Conditions Databases
on going developments in Atlas and Cms

Exploring a new architecture for conditions data management

A.Formica, E.Gallas, D.Barberis, G.Govi, A.Pfeiffer

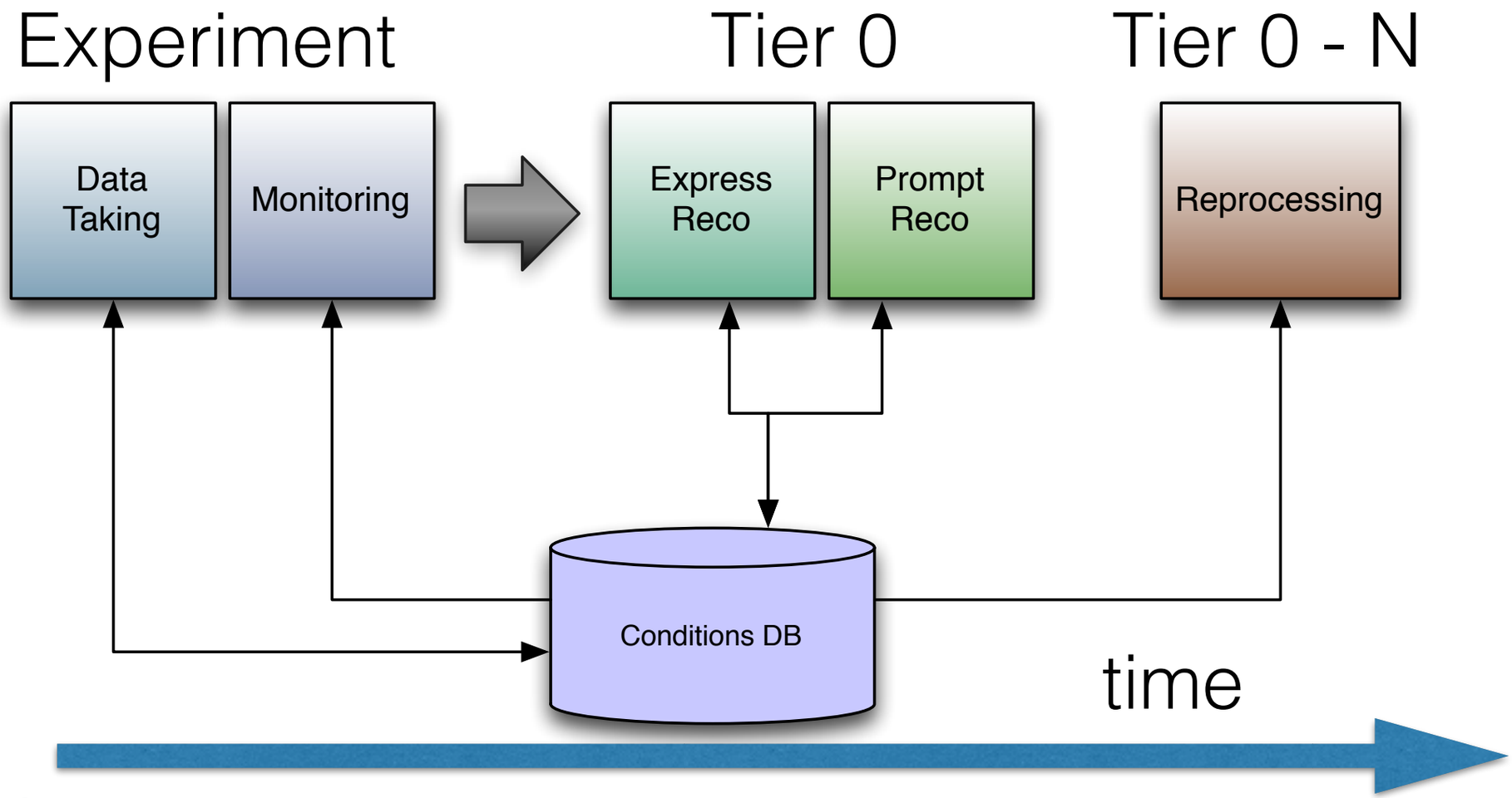
Topics

- What are the conditions data in a physics experiment
  - experience from ATLAS and CMS use cases
- Proposal summary: requirements
  - data model and architecture
- Prototype implementation
  - technology choices for middle tier
  - REST service documentation
  - stress tests and integration
- Summary

