UDLAP
UNIVERSIDAD DE LAS
AMÉRICAS PUEBLA

Applications in Distributed Environments: *Architectures & Design*

Dr. José Luis Zechinelli Martini
joseluis.zechinelli@udlap.mx
LIS – 4021

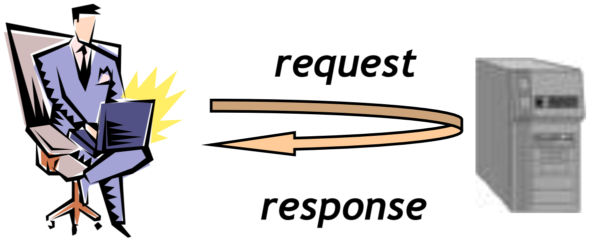
Thanks to Dr. Genoveva Vargas Solar, CNRS-LIG, Francia & Dr. Javier Espinosa, Postdoctoral fellow Centro de Supercómputo, Barcelona, for sharing material about this topic.

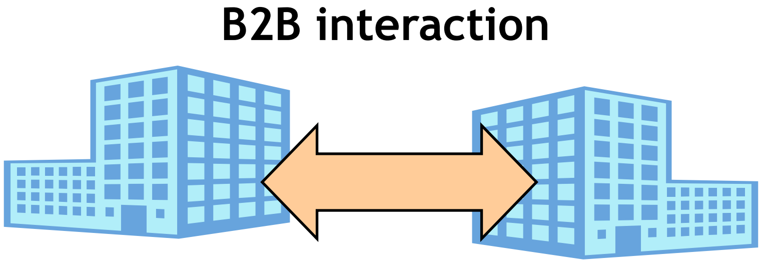
Web Design Planning

http://www.ironpaper.com/webintel/articles/web-design-statistics-2015/#.VcJKIXhZHJI

