# *Lecture 6: Web Services, Rest & Mastodon Demonstration*

Professor Richard O. Sinnott, Luca Morandini

University of Melbourne

rsinnott@unimelb.edu.au

What's in an Architecture?

A (system) architecture, is just the way different software components are distributed on computers, and the way in which they interact with each other

Architectures are often difficult to describe in words, hence diagrams are often used. Many ways to document them, e.g. UML, …
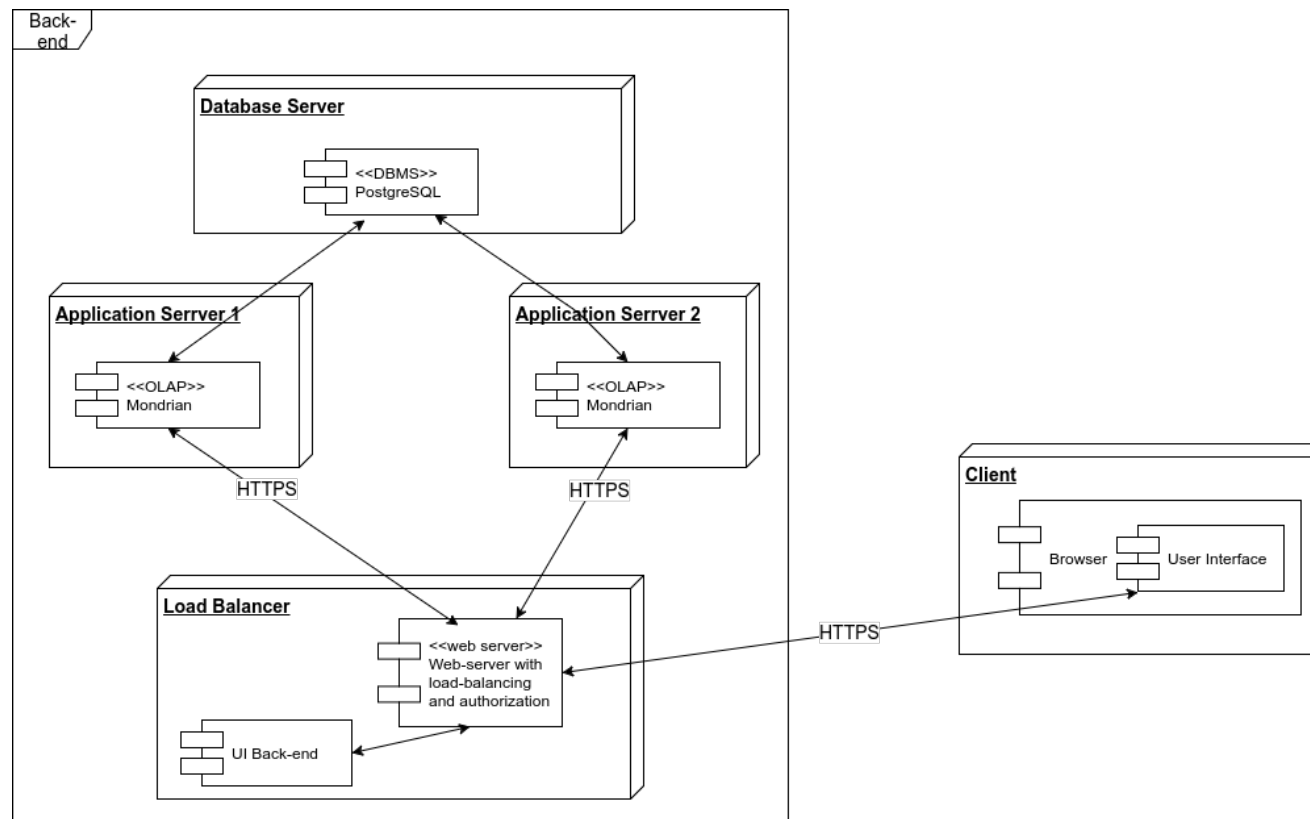
Service-oriented Architectures?

