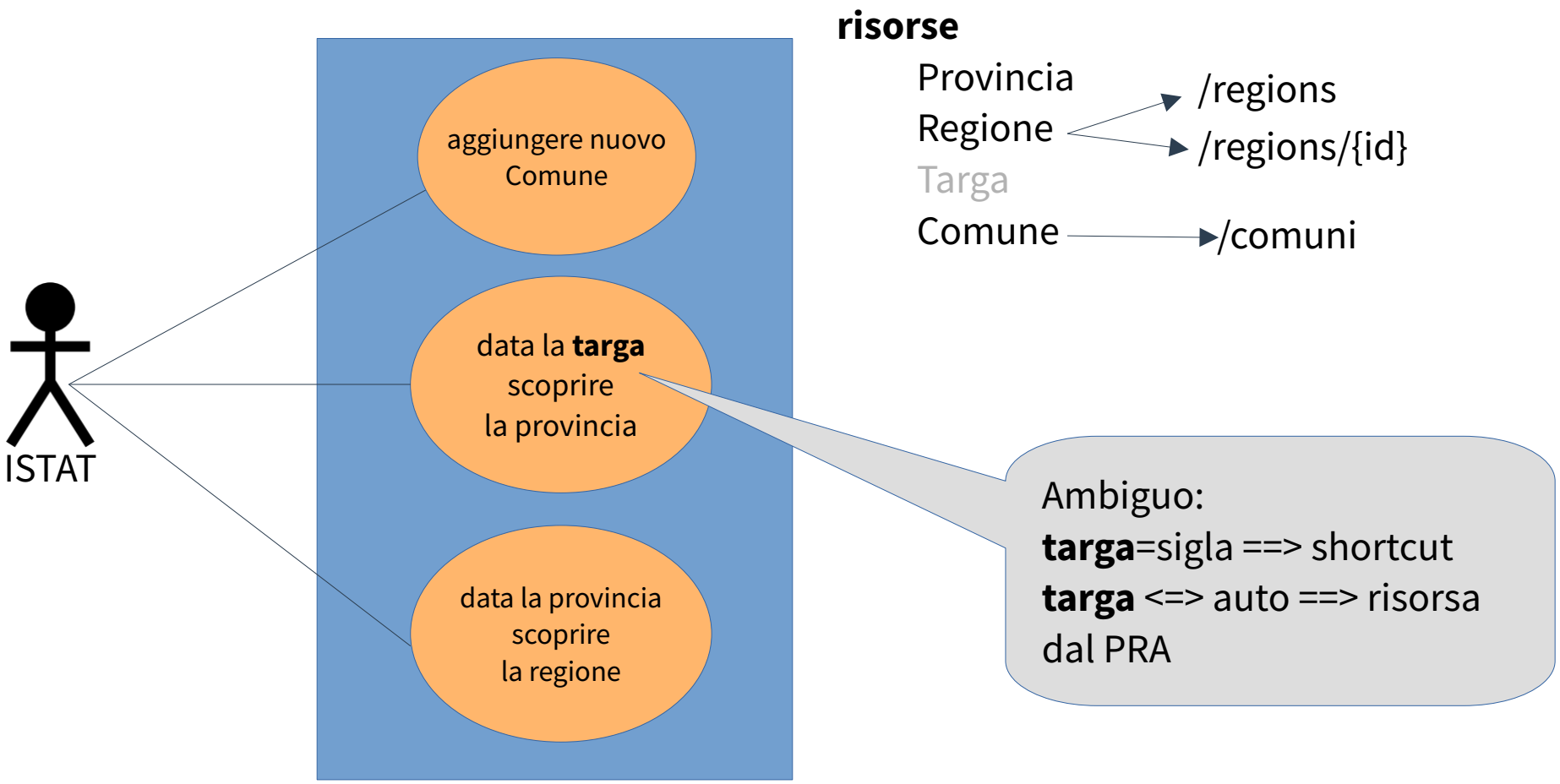
# Roadmap

**Progettare una piccola API design-first, usando quello che abbiamo visto:**

- **risorse**
- **URI**
- **metodi HTTP**
- **parametri**
- **pagination**
- **esempi**
- **OpenAPI (livello base)**