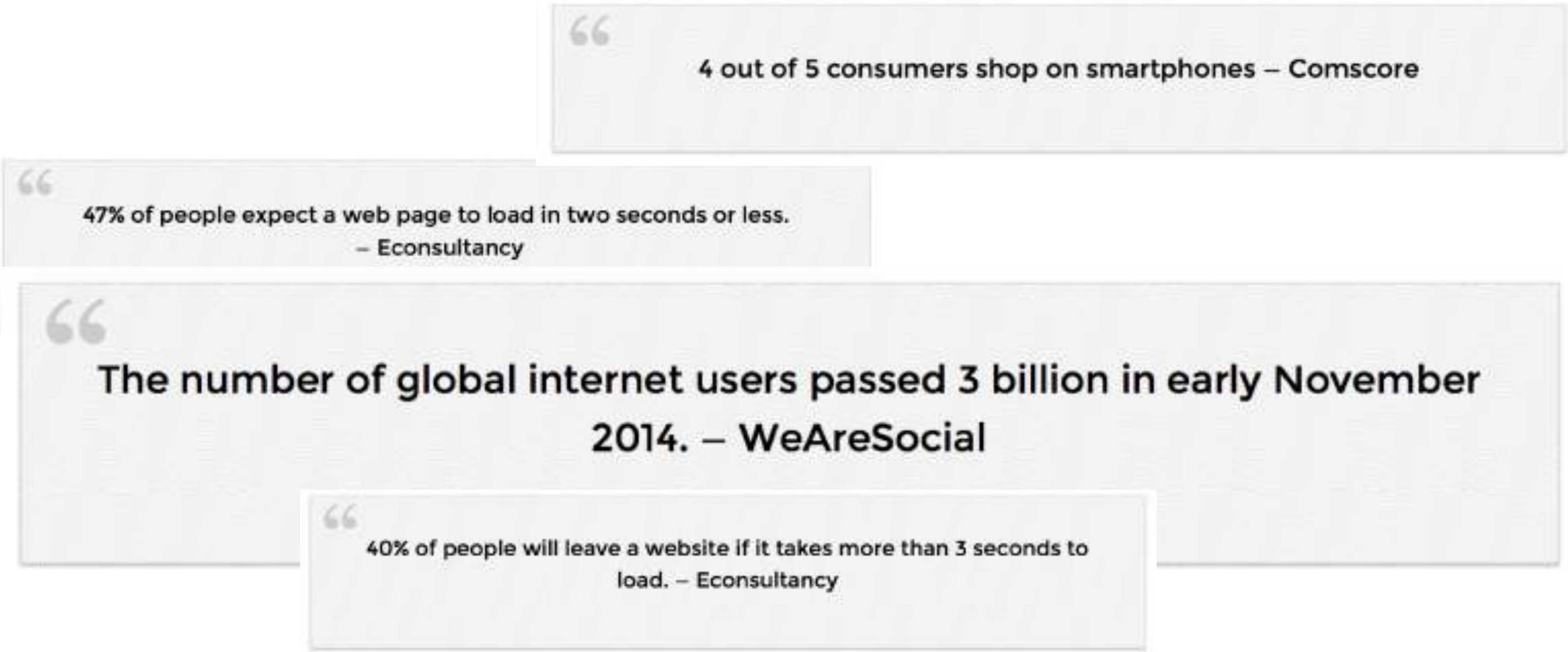
A Web App is a Distributed System

- Collection of heterogeneous networked computers which communicate and coordinate their actions by passing messages
    - □ Distribution is transparent to the user so that the system appears as a single integrated facility
    - □ Processes are not executed on a single processor but rather span a number of processors





What is so particular about Web applications?



4 out of 5 consumers shop on smartphones – Comscore

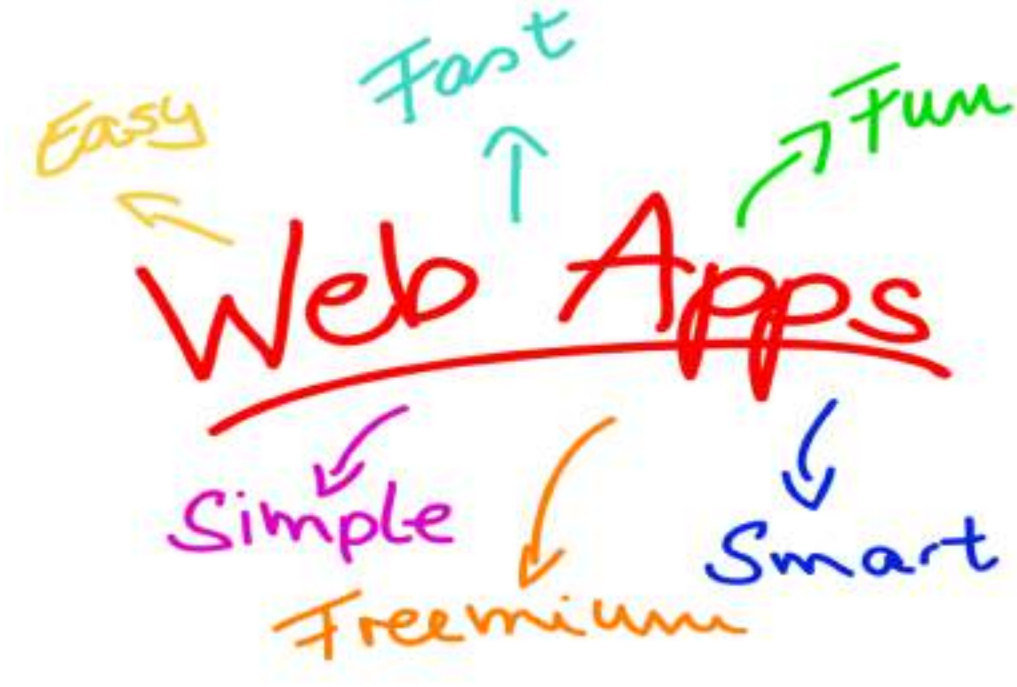
47% of people expect a web page to load in two seconds or less. – Econsultancy

The number of global internet users passed 3 billion in early November 2014. – WeAreSocial

40% of people will leave a website if it takes more than 3 seconds to load. – Econsultancy

http://www.ironpaper.com/webintel/articles/web-design-statistics-2015/#.VcJKIXhZHJI

