



# **Internet of Things and Services**

## Service-oriented architectures

# **RESTful Web services & REST API**

Department of Computer Science  
Faculty of Electronic Engineering, University of Nis

**Internet of Things and Services**  
Computing and informatics

Prof. dr Dragan Stojanović



# References

## Books

- Michael Papazoglou, [Web Services and SOA: Principles and Technology, 2 edition](#), Pearson 2013.
- Gustavo Alonso, Fabio Casati, Harumi A. Kuno, Vijay Machiraju, [Web Services - Concepts, Architectures and Applications](#), Data-Centric Systems and Applications, Springer 2004.

## Courses

- [Introduction to Service Design and Engineering](#)
  - *University of Trento, Italy*
- [Service Technologies](#)
  - *Politechnico di Milano, Italy*



# REST API

# OpenAPI specification

- ✿ The OpenAPI Specification is a community-driven open specification within the OpenAPI Initiative, a Linux Foundation Collaborative Project.
- ✿ The OpenAPI Specification (OAS) defines a standard, programming language-agnostic interface description for HTTP APIs, which allows both humans and computers to discover and understand the capabilities of a service without requiring access to source code, additional documentation, or inspection of network traffic.
- ✿ Version 3.1.0 - Published 15 February 2021
  - ✿ <http://spec.openapis.org/oas/v3.1.0>
  - ✿ <https://github.com/OAI/OpenAPI-Specification>



# REST API tools (open source)

## ✚ Swagger

- ✚ <https://swagger.io/tools/open-source/>
- ✚ Swagger Editor, Swagger Codegen, Swagger UI

## ✚ Nswag

- ✚ <https://github.com/RicoSuter/NSwag>
- ✚ NSwagStudio  
<https://github.com/RicoSuter/NSwag/wiki/NSwagStudio>

## ✚ Swashbuckle

- ✚ <https://github.com/domaindrivendev/Swashbuckle.AspNetCore>

## ✚ AutoRest

- ✚ <https://github.com/Azure/autorest>

## ✚ OpenAPI Generator

- ✚ <https://github.com/OpenAPITools/openapi-generator>

**RESTful Web services & REST API**



# gRPC



# gRPC

- gRPC is a language agnostic, high-performance RPC framework - <https://grpc.io/>
- Contract-first API development, using Protocol Buffers by default, allowing for language agnostic implementations.
- Tooling available for many languages to generate strongly-typed servers and clients.
- Supports client, server, and bi-directional streaming calls.
- Reduced network usage with Protobuf binary serialization.
- Ideal for:
  - Lightweight microservices where efficiency is critical.
  - Polyglot systems where multiple languages are required.
  - Point-to-point real-time services that need to handle streaming requests or responses.



# gRPC ↔ HTTP APIs with JSON

Feature	gRPC	HTTP APIs with JSON
Contract	Required (.proto)	Optional (OpenAPI)
Protocol	HTTP/2	HTTP
Payload	Protobuf (small, binary)	JSON (large, human readable)
Prescriptiveness	Strict specification	Loose. Any HTTP is valid.
Streaming	Client, server, bi-directional	Client, server
Browser support	No (requires grpc-web)	Yes
Security	Transport (TLS)	Transport (TLS)
Client code-generation	Yes	OpenAPI + third-party tooling





# gRPC Code generation

- ✿ All gRPC frameworks provide first-class support for code generation.
- ✿ A core file to gRPC development is the .proto file, which defines the contract of gRPC services and messages.
- ✿ From this file, gRPC frameworks generate a service base class, messages, and a complete client.
- ✿ By sharing the .proto file between the server and client, messages and client code can be generated from end to end.



# gRPC recommended scenarios

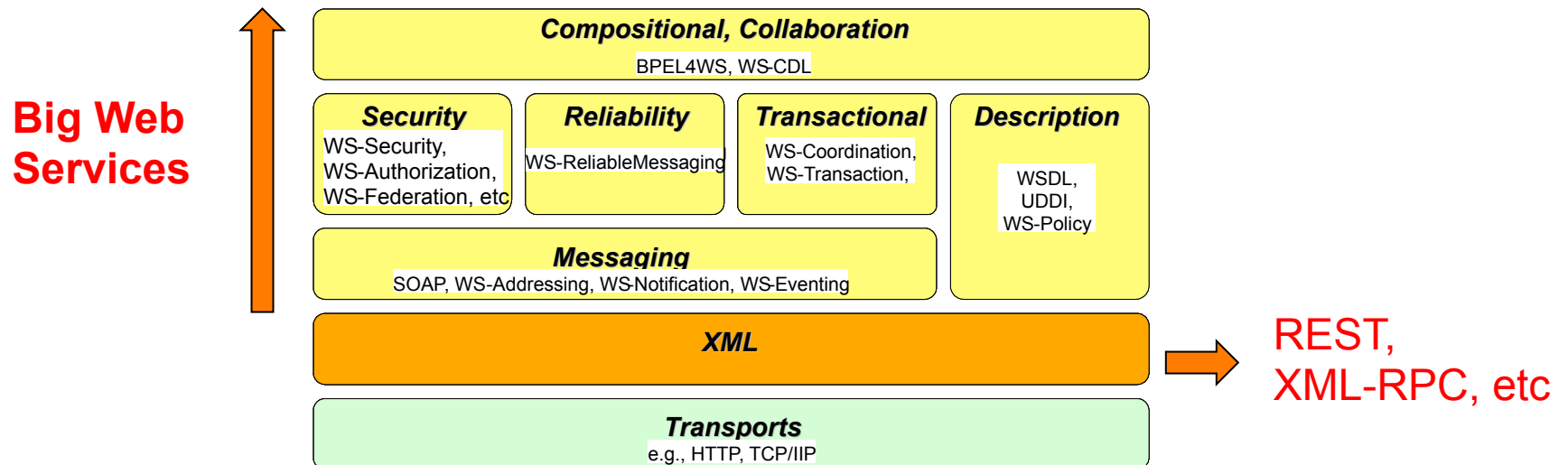
- ✦ **Microservices**: gRPC is designed for low latency and high throughput communication in lightweight microservices.
- ✦ **Point-to-point real-time communication**: gRPC has excellent support for bi-directional streaming, and gRPC services can push messages in real-time without polling.
- ✦ **Polyglot environments**: gRPC tooling supports all popular development languages, making it a good choice for multi-language environments.
- ✦ **Network constrained environments**: gRPC messages are serialized with Protobuf, a lightweight message format, so a gRPC message is always smaller than an equivalent JSON message.
- ✦ **Inter-process communication (IPC)**: IPC transports such as Unix domain sockets and named pipes can be used with gRPC to communicate between apps on the same machine.
- ✦ **RESTful JSON Web APIs** can be automatically created from gRPC services by annotating the *.proto* file with HTTP metadata.