Conditions data

- A definition attempt
  - in general they are non-event data varying with time
  - a specific subset is critical for the data flow
    - Status and Configuration for detectors and Trigger
    - Run information
    - Detector control system: HV, LV, ….
    - Detector calibrations (calibrations, alignments, …)
    - Beam and Luminosity information

Dataflows

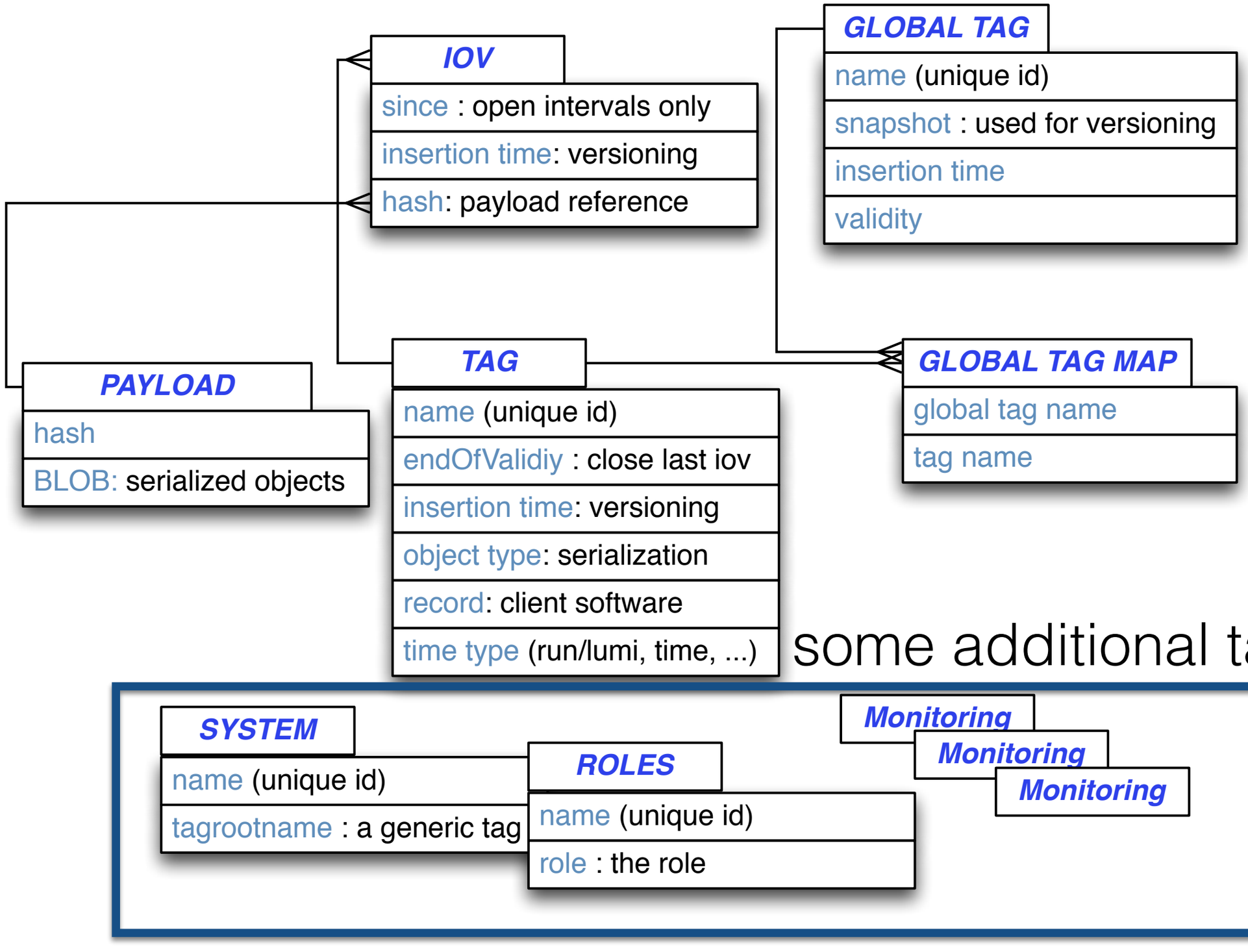


- usage
  - conditions data are used and produced at different steps of our dataflows
  - calibrations are refined in the express and prompt reconstruction
  - in general data reprocessing should benefit of the best knowledge we have on our calibrations