# Risorse esempio (italia=>regioni,province,comuni)

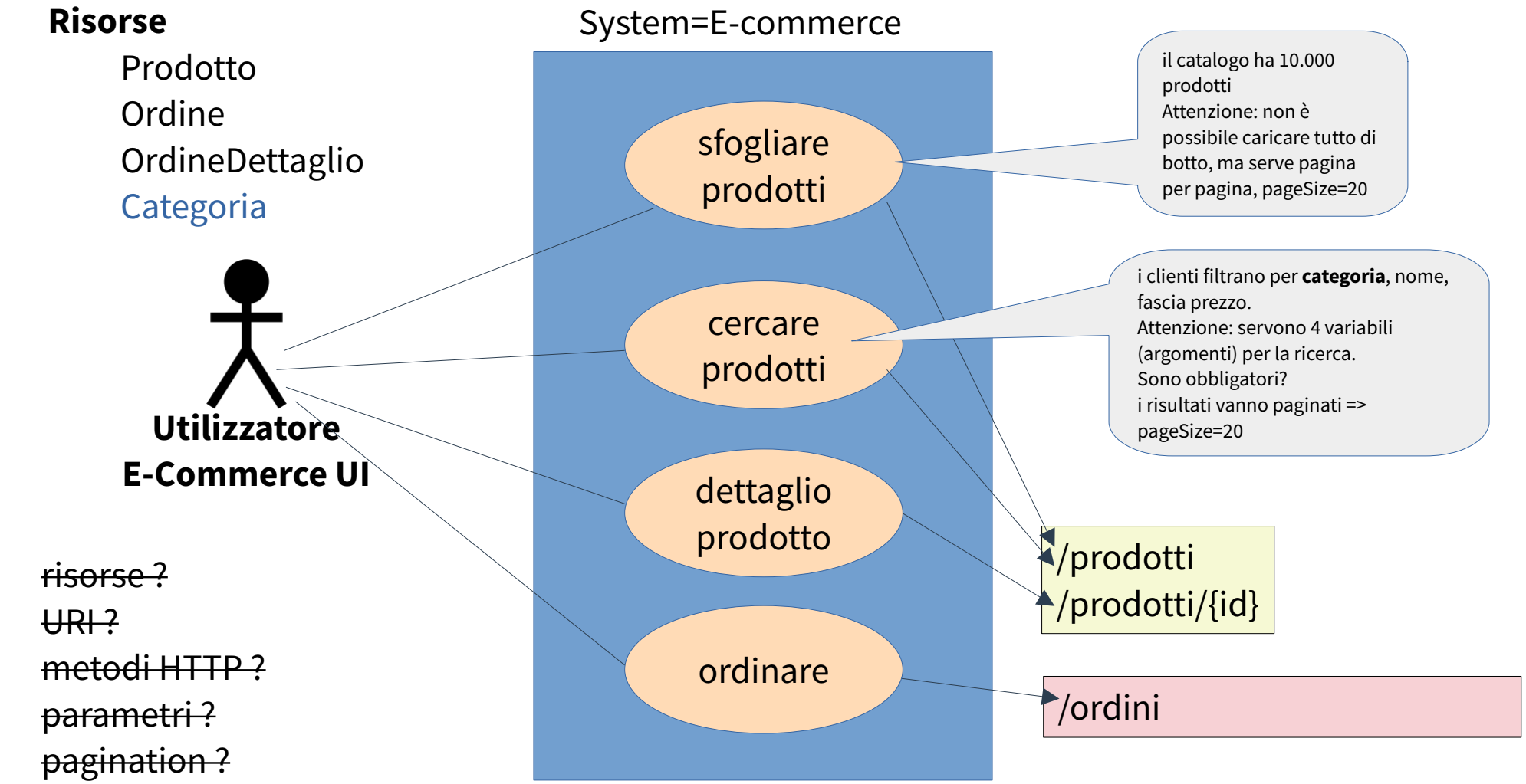


# E-commerce API
# UML Use Case Diagram

**Domande guida:**
A cosa serve questa realizzazione?
Quali parametri servono davvero al consumer?
Ci sono casi d'uso reali?