When an architecture is completely contained within the same machine, components can communicate directly, e.g. through *function calls.*

However, when components are distributed such a direct approach typically cannot be used (e.g. Assignment 2!)

Therefore, components and *systems* have to interact in more loosely-coupled ways.

Services are often used for this. Typically *combinations* and *commonality* of services can be used to form a Service-oriented Architecture (SoA).

SoA Core Goals

<mark>Service-oriented Architectures</mark> can serve many goals:

- A set of externally facing services that an organization wants to provide to external collaborators, e.g., to access local data
- An architectural pattern based on service providers, one or more brokers, and service requestors based on *agreed** service descriptions

  *more later

- A set of architectural principles, patterns and criteria that support modularity, encapsulation, loose coupling, separation of concerns, reuse and composability
- A programming model complete with standards, tools and technologies that supports development and support of services*

  *there can be many flavours of services

- A middleware solution supporting service orchestration, monitoring, and management, e.g., as workflows.

SoA Design Principles

- **Standardized service contract**: *Services adhere to a communications agreement, as defined collectively by one or more service-description documents.*

- **Service loose coupling**: *Services maintain a relationship that minimizes dependencies and only requires that they maintain an awareness of each other.*

- **Service abstraction**: *Beyond descriptions in the service contract, services hide logic from the outside world.*

- **Service reusability**: *Logic is divided into services with the intention of promoting reuse.*

- **Service autonomy**: *Services have control over the logic they encapsulate.*

- **Service statelessness**: *Services minimize resource consumption by deferring the management of state information when necessary.*

- **Service discoverability**: *Services are supplemented with communicative metadata by which they can be effectively discovered and interpreted.*

- **Service composability**: *Services should be composable, regardless of the size and complexity of the composition.*

SoA Design Principles…ctd

- **Service granularity**: *services should provide optimal scope at the right granular level of the business functionality.*

- **Service normalization**: *services should be decomposed and/or consolidated to a level that minimizes redundancy, for performance optimization, access, and aggregation.*

- **Service optimization**: *services should be robust and of high quality so that they can serve specific functions.*

- **Service relevance**: *service functionality is presented at a level of granularity recognized* by the user as a meaningful service.*
  - **more later*

- **Service encapsulation**: *services should hide their inner workings.*

- **Service location transparency**: *services should allow consumers to invoke a service regardless* of its actual location in the network.*

   *subject to security demands (more in later lectures)

- **Service security**: *services should provide secure and standardized ways for their invocation to minimize attack vectors.*

   * more in later lectures

SoA and Web Services

- **Web** services used to implement service-oriented architectures

- There are MANY other flavours of web service
    - **Simple Object Access Protocol (SOAP)-based Web Services**
    - **Representational State Transfer (ReST)-based Web Services**
        - Both use HTTP hence can run over the web
        - although SOAP/WS often run over other protocols as well
    - **Many others**
        - Geospatial services (WFS, WMS, WPS, …)
        - Health services (HL7)
        - SDMX (Statistical Data Markup eXchange)
        - …

SOAP vs ReST

- Two patterns to call services over HTTP
  - SOAP/WS is built upon the *Remote Procedure Call paradigm*
    - a language independent function call that spans another system

  - ReST is centered around *resources*, and the way they can be manipulated (added, deleted, etc.) remotely
    - (More later)

  - ReST is more of a style to use HTTP than a separate protocol

  - SOAP/WS is a stack of protocols that covers every aspect of using a remote service, from service discovery, to service description, to the actual request/response

WSDL

The *Web Services Description Language* (WSDL) is an XML-based interface description language that describes the functionality offered by a web service.