Conditions DB : history

- A common project (ATLAS / CMS)
  - CMS has developed during LS1 a new database schema trying to simplify a lot the structure respect to Run1: this schema is today in production and used for Run2
  - ATLAS has shown interest in this project, since the main concepts that CMS has been using for this new DB data model are very similar to those who have been identified as useful from our Run1 experience
  - Other developments have been triggered by this collaboration, which are more related to the conditions data management and access (multi-tier model)
- Data model
  - short description of the DB structure: it is taken essentially from the CMS conditions database used in Run2 for the production system.
- Towards a REST architecture
  - short description about the architecture for the conditions data management

Data model


IOV
since : open intervals only
insertion time: versioning
hash: payload reference
GLOBAL TAG
name (unique id)
snapshot : used for versioning
insertion time
validity
PAYLOAD
hash
BLOB: serialized objects
TAG
name (unique id)
endOfValidiy : close last iov
insertion time: versioning
object type: serialization
record: client software
time type (run/lumi, time, ...)
GLOBAL TAG MAP
global tag name
tag name
some additional tables (TBD)
SYSTEM
name (unique id)
tagrootname : a generic tag
ROLES
name (unique id)
role : the role
Monitoring
Monitoring
Monitoring

Data model in words

- Conditions data
  - **Payload** : values are consumed as an aggregated set, typically an header and some parameters container (s)
- Conditions metadata
  - **IOV**: *time* information, based on 1 time column (time, run number, …), valid until the next entry in *time*
    an IOV point to 1 payload (via an sha256 hash)
  - **Tag**: label to identify a specific set of IOVs
  - **Global Tag**: consistent set of tags, involved in a given data-flow. A tag can be associated to many global tags.