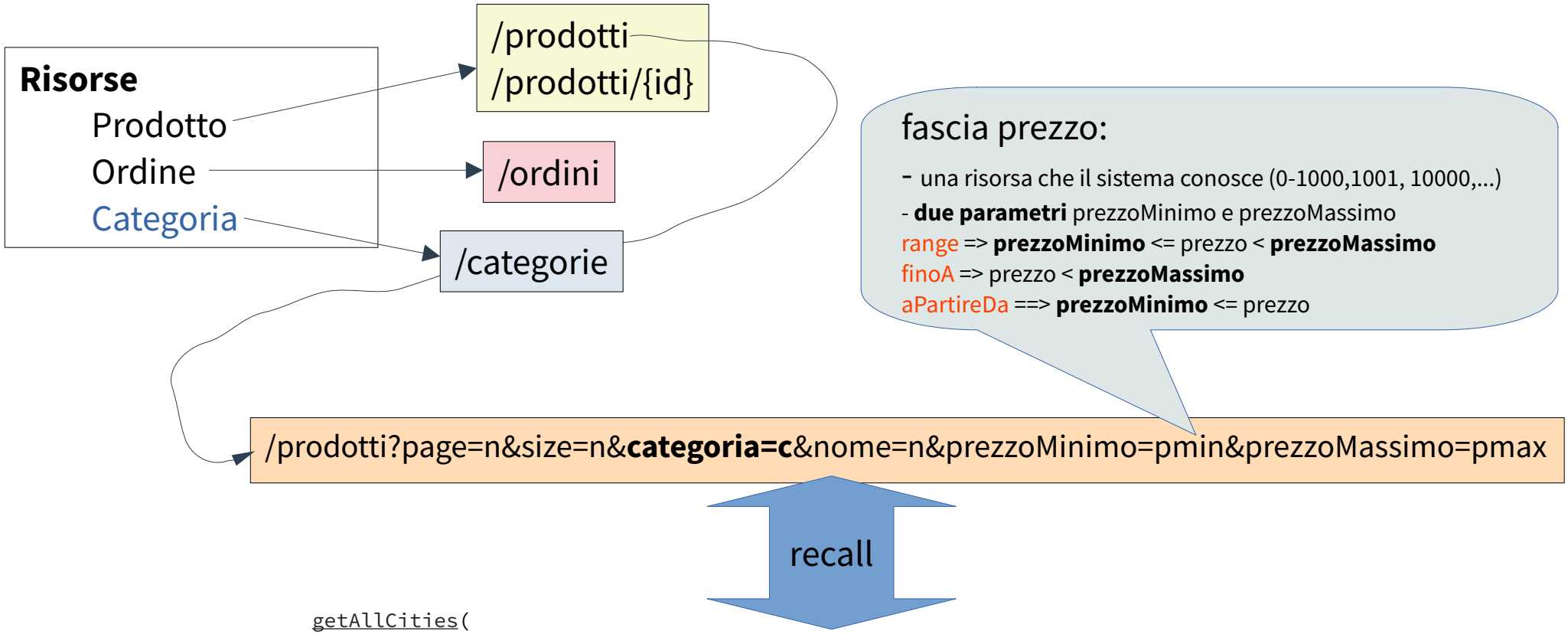
**Risorse**

- Prodotto
- Ordine
- OrdineDettaglio
- Categoria




~~risorse ?~~
~~URI ?~~
~~metodi HTTP ?~~
~~parametri ?~~
~~pagination ?~~

# Risorse




```
getAllCities(
            @RequestParam(name = "regionId", required = false) Integer regionId,
            @RequestParam(name = "provinceId", required = false) Integer provinceId,
            @RequestParam(name = "name", required = false) String name,
            @RequestParam(name = "page", required = false) Integer pageParam,
            @RequestParam(name = "size", required = false) Integer sizeParam)
```