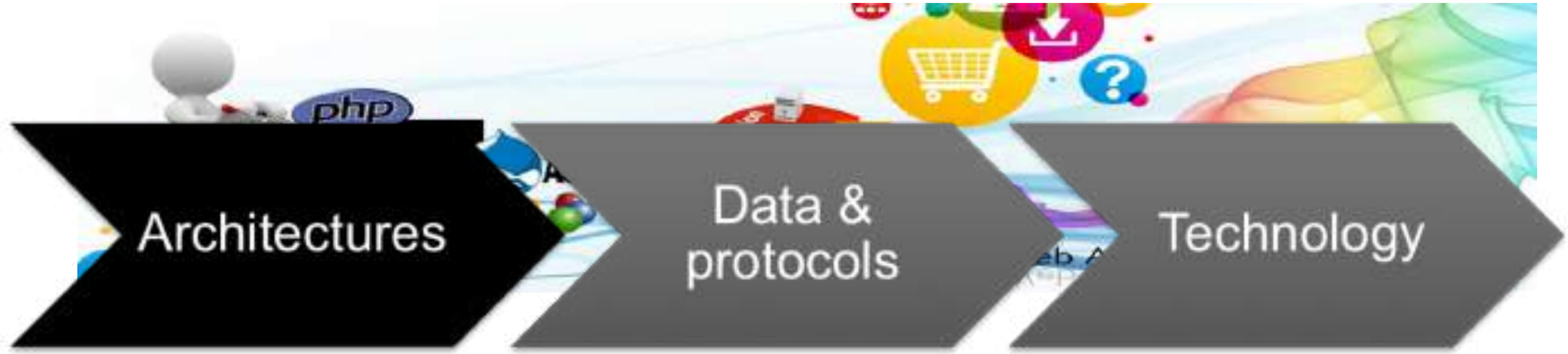
Requirements



Developing Web Applications

COMPLETE PLAN
DESIGN
DEVELOPMENT
MARKETING
MAINTENANCE

Developing Web Applications



Architectures

Data & protocols

Technology

To read

Top 10 Digital Transformation Trends For 2019

https://www.forbes.com/sites/danielnewman/2018/09/11/top-10-digital-transformation-trends-for-2019/

Web Design Trends for 2019

https://www.awwwards.com/web-design-trends-2019.html

Syllabus & organization

Objectives and Learning Outcomes

Teach students fundamental concepts and show them how they are applied in the construction of Web applications:

- Understand the characteristics of:
  - Web applications architecture models
  - Data/document languages and standards (HTML, CCS, Javascript, Ruby, Python, Ajax)
  - Different Web development tools
- Master fundamental use of:
  - JavaScript for creating interactive Web pages
  - Asynchronous JavaScript and XML for enhanced Web interaction and applications