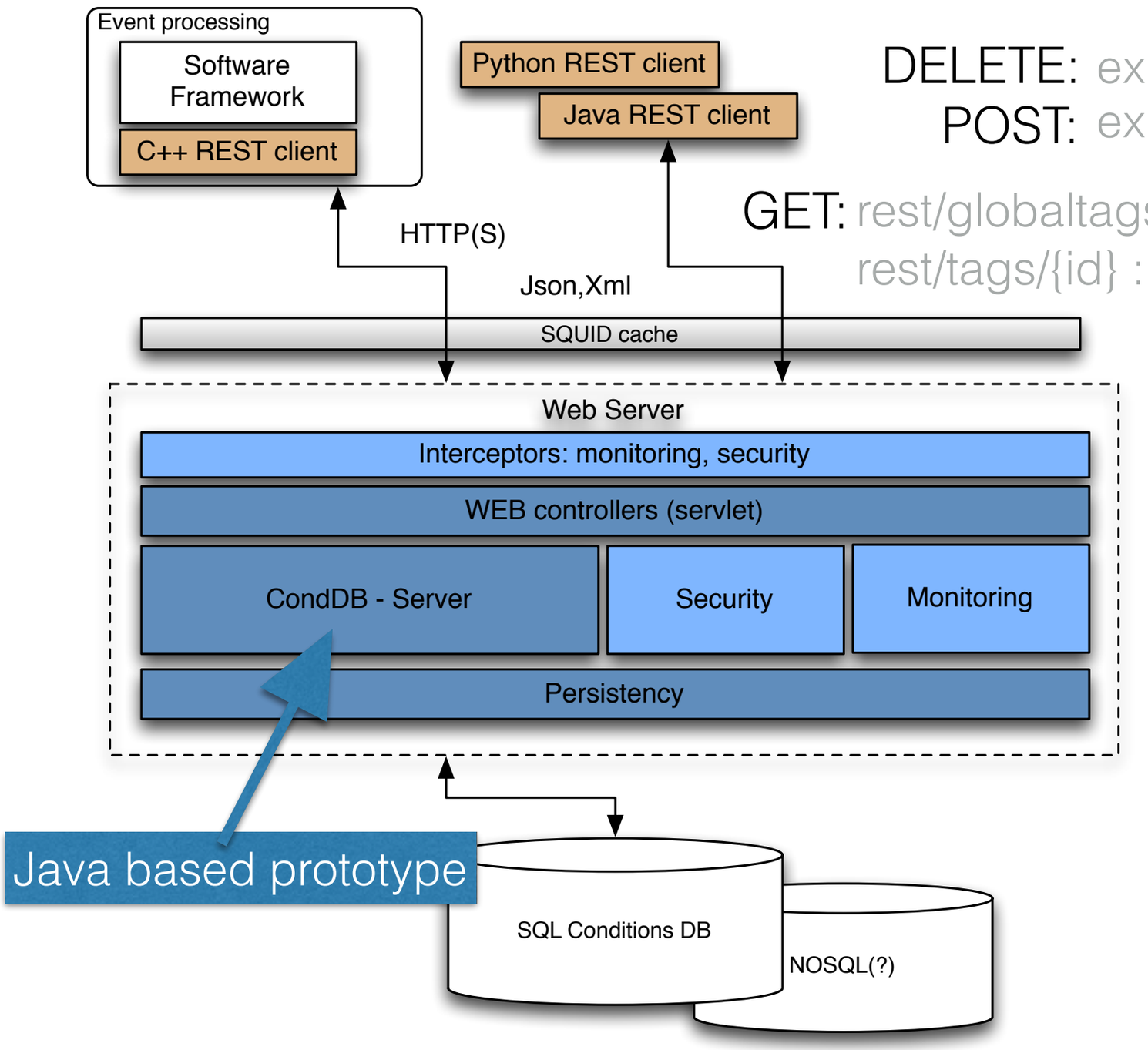
Payload

- A binary large object
  - the payload is stored as a BLOB inside the DB, leaving to the users the choice of the implementation
  - metadata present in the Tag table allow to deserialize the BLOB in the correct way
- no generic implementation
  - CMS has chosen for several reason a Boost library to serialize and deserialize the DB content
  - other choices are possible (ROOT, HDF5, Google Protobuf …), some of them available for multiple languages
  - ASCII serialization is possible (JSON and XML)
- remarks
  - the choice is a compromise between optimisation and flexibility
  - multi-language support could be useful
  - avoid too many serialization formats…

Architecture Requirements

- Multi-tier model architecture based on REST
  - disentangle the client software from the backend system, the data can be available from any framework; just needs to know the REST api, multi-language support can be provided at low development cost.
  - increase monitoring capabilities of conditions usage (server more aware of what are the access patterns, and also aware of insertions, updates etc…)
  - disentangle authentication and authorisation from the DB layer (no more DB passwords in client code)
- Backend
  - The backend can be a relational platform like Oracle, PostgreSQL, MySQL
  - Since the client is not aware of backend details, other solutions can be explored (NoSQL, FileSystem,….)
- Remark:
  - prototype developed using standard Java based solutions

architecture



DELETE: expert/tags/{id} : delete tag <id>
POST: expert/tags {tag-obj} : create a tag

GET: rest/globaltags/{id}?trace=on : trace global tag <id>
rest/tags/{id} : list info on tag <id>

Remarks:

use standard libraries for persistency

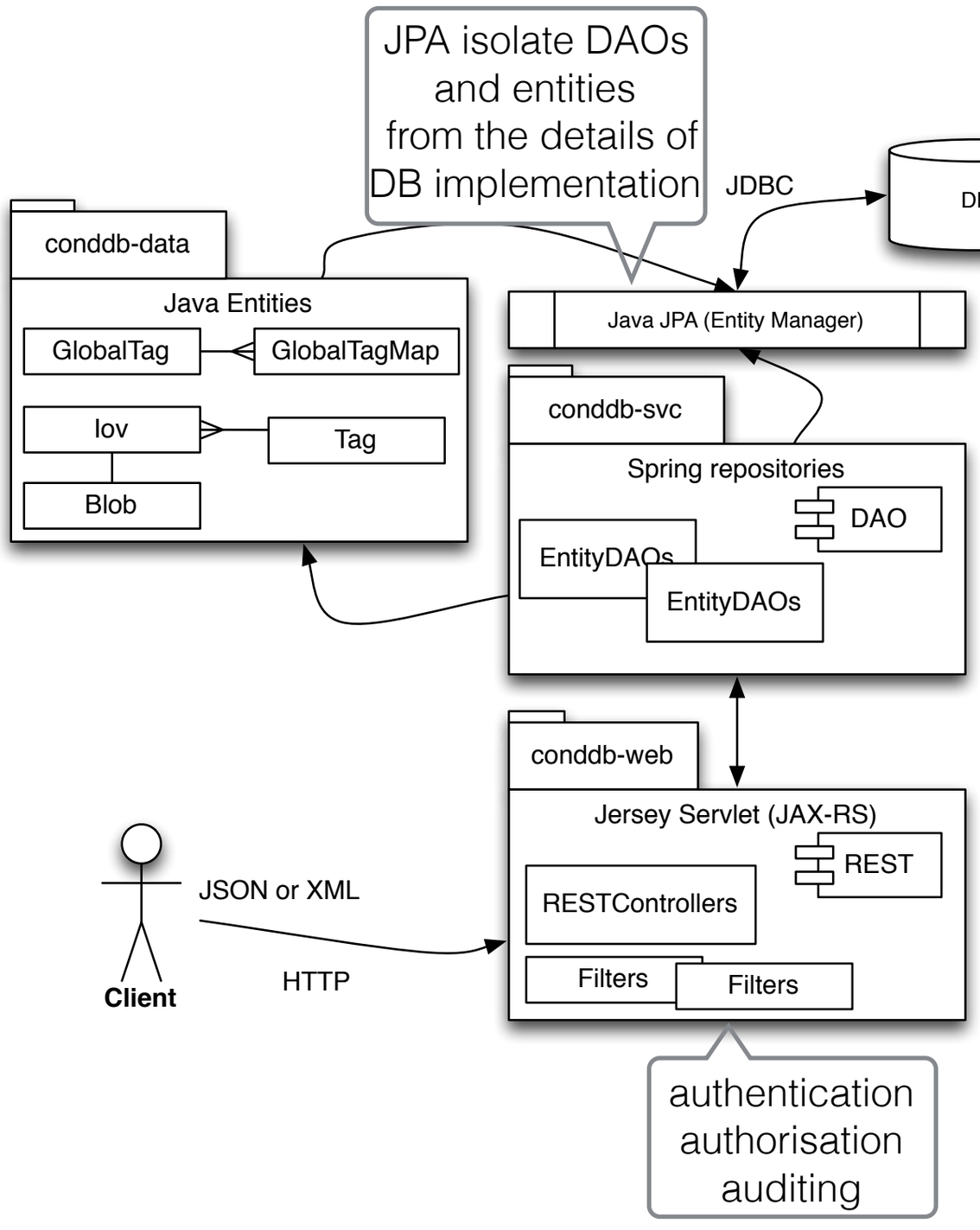
multiple DB support
portable format in requests: Json or Xml data exchanged between client and server

follow REST design in URLs

Implementation

- First implementation use Java
  - Main goal here is to stay as close as possible to Frontier technologies, so that this prototype could be considered a sort of “plugin”, and profit of the huge experience of Frontier group in terms of caching and in general in distributed computing issues
- In general: re-use as much as possible the specifications already available in Java (JEE)
  - JPA (Java Persistency API) for management of relational data
  - JAX-RS (Java API for Restful web Services)
- **Spring**: useful to avoid lot of boiler plate code
  - also practical to deploy the same application in several web servers (hides many “glueing” details)
- Build system : maven
  - Automatic download of *all* project dependencies (defined in pom.xml) from remote maven repositories
- Tests and documentation
  - JUnit (for the moment not a lot of unit tests developed :-((( )
  - Javadoc
- Deployment
  - Web server: tested on Wildfly, Tomcat and Jetty
  - DB: tested on Oracle, h2 database, MySQL, PostgreSQL