# URI

GET /prodotti?page=n&size=n&categoria=c&nome=n&prezzoMinimo=pmin&prezzoMassimo=pmax
GET /prodotti/{id}

GET /ordini?page=n&size=n&clienteId=cid
GET /ordini/{id}
POST /ordini
DELETE /ordini/{id}
PUT /ordini/{id}
PATCH /ordini/{id}

GET /categorie
GET /categorie{id}

**TODO-LIST**
per le GET che ritornano liste
occorre definire il criterio di sort

Manca il sorting, per i prodotti potrebbe essere
**sort=price,asc**
Mancano default per paging
**size = 20**
**max size = 100**

**GET** lettura
**POST** crea nuova risorsa
**PUT** modifica tutta la risorsa
**PATCH** modifica parzialmente la risorsa
**DELETE** elimina risorsa

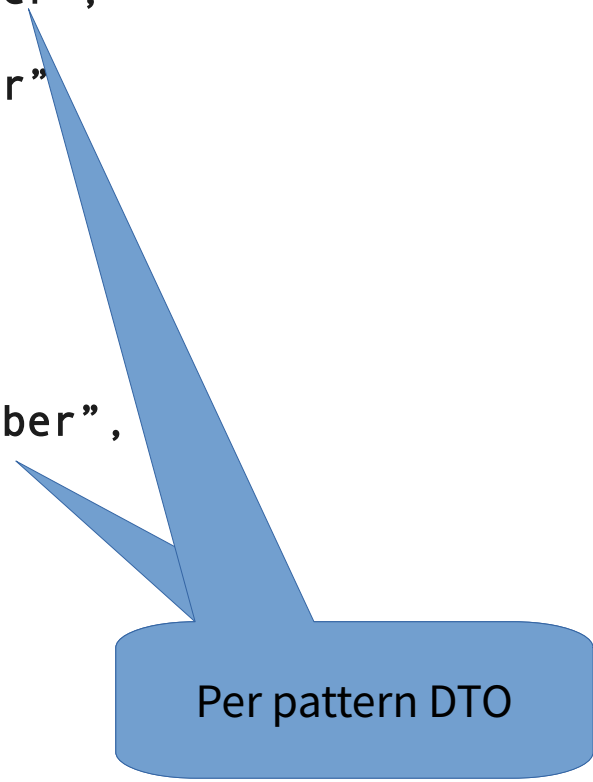
# JSON delle risorse Prodotto e Categoria

```
Prodotto
{
“idProdotto” : “number”,
“nome” : “string”,
“categoria” : “number”,
“prezzo” : “number”,
“stock” : “number”
}

Categoria
{
“idCategoria” : “number”,
“nome” : “string”
}
```

```
ProdottoResponse
{
“idProdotto” : “number”,
“nome” : “string”,
“categoria” : “number”
“prezzo” : “number”,
“stock” : “number”
}

CategoriaResponse
{
“idCategoria” : “number”,
“nome” : “string”
}
```