# BACKGROUND: XML-RPC

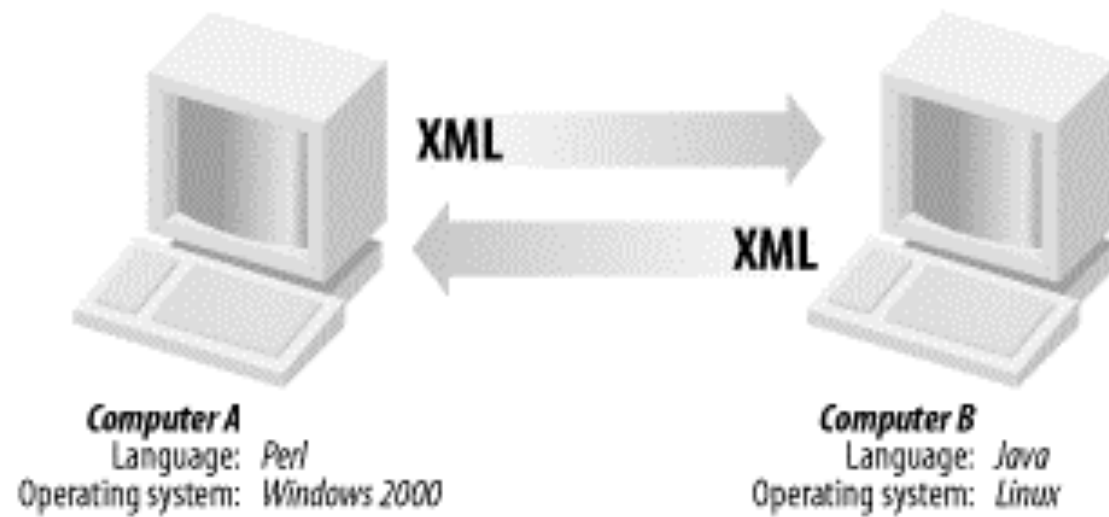
# Web Service Arena

- Basically, two directions
  - XML over HTTP, or extensions thereof
  - WS-\* standards