CondDB module



- Three main packages : Entities, Services and DAOs, Controllers

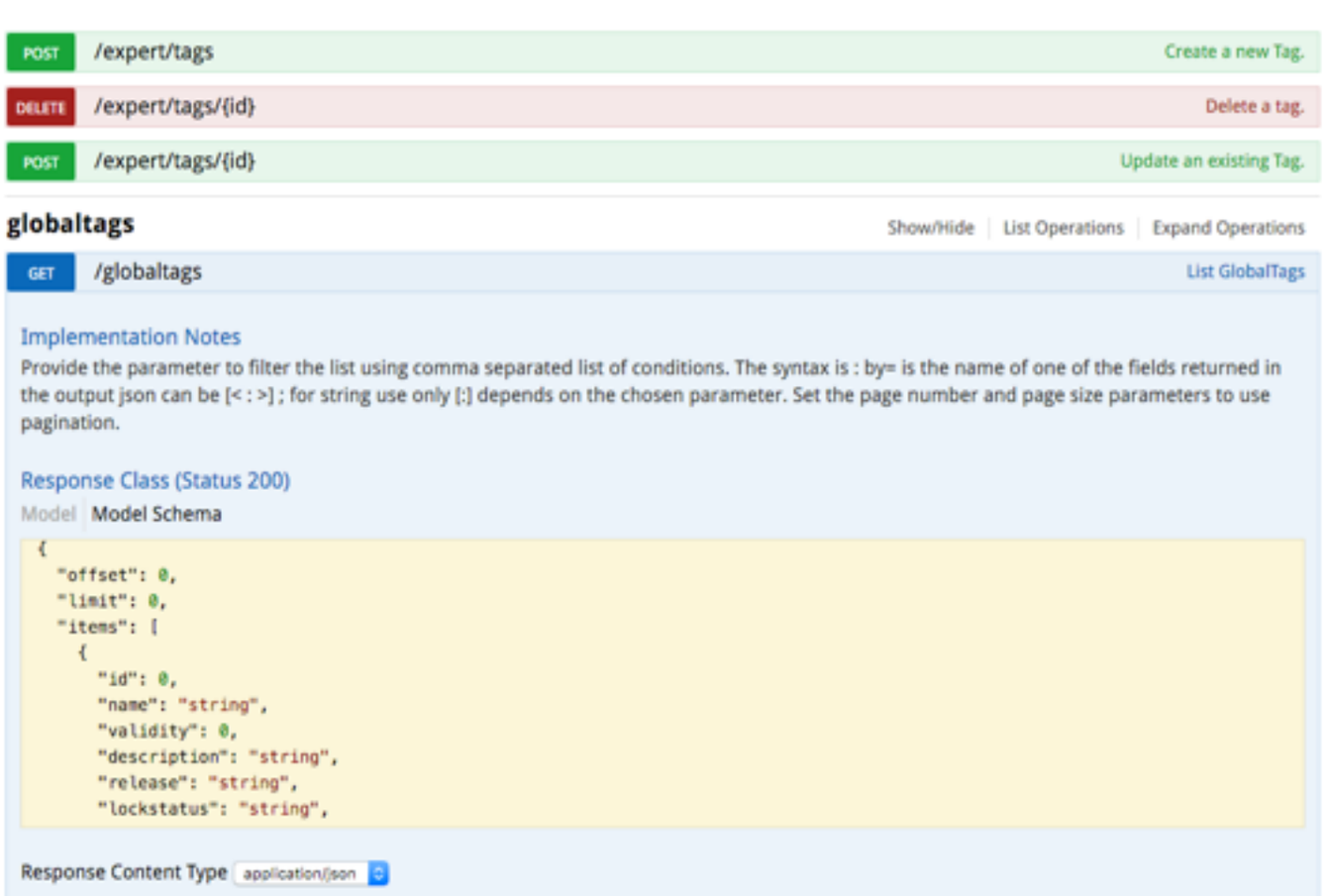
  - conddb-data: entities managed using JPA (hibernate); model created automatically directly from the DB connection where the table are defined, or vice-versa.