**CONTRATTO**

# JSON delle risorse Ordine (segue swagger editor)

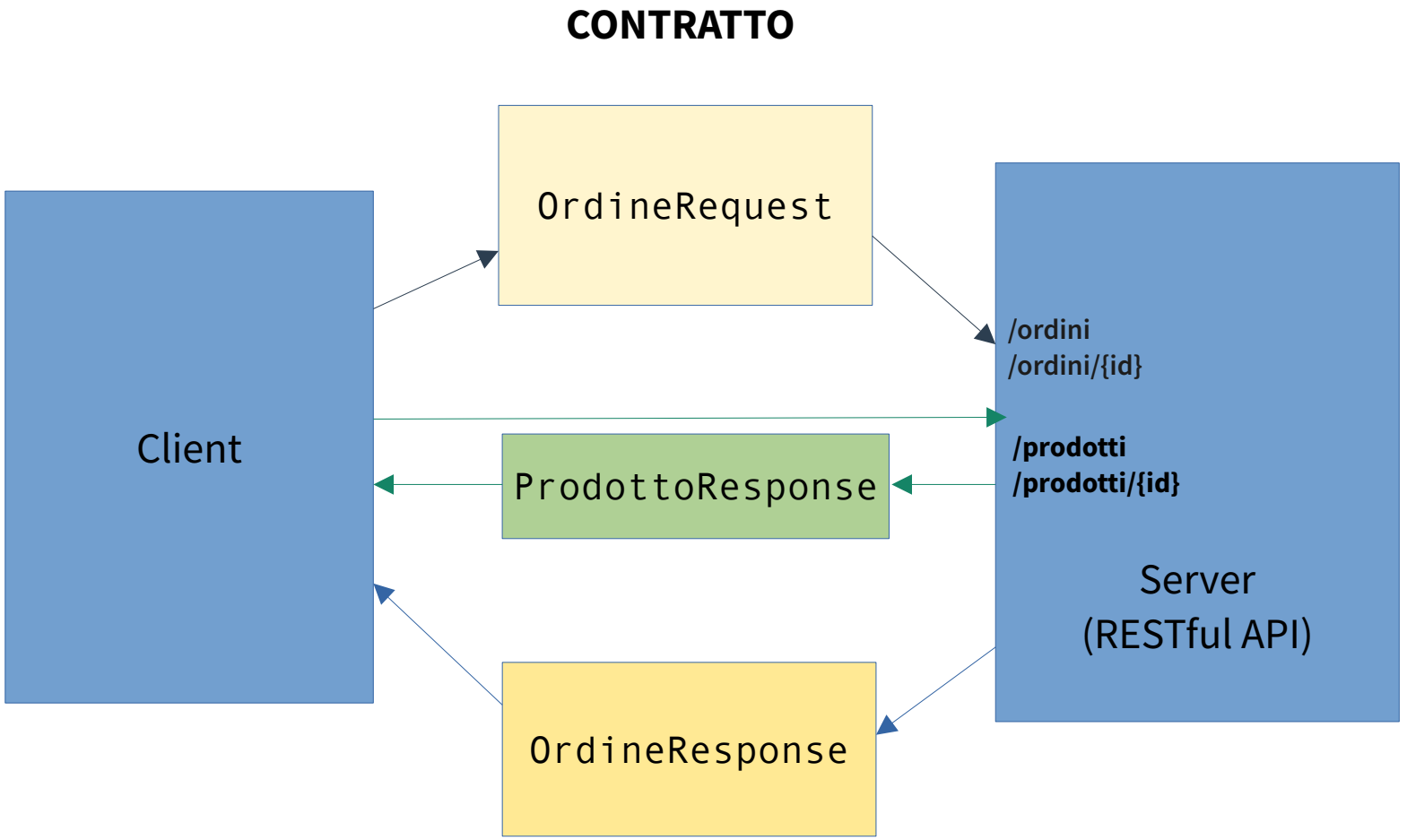
```
Ordine
{
“idOrdine” : “number”,
“data” : “string”,
“valore” : “number”,
“customer” : “?”,
“linee” : [
   “idProdotto” : “number”,
   “quantita” : “number”,
   “prezzo_unitario” : “number”,
   ]
   }
```

```
OrdineRequest
{
“data” : “string”,
“valore” : “number”,
“customer” : “?”,
“linee” : [
   “idProdotto” : “number”,
   “quantita” : “number”,
   “prezzo_unitario” : “number”,
   ]
   }
```