Maurizio Marchese – IntroSDE

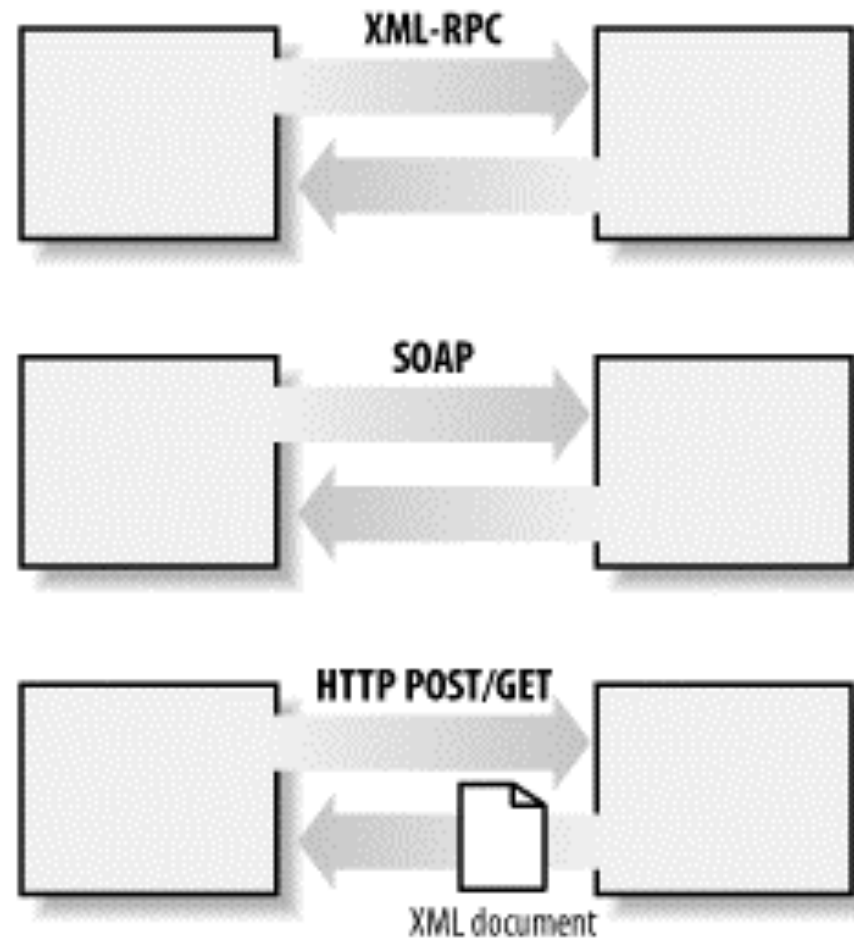


# XML Messaging



# XML- messaging

Maurizio Marchese – IntroSDE





# Web Services Basic Protocols

---

- **XML-RPC**: is a remote procedure call protocol which uses XML for marking up both requests and responses.
- **SOAP** (Simple Object Access Protocol): is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks.
- **REST** (Representational State Transfer): uses plain old HTTP to make method calls and you get XML, JSON, or other data back.



# XML-RPC Web Services

---

- | ■ Remote Procedure Call
  - Encodes commands for the web service
- Request encoded in plain XML
- Sent as payload of http POST requests
- Web service
  - Reads XML
  - Executes commands
  - Returns result in XML





# Example: XML-RPC call

---

- An HTTP envelope that contains an XML document describing an XML-RPC request

```
POST /rpc HTTP/1.1
Host: www.unitn.it
Agent: XMLRPC::Client
Content-Type: text/xml; charset=utf-8
Content-Length: 158
Connection: keep-alive

<?xml version="1.0" encoding = "ISO-8859-1" ?>
<methodCall>
  <methodName> weather.getweather </methodName>
  <params>
    <param>
      <value>Aberdeen AB25</value>
    </param>
  </params>
</methodCall>
```



# Example: XML-RPC response

```
<?xml version="1.0" encoding = "ISO-8859-1" ?>
<!-- The web service generates a response -->
<!-- This is appended to the http response in the normal way
      after the header information-->

<methodResponse>
  <params>
    <param>
      <value>Horrible and wet</value>
    </param>
  </params>
</methodResponse>
```



# XML-RPC pro & contra

---

## ■ PRO

- Uses current technologies (XML, HTTP,...) and remote procedure call architecture (RPC)
- A complex middleware infrastructure is NOT needed

## ■ CONTRA

- An XML-RPC service ignores most features of HTTP
  - It exposes only one URI (the “endpoint”), and supports only one action on that URI (POST)
- The server needs to implement the RPC method(s) with **no standardization**
- Explicitly designed to support only RPC architecture

# RESTFUL SERVICES: INTRO



# What is REST ?

---

- An *architectural style* of networked systems (not a protocol - not a specification)
- Objective: expose **resources** on a networked system (the Web)
- REST itself is not an official standard or a recommendation. It is just a “**design guideline.**”



# Origin of REST

---

- **REST** is an acronym standing for Representational State Transfer.
- First introduced by **Roy T. Fielding** in his PhD dissertation "*Architectural Styles and the Design of Network-based Software Architectures*", UC Irvine, 2000
- He focused on the rationale behind the design of the modern Web architecture and how it differs from other architectural styles.
  - REST is a **client-server** architecture
  - Only **representations** of resources are exposed to the client
  - The representation of resources places the client application in a **state**.
  - Client state may **evolve** with by traversing hyperlinks and obtaining new representations



# What is a Resource?

---

A **resource** is something that:

- is **unique** (i.e., can be identified uniquely)
- has at least one **representation**,
- has one or more **attributes** beyond ID
- has a potential **schema**, or definition
- can provide **context**
- is reachable within the **addressable** universe