WSDL provides a *machine-readable* description of how the service can be called, what parameters it expects, and what results/data structures it returns:

Definition – what it does
Target Namespace – context for naming things
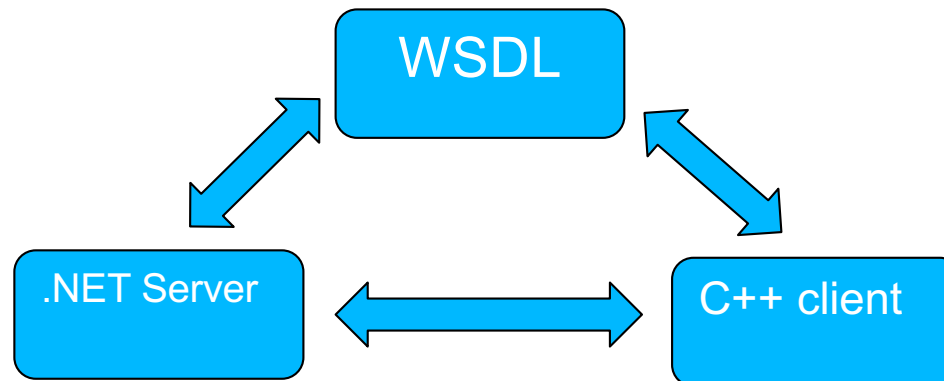Data Types – simple/complex data structures inputs/outputs
Messages – messages and structures exchanged between client and server
Port Type - encapsulate input/output messages into one logical operation
Bindings - bind the operation to the particular port type
Service - name given to the web service itself
Language independence and location transparency…

Simple WSDL Example

```xml
<definitions name = "HelloService"
   targetNamespace = "http://www.examples.com/wsdl/HelloService.wsdl"
   xmlns = "http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap = "http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns = "http://www.examples.com/wsdl/HelloService.wsdl"
   xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
   <message name = "SayHelloRequest">
      <part name = "firstName" type = "xsd:string"/>
   </message>
   <message name = "SayHelloResponse">
      <part name = "greeting" type = "xsd:string"/>
   </message>
   <portType name = "Hello_PortType">
      <operation name = "sayHello">
         <input message = "tns:SayHelloRequest"/>
         <output message = "tns:SayHelloResponse"/>
      </operation>
   </portType>
```