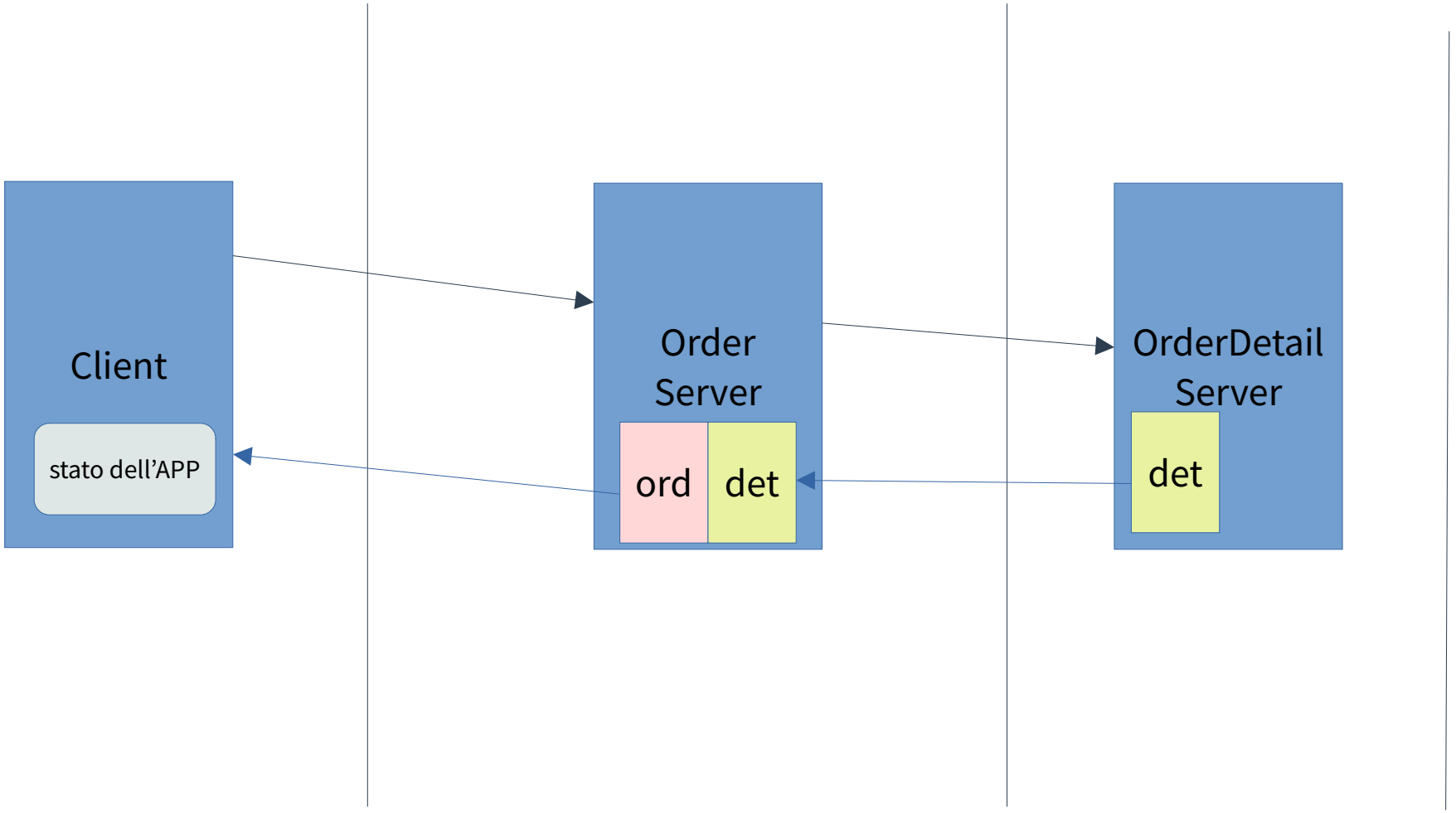
```
OrdineResponse
{
“idOrdine” : “number”,
“data” : “string”,
“valore” : “number”,
“customer” : “?”,
“linee” : [
   “idProdotto” : “number”,
   “quantita” : “number”,
   “prezzo_unitario” : “number”,
   ]
   }
```