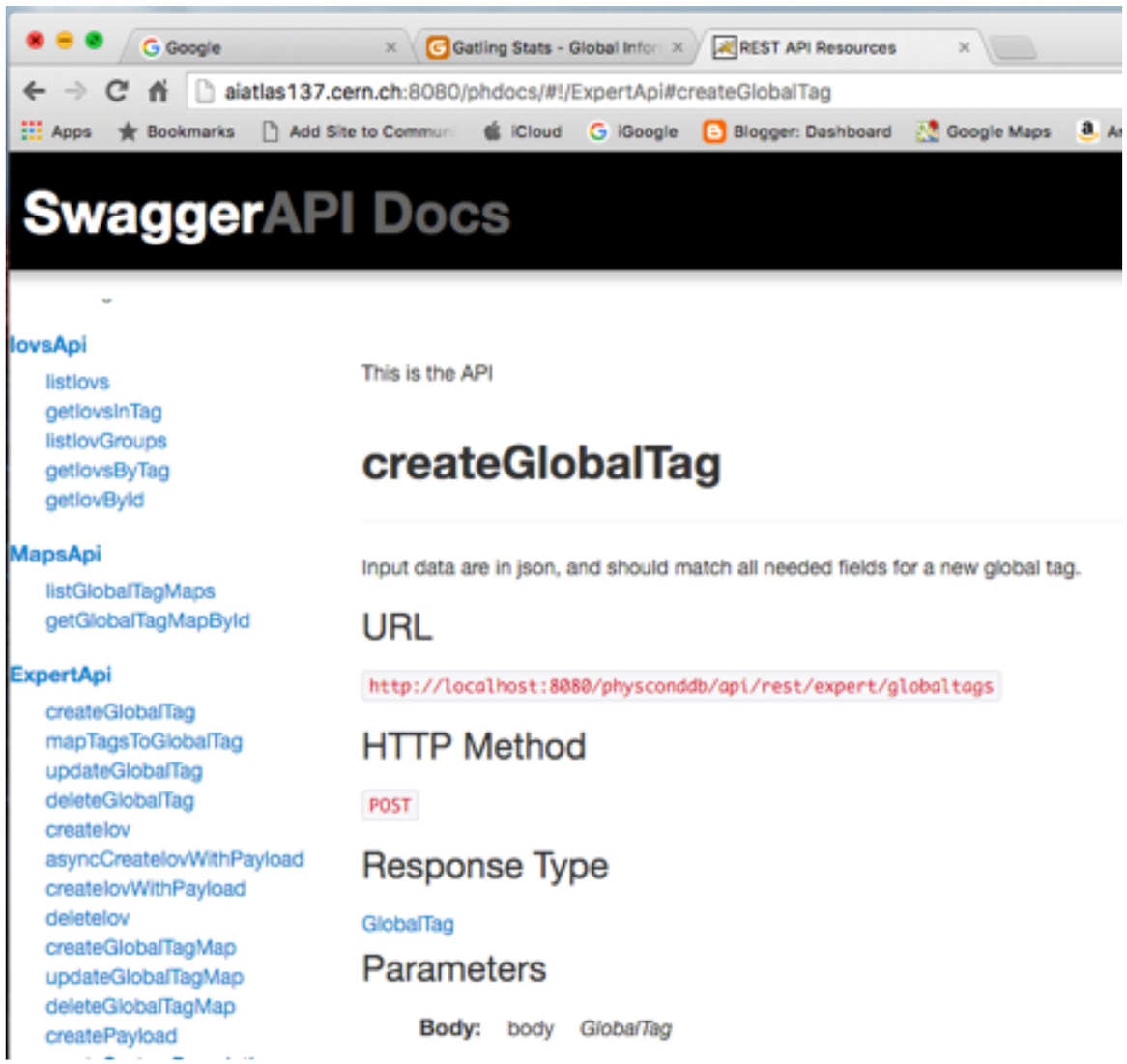
  - conddb-svc: services and dao's are implemented using **spring** repositories. Queries can be declared in annotations or directly generated by spring using the method name.

  - conddb-web: controllers are managed by **Jersey** servlet. Http calls can be intercepted by a chain of filters

Documenting REST

- Use swagger for REST services documentation
  - not intrusive in server code (just few annotations in web controllers methods)
  - generate a JSON documentation file (*swagger.json*) that can be used to:
    - produce an html documentation of the services
    - produce a test web User Interface
    - generate client code (java, python, php, qt5)