Note

- SOAP for exchange of structured information
- Specific names for operations
- Specific data types
- Specific bindings
- …

```xml
<binding name = "Hello_Binding" type = "tns:Hello_PortType">
   <soap:binding style = "rpc"
   transport = "http://schemas.xmlsoap.org/soap/http"/>
   <operation name = "sayHello">
      <soap:operation soapAction = "sayHello"/>
      <input>
         <soap:body
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="urn:examples:helloservice"
            use = "encoded"/>
      </input>
      <output>
         <soap:body
            encodingStyle=http://schemas.xmlsoap.org/soap/encoding/"
            namespace = "urn:examples:helloservice"
            use = "encoded"/>
      </output>
   </operation>
</binding>
<service name = "Hello_Service">
   <documentation>WSDL File for HelloService</documentation>
   <port binding = "tns:Hello_Binding" name = "Hello_Port">
      <soap:address
         location = "http://www.examples.com/SayHello/" />
   </port>
</service>
</definitions>
```
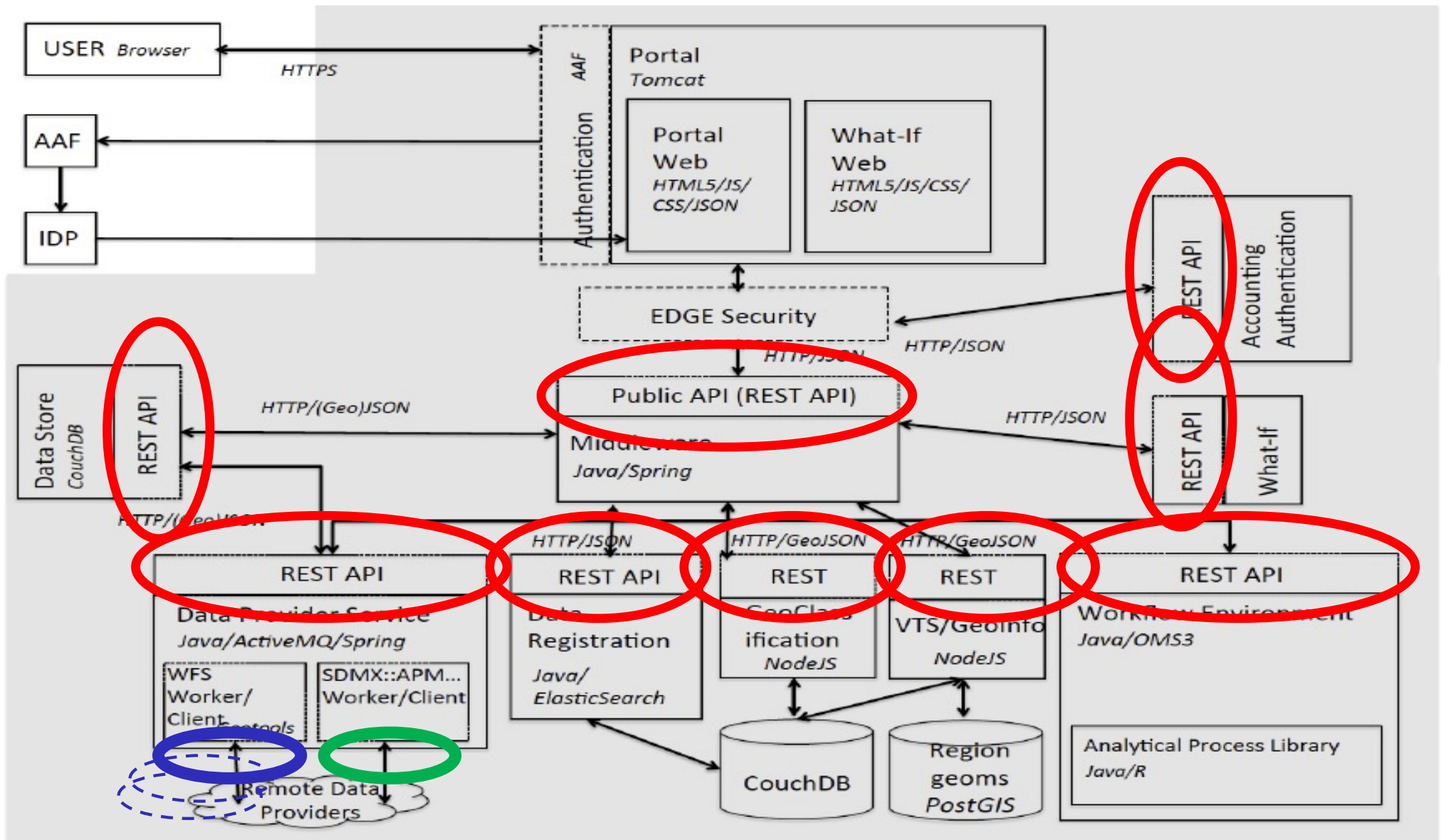
Remembering SUDO Simplified
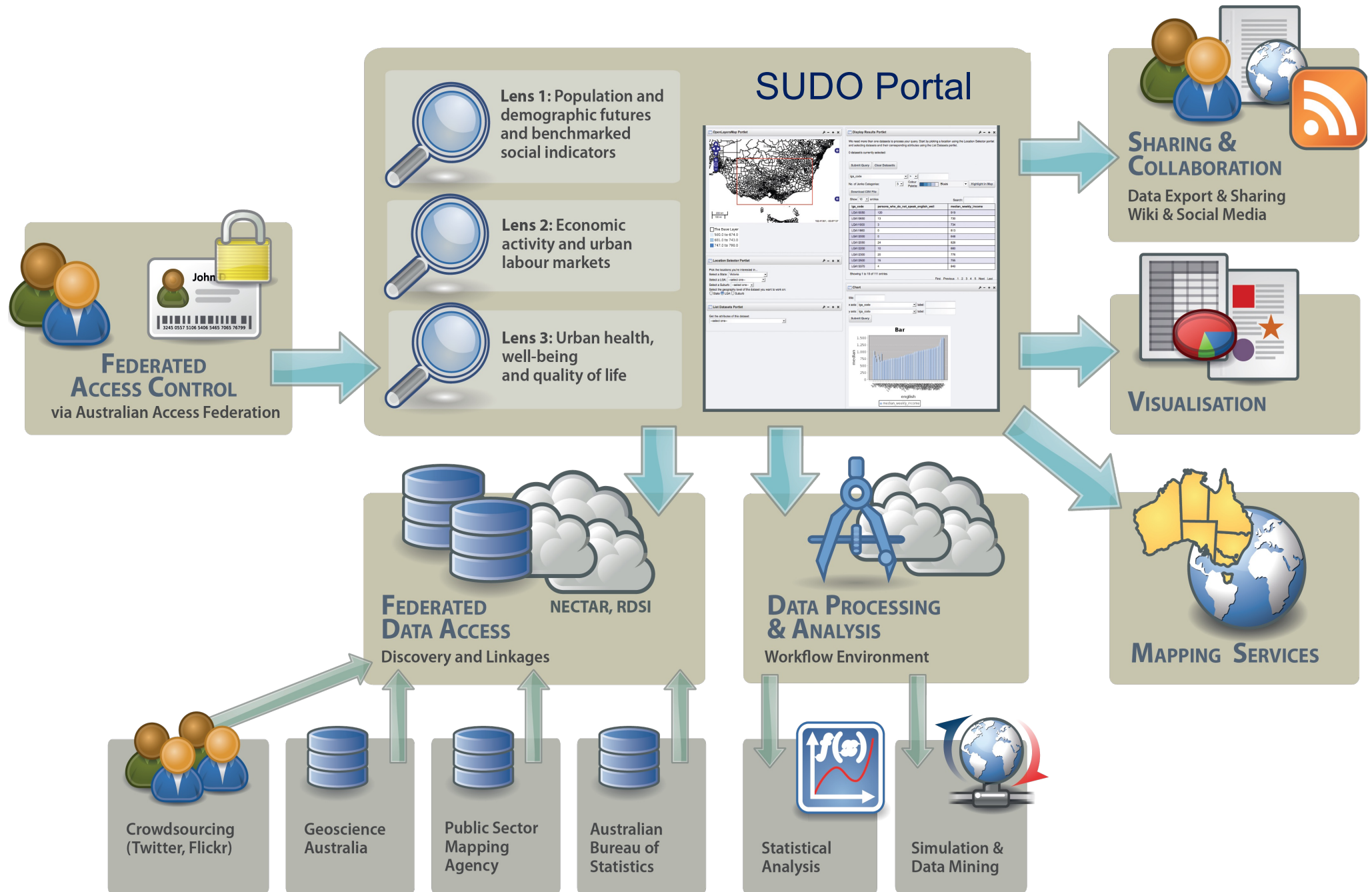
Example of SUDO SoA