# What is a Resource?

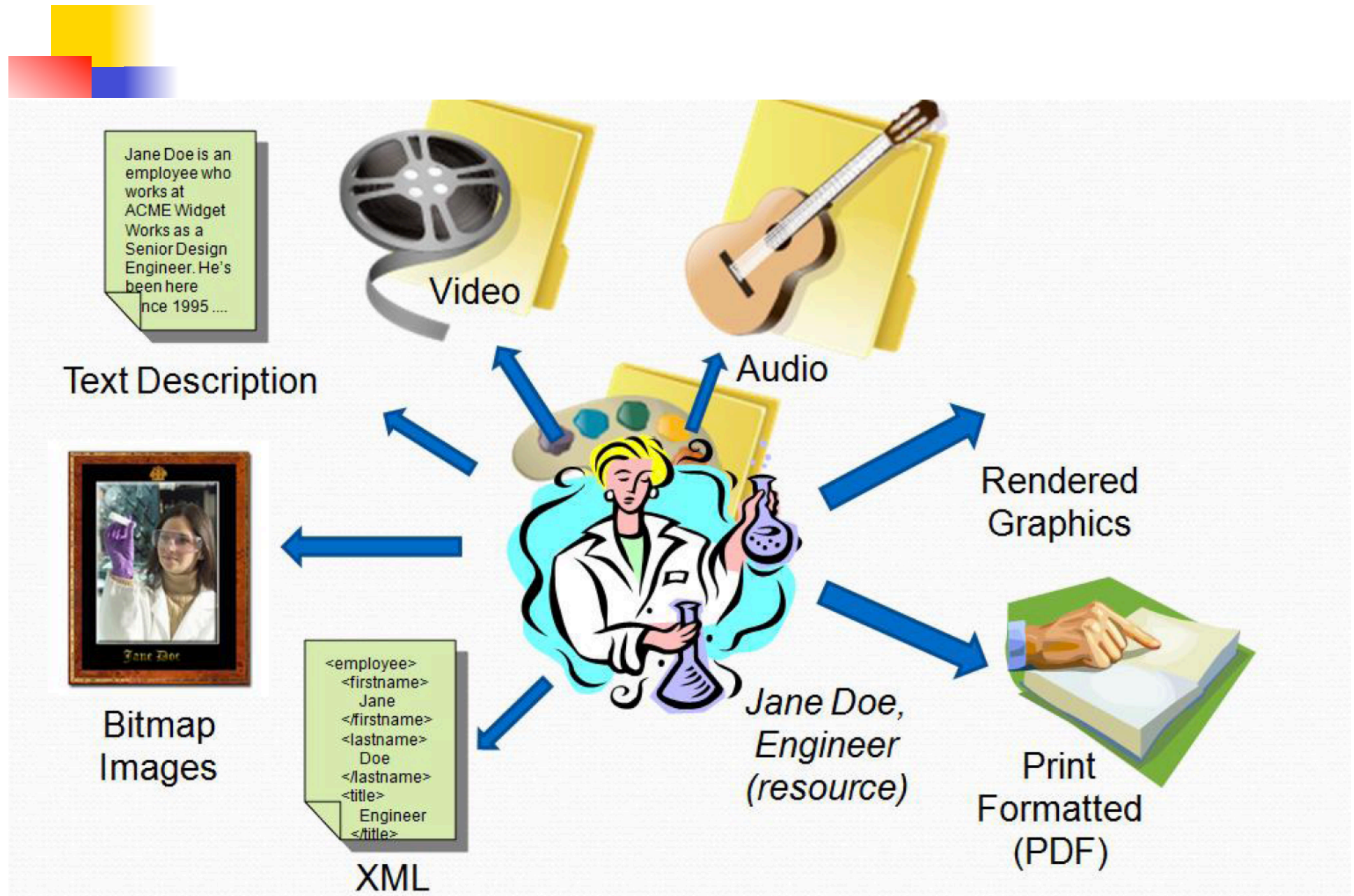
---

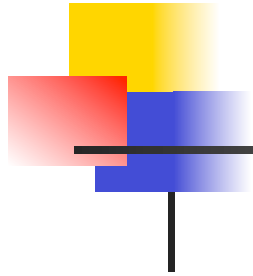
Examples:

- Web Site
- Resume
- Aircraft
- A song
- A transaction
- An employee
- An application
- A Blog posting
- A printer
- Winning lottery numbers
- ..