**CONTRATTO**

# uso del contratto



# layered system



# POST e idempotenza

POST non è idempotente by default

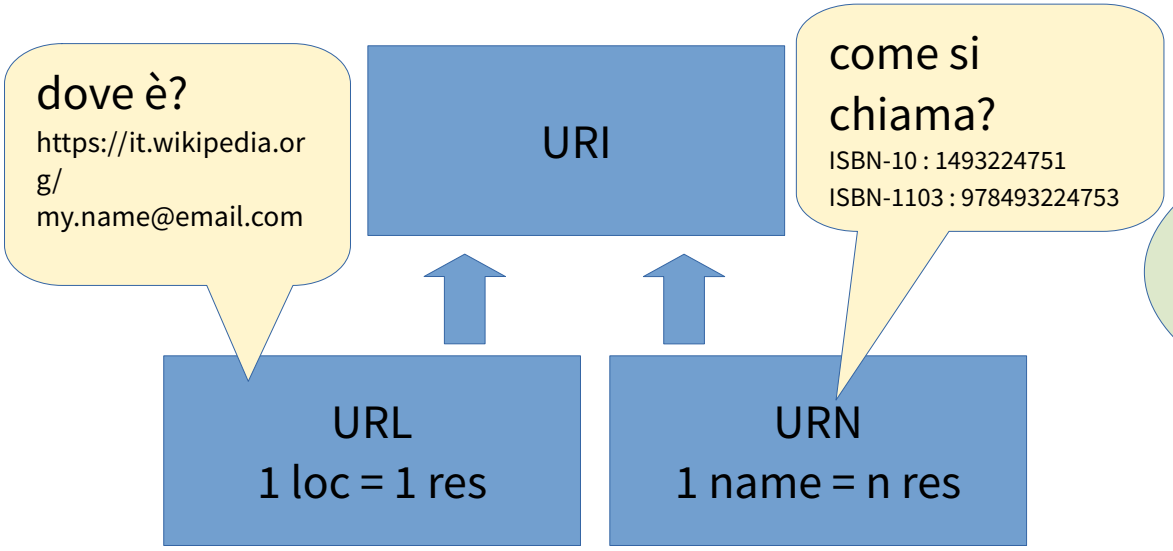
se mi serve una POST idempotente devo poter distinguere la risorsa da creare in qualche modo e controllare che non sia sta già creata. Implicare una GET con criteri

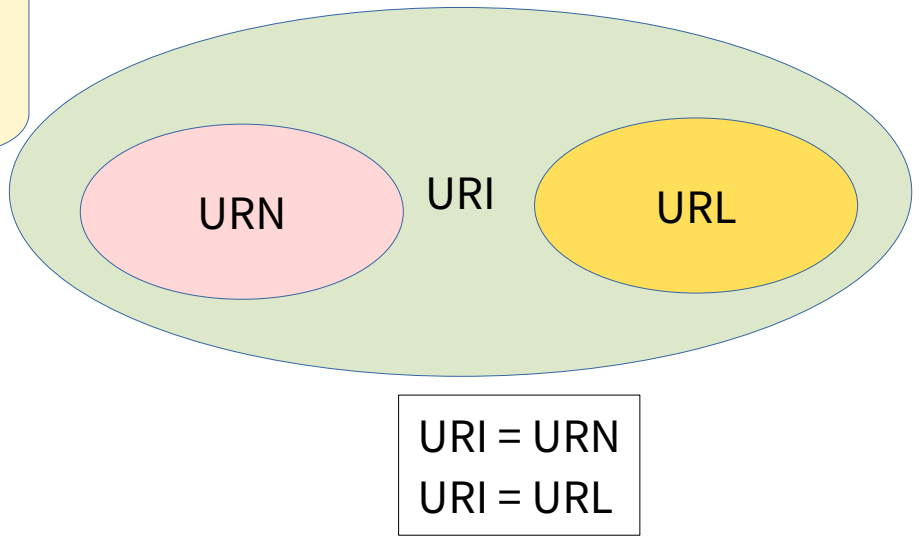
# URI <==> URL e URN

URI = Uniform Resource Identifier

URL = Uniform Resource Locator

URN = Uniform Resouce Name




URI = URN
URI = URL

# Pausa caffè

rientro 11:35

# Pausa Caffè

rientro 16:10