What is ReST?

"*Representational State Transfer (ReST)* is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the ==user progresses through an application by selecting links== (state transitions), ==resulting in the next page== (representing the next state of the application) ==being transferred to the user and rendered for their use==."

- Roy Fielding

What's in a Name?

Client → http://amazon.com/product/123 → **Resource**

Product.html

1) Clients request Resource through Identifier (URL)
2) Server/proxy sends representation of Resource
3) This puts the client in a certain state.
4) Representation contains URLs allowing navigation.
5) Client follows URL to fetch another resource.
6) This transitions client into yet another state.
7) Representational **S**tate **T**ransfer!

RESTful applications *navigate* instead of *calling*

- representations contain information about possible traversals

- application navigates to the next resource depending on link semantics

- navigation can be delegated since all links use identifiers

Resource-Oriented Architecture (RoA)

- A RoA is a way of turning a problem into a RESTful web service: an arrangement of URIs, HTTP, and XML that works like the rest of the Web

- A resource is anything that's important enough to be referenced as a thing in itself.

- If your users might

  - want to create a hypertext link to it

  - make or refute assertions about it

  - retrieve or cache a representation of it

  - include all or part of it by reference into another representation

  - annotate it

  - or perform other operations on it

      ...then you should make it a resource.

Resources and URL Examples

Resource	Potential URL
Version 1.0.3 of the software release	http://www.example.com/software/releases/1.0.3.tar.gz
The latest version of the software release	http://www.example.com/software/releases/latest.tar.gz
The first weblog entry for October 24, 2006	http://www.example.com/weblog/2006/10/24/0
A road map of Little Rock, Arkansas	http://www.example.com/map/roads/USA/AR/Little_Rock
Some information about jellyfish	http://www.example.com/wiki/Jellyfish
A list of the open bugs in the bug database	http://www.example.com/bugs/by-state/open
The relationship between two acquaintances, Alice and Bob	http://www.example.com/relationships/Alice;Bob

Mapping Actions to HTTP Methods

ACTION	HTTP METHOD
Create Resource	PUT POST
Retrieve Resource	GET
Update Resource	PUT
Delete Resource	DELETE

- PUT should be used when target resource URL is known by the client
- POST should be used when target resource URL is server generated
- Also:
  - Http HEAD, OPTIONS, CONNECTION, TRACE, PATCH

ReST Best Practices #1

1) Keep URIs short – and create URIs that don't change.
2) URIs should be opaque identifiers that are meant to be discovered by following hyperlinks, not constructed by the client.
3) Use nouns, not verbs in URLs
4) Make all HTTP GETs side-effect free. Doing so makes the request "safe".
5) Use links in your responses to requests! Doing so connects your response with other data. It enables client applications to be "self-propelled", i.e. "what is the next step to take".

ReST Best Practices #2

6) Minimize use of query strings. For example:
   Prefer:
   `http://www.amazon.com/products/AXFC`
   Over:
   `http://www.amazon.com/products?product-id=AXFC`

7) Use HTTP status codes to convey errors/success

200 OK	400 Bad Request	500 Internal Server Error
201 Created	401 Unauthorized	501 Not Implemented
202 Accepted	402 Payment Required	502 Bad Gateway
203 Non-Authoritative	403 Forbidden	503 Service Unavailable
204 No Content	404 Not Found	504 Gateway Timeout
205 Reset Content	405 Method Not Allowed	505 Version Not Supported
206 Partial Content	406 Not Acceptable	
300 Multiple Choices	407 Proxy Auth Required	
301 Moved Permanently	408 Request Timeout	
	409 Conflict	

ReSTful Services should have <mark>Uniform Interface</mark>

Uniform Interface :

- ## Identification of Resources

  All important resources are identified by one (uniform) resource identifier mechanism (e.g. HTTP URL)

- ## Manipulation of Resources through representations

  Each resource can have one or more representations. Such as application/xml, application/json, text/html, etc. Clients and servers negotiate to select representation.

- ## Self-descriptive messages

  Requests and responses contain not only data but additional headers describing how the content should be handled. Such as if it should be cached, authentication requirements, etc. Access methods (actions) mean the same for all resources (universal semantics)

HTTP Methods

- HTTP methods can be
  - Safe
  - Idempotent
  - Neither
- Safe methods
  - Do not change repeating a call is equivalent to not making a call at all.
- <mark>Idempotent methods</mark>
  - Effect of repeating a call is equivalent to making a single call
- GET, OPTIONS, HEAD – Safe
- PUT, DELETE – Idempotent
- POST – Neither safe nor idempotent

Demonstration of Mastodon Client

References

- *Restful Web Services*, Leonard Richardson and Sam Ruby, 2007, O'Reilly.

- *Architectural Styles and the Design of Network-based Software Architectures*. Fielding, Roy Thomas, UC - Irvine, 2000.