# Resource Representation





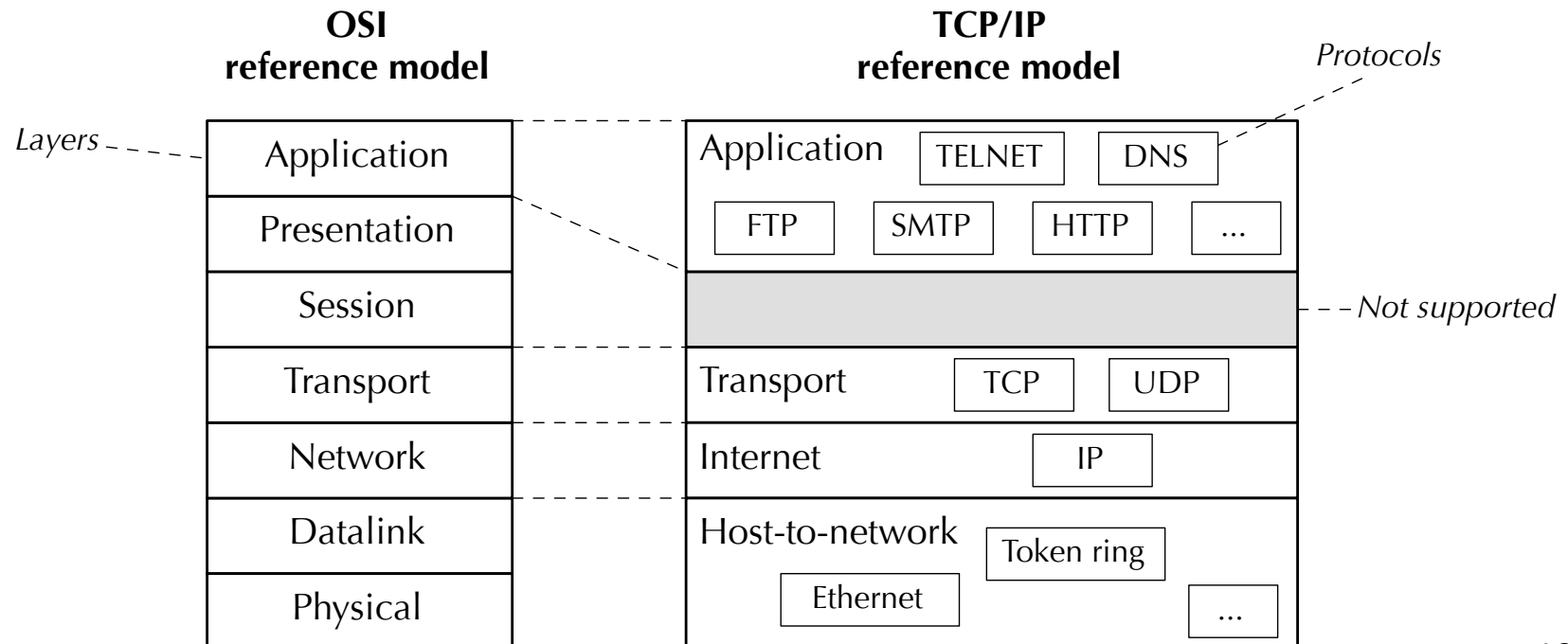
# HTTP

# HTTP

**HTTP** = Hypertext Transfer Protocol

Application protocol for distributed, collaborative hypermedia systems

Maurizio Marchese – IntroSDE





# HTTP Request Methods

---

Methods define **actions** on resources.

Most important methods:

- **GET**: requests a representation of the specified resource
- **HEAD**: retrieves meta-information about a resource
- **POST**: requests that the server accept the entity enclosed in the request as a new subordinate of the web resource identified by the URI
- **PUT**: requests that the enclosed entity be stored under the supplied URI.
- **DELETE**: deletes the specified resource

# REST PRINCIPLES



# Rest principles

---

- Resource Identification
- Addressability
- Statelessness
- Resource Representations
- Links and Connectedness
- Uniform Interface

# **RESOURCE IDENTIFICATION AND ADDRESSABILITY**



# REST principles: Resource Identification

---

- Resources are identified by a URI (Uniform Resource Identifier)
  - <http://www.example.com/software/release/1.0.3.tar.gz>
- **A resource has to have at least one URI**
- Most known URI type are URN and URL
  - URN (Uniform Resource Name)
    - <urn:isbn:0-486-27557-4>
  - URL (Uniform Resource Locator)
    - <file:///home/username/RomeoAndJuliet.pdf>
- Every URI designates exactly one resource
  - <http://www.example.com/software/releases/1.0.3.tar.gz>
  - <http://www.example.com/software/releases/latest>





# REST principles: Addressability

---

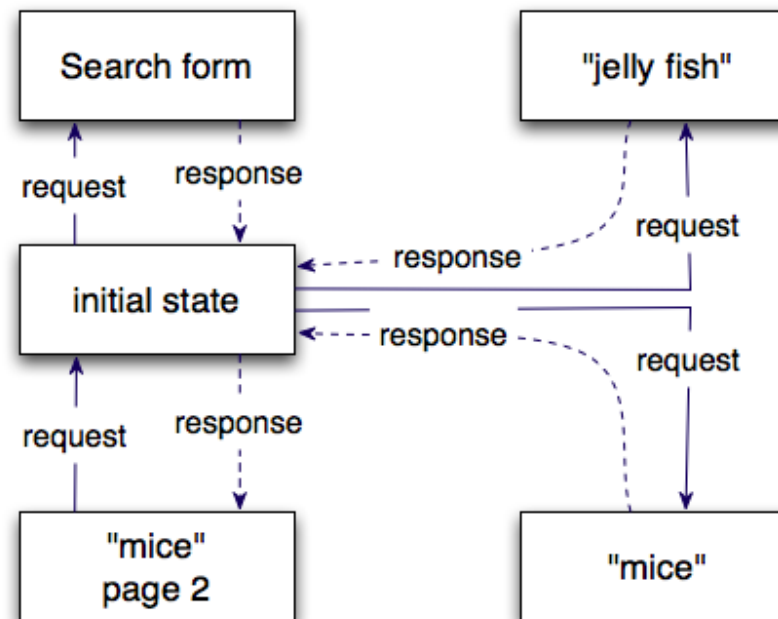
*An application is 'addressable' if it exposes its data set as resources (i.e., a conceptually **infinite** number of URIs)*

- Filesystem on your computer is an addressable system
- The cells in a spreadsheet are addressable (cell referencing)
- Google is addressable: a resource  
'<http://www.google.com/search?q=unitn>'
  - you can bookmark, use it as link in a program, you can email it , etc.
- REST advocates addressability as a main feature of its principles
- This seems natural, the way the Web should work, but many applications don't work this way all the time

# STATELESSNESS

# REST principles: Statelessness

*Stateless means every HTTP request happens in a complete isolation. When a client makes an HTTP request, it includes the necessary information for the server to fulfill the request*



*A Stateless Search Engine: Page 88 RESTful Web Services*



# REST principles: Statelessness

---

*Stateless is good !! - **scalable**, easy to cache, addressable URI can be bookmarked (e.g., page 50 of a search result)*

- HTTP is by nature stateless
- We do something to break it (cookies, sessions...)
- The most common way to break it are 'HTTP sessions'
- The first time a user visits your site, he gets a unique string that identifies his session with the site
  - <http://www.example.com/forum?PHPSESSIONID=27314962133>
  - The string is a key into a data structure on the server.
  - This data structure contains client state.
- *"HTTP sessions break a web service client's back button"*



# REST principles: Statelessness

---

*REST principle says: **URI** needs to contain the client state within it, not just a key to state stored on the server*

- A RESTful service requires that the state stays on the client side
- The client transmits the state to the server for every request
- The server can nudge the client toward new states by adding 'next links' for the client to representations
- The server does not keep state of its own on behalf of a client



# REST principles: Statelessness

---

## ***Application State vs. Resource State***

What exactly counts as 'state' then? Think of flickr.com: would statelessness mean that I have to send every one of my pictures along with every request to flickr.com ???? Of course not.