Testing REST services

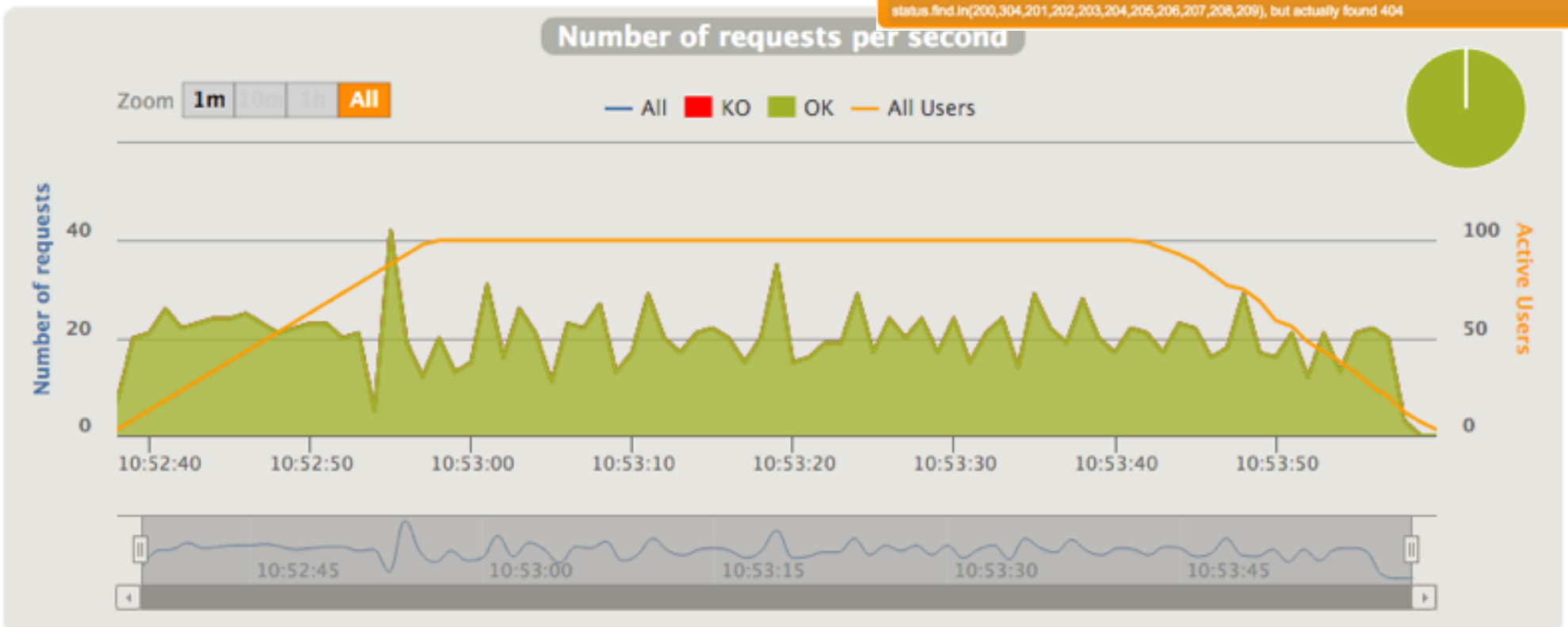
- Use **gatling** for load testing
  - useful for simulating heavy load and spot bottlenecks in queries or web controllers
  - other tools: **jmeter**

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	300	291	9	3%	7.321	96	267	466	3722	3910	3995	901	1275
Page globaltags	100	100	0	0%	2.44	220	323	383	496	573	577	340	89
Page tags	100	100	0	0%	2.44	144	185	206	273	278	281	196	36
Trace	100	91	9	9%	2.44	96	2944	3507	3869	3963	3995	2167	1567

ERRORS

Error	Count	Percentage
status.find.in(200,304,201,202,203,204,205,206,207,208,209), but actually found 404	9	100 %



Summary

- Exploring Conditions Database solutions
  - Joint effort (between ATLAS and CMS) to evaluate a prototype for a new conditions database infrastructure is on going
  - In ATLAS we also started to try using this system as a “master copy” for user analysis calibration files: the files should then be dumped in a cvmfs directory structure for client access, allowing to keep the existing code on client side
  - Many things yet to be done could profit from existing experience in the domain (Frontier caching)
    - Developments still on going both at the level of server features and client libraries
    - Other experiments are also developing conditions data infrastructure (e.g. Belle II is adopting a similar architecture to the one exposed)
    - People also exploring solutions based on CVMFS only
- Open areas
  - Monitoring tools, alternative backends, and server side developments are open