- Two kinds of states
  - Photos are resources and they have a state → **resource state**
  - A client accessing those resources has a state, too → **client application state**
- Example:
  - Google's resources → indexed web pages.
  - Client application state → your query and your current page (different for every client because each client takes a different 'path' to its current state)



# REST principles: Statelessness

---


*Statelessness in REST applies to the **client application state**.*

***Resource state** is the same for every client and its state is managed on the server.*

- A web service should only know about client application state when a request is made → Hence, the term “REST.”
- A client should be in charge of managing its own path through the application
- The server might send a page with links telling the client about other requests it might want to make in the future - and forget about the client until the next request

# REPRESENTATIONS



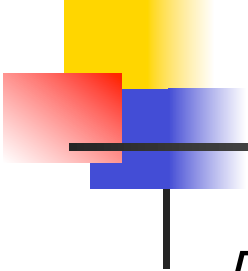


# REST principles: Resource Representations

---

*A resource needs a representation for it to be sent to the client*

- Representation of a resource - some data about or a view of the 'current state' of a resource
  - Example: a list of open bugs can be represented in XML, a web page, comma-separate-values, printer-friendly-format, etc.
- A representation of a resource may also contain metadata about the resource
  - Example: books may come with metadata such as cover-image, reviews, stock-level, etc.
- Representations can flow the other way too: a client sends a 'representation' of a new resource, and the server creates the actual resource.



# REST principles: Resource Representations


---

*Deciding between multiple representations*

## *Option 1*

- Have a distinct URI for each representation of a resource:
  - <http://www.example.com/release/104.en> (English)
  - <http://www.example.com/release/104.es> (Spanish)
  - <http://www.example.com/release/104.fr> (French)

This meets the addressability and statelessness principles in that the server knows all info necessary for the server to fulfill the request



# REST principles: Resource Representations

---

*Deciding between multiple representations*

*Option 2*

- Use HTTP HEAD (metadata) for content negotiation:
  - E.g., expose only <http://www.example.com/release/104>
  - Client HTTP Header request contains Accept-Language

Other types of request metadata can be set to indicate all kinds of client preferences, e.g., file format, payment information, authentication credentials, IP address of the client, caching directives, and so on.

Option 1 or 2 are both acceptable REST-based solutions...  
But the addressable URI option is preferred.

Example: say you want to use an HTML page translation service which accepts URIs...  
Which representation will give you the most flexibility?

# CONNECTEDNESS

# REST principles: Links and Connectedness

*In REST, resource representations are hypermedia =  
resources (data) + references to other resources (hyperlinks)*

e.g., Google Search representation:

[Jellyfish](#) ☆

**Jellyfish** are most recognised because of their jelly like appearance and this is where they get their name. They are also recognised for their bell-like ...

[www.reefed.edu.au](http://www.reefed.edu.au) > ... > [Corals and Jellyfish](#) - [Cached](#) - [Similar](#)

[Jellyfish](#) - [Wikipedia, the free encyclopedia](#) ☆

**Jellyfish** (also known as jellies or sea jellies) are free-swimming members of the phylum Cnidaria. **Jellyfish** have several different morphologies that ...

[Terminology](#) - [Anatomy](#) - [Jellyfish blooms](#) - [Life cycle](#)

[en.wikipedia.org/wiki/Jellyfish](http://en.wikipedia.org/wiki/Jellyfish) - [Cached](#) - [Similar](#)

Searches related to **jellyfish**

[jellyfish facts](#)

[types of jellyfish](#)

[jellyfish pictures](#)

[blue bottle jellyfish](#)

[jellyfish stings](#)

[jellyfish photos](#)

[jellyfish reproduction](#)

[jellyfish life cycle](#)

Go o o o o o o o o o o g l e ▶  
1 2 3 4 5 6 7 8 9 10 [Next](#)



# REST principles: Links and Connectedness

---

R. Fielding talks about:

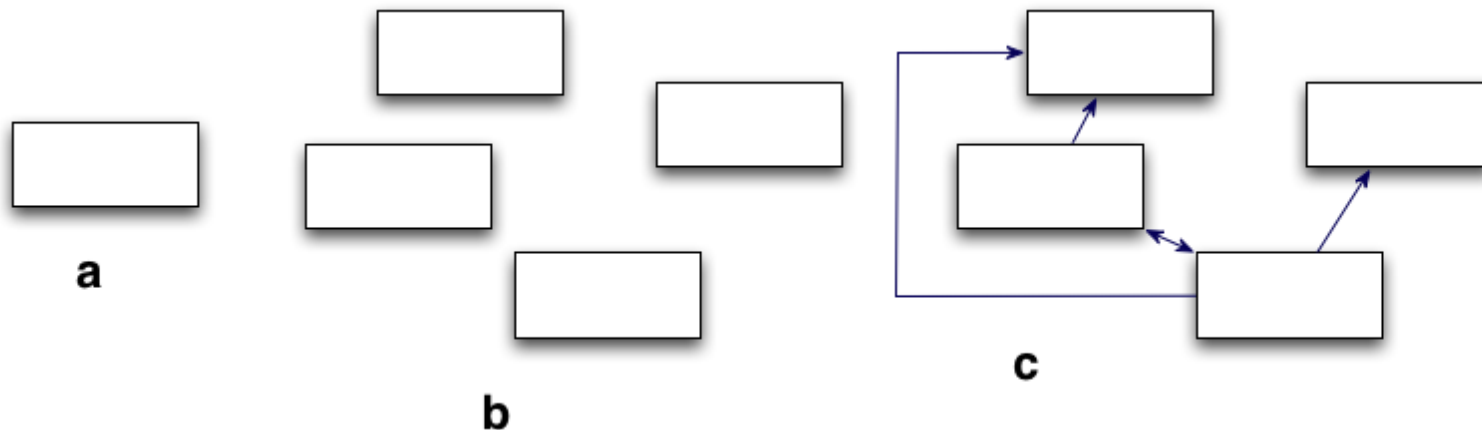
*"Hypermedia as the engine of application state"*

*The current state of an HTTP `session' is not stored on the server as a resource state, but tracked by the client as an application state, and created by the path the client takes through the Web. The server guides the client's path by serving hypermedia: links and forms inside resource representations.*

The server provides the client with guidelines about *which states are reachable from the current one.*

# REST principles: Links and Connectedness

Connectedness in REST: 3 example services



- All three services expose the same functionality, but their usability increases towards the right
- Service A is a typical RPC-style, exposing everything through a single URI. Neither addressable, nor connected
- Service B is addressable but not connected; there is no indication of the relationships between resources. A hybrid style ...
- Service C is addressable and well-connected; resources are linked to each other in ways that make sense. A fully RESTful service

# UNIFORM INTERFACE




# REST principles: Uniform Interface

*The REST Uniform Interface Principle has 4 main HTTP operations on resources*

- Retrieve a representation of a resource: **GET**
- Create a new resource: **PUT** to client-decided URI, or **POST\*** if server decides URI
- Modify an existing resource: **PUT/POST** to URI
- Delete an existing resource: **DELETE** URI
- Get metadata about an existing resource: **HEAD**
- See which of the verbs the resource understands: **OPTIONS**

Similar to the **CRUD** (Create, Read, Update, Delete) databases op.

\*POST: a debate about its exact best-practice usage ...



Decreasing likelihood of being understood  
by a Web server today



# REST principles: Uniform Interface

*Maurizio Marchese – IntroSDE*

Uniform

Get(URI)

Put(URI, Resource)

Delete(URI)

Non Uniform

getCustomer()

updateCustomer(Customer)

delete(customerId);



# REST principles: Uniform Interface

---

REST Uniform Interface, if properly followed, gives you two properties:

- **Safety (GET)**

Read-only operations ... The operations on a resource do not change any server state. The client can call the operations 10 times, 1000 times, it has no effect on the server state.

- **Idempotence (GET, PUT and DELETE)**

Operations that have the same "effect" whether you apply them once or more than once. An effect here may well be a change of server state. An operation on a resource is idempotent if making a request once has the same effect as making the identical request multiple times.



# REST principles: Uniform Interface

---

## *Safety and idempotence in math terms:*

- Multiplying a number by zero is idempotent:  $4 \times 0 \times 0 \times 0$  is the same as  $4 \times 0$
- Multiplying a number by one is both safe and idempotent:  $4 \times 1 \times 1 \times 1$  is the same as  $4 \times 1$  (idempotence) but also it does not change 4 (safe)

## *Safety and idempotence in REST:*

- **GET** (and **HEAD**): safe and idempotent
- **PUT**: idempotent
  - If you create a resource with PUT, and then resend the PUT request, the resource is still there and it has the same properties you gave it when you created it.
  - If you update a resource with PUT, you can resend the PUT request and the resource state won't change again
- **DELETE**: idempotent
  - If you delete a resource with DELETE, it's gone. You send DELETE again, it is still gone!
- **POST**: not safe and not idempotent



# REST principles: Uniform Interface

---

*Why do Safety and Idempotence matter ?*

- The two properties let a client make **reliable HTTP requests** over an **unreliable network**.
- Your GET request gets no response? Make another one. It's safe.
- Your PUT request gets no response, make another one. Even if your earlier one got through, your second PUT will have no side-effect.
- There are many applications that misuse the HTTP uniform interface:
  - GET <https://api.del.icio.us/posts/delete>
  - GET [www.example.com/registration?new=true&name=aaa&ph=12](http://www.example.com/registration?new=true&name=aaa&ph=12)

These types of usages, expose unsafe operations as GET operations. There are many uses of POST operations that are neither safe nor idempotent. Repeating them has consequences ...



# REST principles: Uniform Interface

---

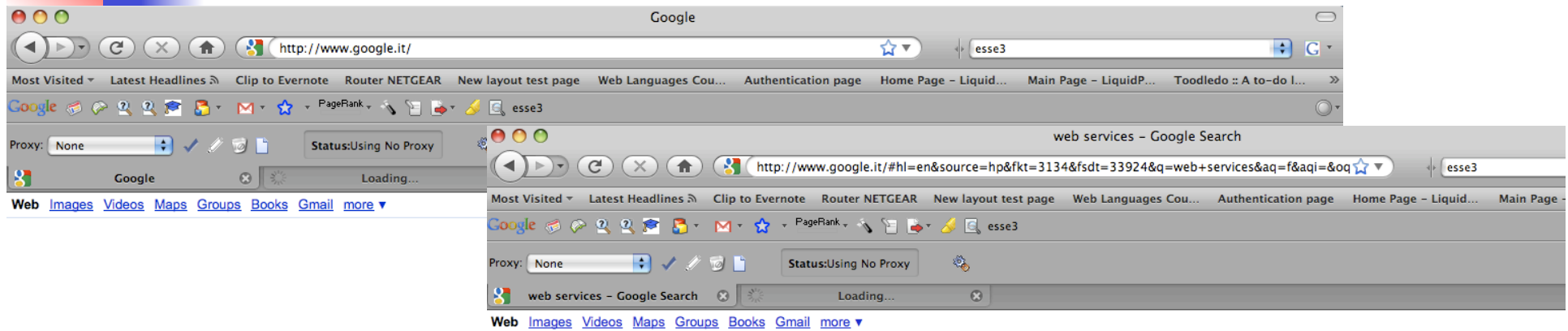
*Why does a Uniform Interface matter?*

## **Uniformity**

The point about the REST Uniform Interface is in the 'uniformity': every service uses HTTP interface **the same way**.

- It means, for example, GET does mean 'read-only' all across the Web, no matter which resource you are using it on.
- It means, we do not use methods in place of GET like *doSearch* or *getPage* or *nextNumber*.
- But it is not just about using GET in your service. It is about using it the way it was meant to be used.

# Every Web Site is a "service"



Web [Show options...](#) Results 1 - 10 of about **416,000,000** for **web services**. (0.20 seconds)

**Web service - Wikipedia, the free encyclopedia**  
A **Web Service** (also Webservice) is defined by the W3C as "a software system designed to support interoperable machine-to-machine interaction over a network. ...  
[Specifications](#) - [Styles of use](#) - [Criticisms](#) - [Similar efforts](#)  
[en.wikipedia.org/wiki/Web\\_service](#) - [Cached](#) - [Similar](#)

**Web Services Tutorial**  
**Web Services** can convert your application into a **Web**-application, which can publish its function or ... The basic **Web Services** platform is XML + HTTP. ...  
[WS Intro](#) - [WS Example](#) - [Why WS](#) - [How to Use](#)  
[www.w3schools.com/webservices/default.asp](#) - [Cached](#) - [Similar](#)

**Web Services Architecture**  
This document defines the **Web Services** Architecture. It identifies the functional components and defines the relationships among those components to effect ...  
[www.w3.org/TR/ws-arch/](#) - [Cached](#) - [Similar](#)

**WebServices.Org**  
Vendor-neutral **Web Services** industry portal with news, analysis, papers, and forums.  
[www.webservices.org/](#) - [Cached](#) - [Similar](#)

**Java Web Services At a Glance**  
A free integrated toolkit to build, test and deploy XML applications, **Web services**, and **Web** applications with the latest **Web service** technologies and ...  
[java.sun.com/webservices/](#) - [Cached](#) - [Similar](#)

**Welcome to Web Services Project @ Apache**

**Sponsored Links**

**Microgen Aptitude**  
Integrating Legacy Systems with Business Process Driven SOA  
[www.microgen.com](#)

**Free Web Services Testing**  
SOAPSonar Personal is now free!  
Test SOAP, XML, and REST  
[www.crosschecknet.com](#)

**Web, XML, Web Services**  
**Web** app security by Bee Ware  
WAF, **Web** SSO, XML Gateway, Audit  
[www.bee-ware.net](#)

**Web Services**  
Struggling with **Service**-based Apps?  
Take Control of Your **Web Services**.  
[www.progress.com/actional](#)

**Innovative Web Designs**  
Let Us Build Your Site  
So You Can Update It Yourself!  
[www.hopeworks.org](#)

**Web Services**

# REFERENCES





# Other References

---

- Architectural Styles and the Design of Network-based Software Architectures, Roy Thomas Fielding, 2000  
<http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>
- RESTful Web Services, Leonard Richardson and Sam Ruby, 2007, O'Reilly
- How to create a REST Protocol  
<http://www.xml.com/pub/a/2004/12/01/restful-web.html>
- Show me the Code – A RESTful Bookmark Web Service  
<http://www.xml.com/pub/a/2005/03/02/restful.html>
- The XML Bookmark Exchange Language  
<http://pyxml.sourceforge.net/topics/xbel/>
- REST Wiki  
<http://rest.blueoxen.net/cgi-bin/wiki.pl?FrontPage>
- HTTP/1.1 RFC 2616 June 1999
- Building Web Services the REST Way  
<http://www.xfront.com/REST-Web-Services.html>
- RESTful Web Services, John Cowan, 2005  
<http://mercury.ccil.org/~cowan/restws.pdf>
- REST, Roger L. Costello and Timothy D. Kehoe  
[www.xfront.com/REST-full.ppt](http://www.xfront.com/REST-full.ppt)