Evaluation

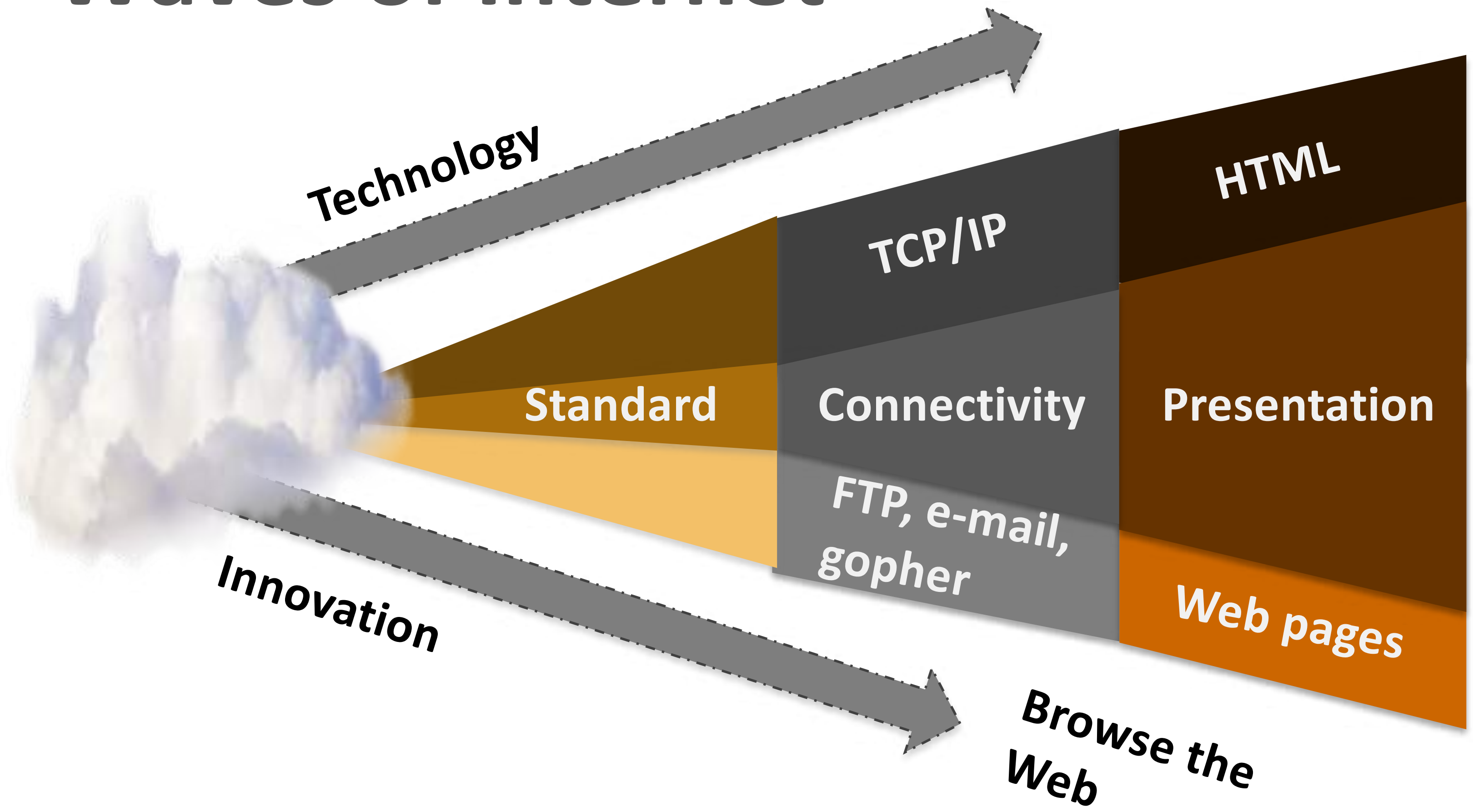
First partial exam	15%	(according to the official planning)
Second partial exam	15%	(according to the official planning)
Third partial exam	15%	(according to the official planning)
Final exam	15%	(according to the official planning)
Hands on HTML, CSS	10%	(first week of October)
Hands on JavaScript	15%	(first week of November)
Hands on Frameworks	15%	(first week of December)

http://portafolios.udlap.mx/portafolios/joseluis.zechinelli/

Plan

- ✓ Context and motivation
- The Web as a content provider
- The Web as a service provider
- Distributed architecture models

Waves of Internet



Web 1.0: Content centred

- Few content creators with the vast majority of users acting as **content consumers**
- Personal web pages were common, consisting of **static pages** hosted on **web hosting servers**
- Content served from the server's **files system** instead of a **RDBMS**
- Pages built using **Server Side Includes** or **CGI**
- **HTML 3.2-era elements** such as **frames** and **tables** to position and align elements on a page
- Proprietary HTML extensions
- HTML forms sent via email

URL Schema and Syntax (i)

- **URL Schema**
  - Identifies and provides means for **locating a resource**

    scheme: // host [: port] path [? query] [# fragment]

    e.g., http://portafolios.udlap.mx/portafolios/joseluis.zechinelli/LIS-4021/_layouts/15/start.aspx#/

Allowed characters

0 … 9	Digit
A … Z	Alphabet
- … ~	ASCII symbols

Reserved characters

/ ? # [ ] @ :

! $ & ‘ ( ) * + , ; =

URL Schema and Syntax (ii)

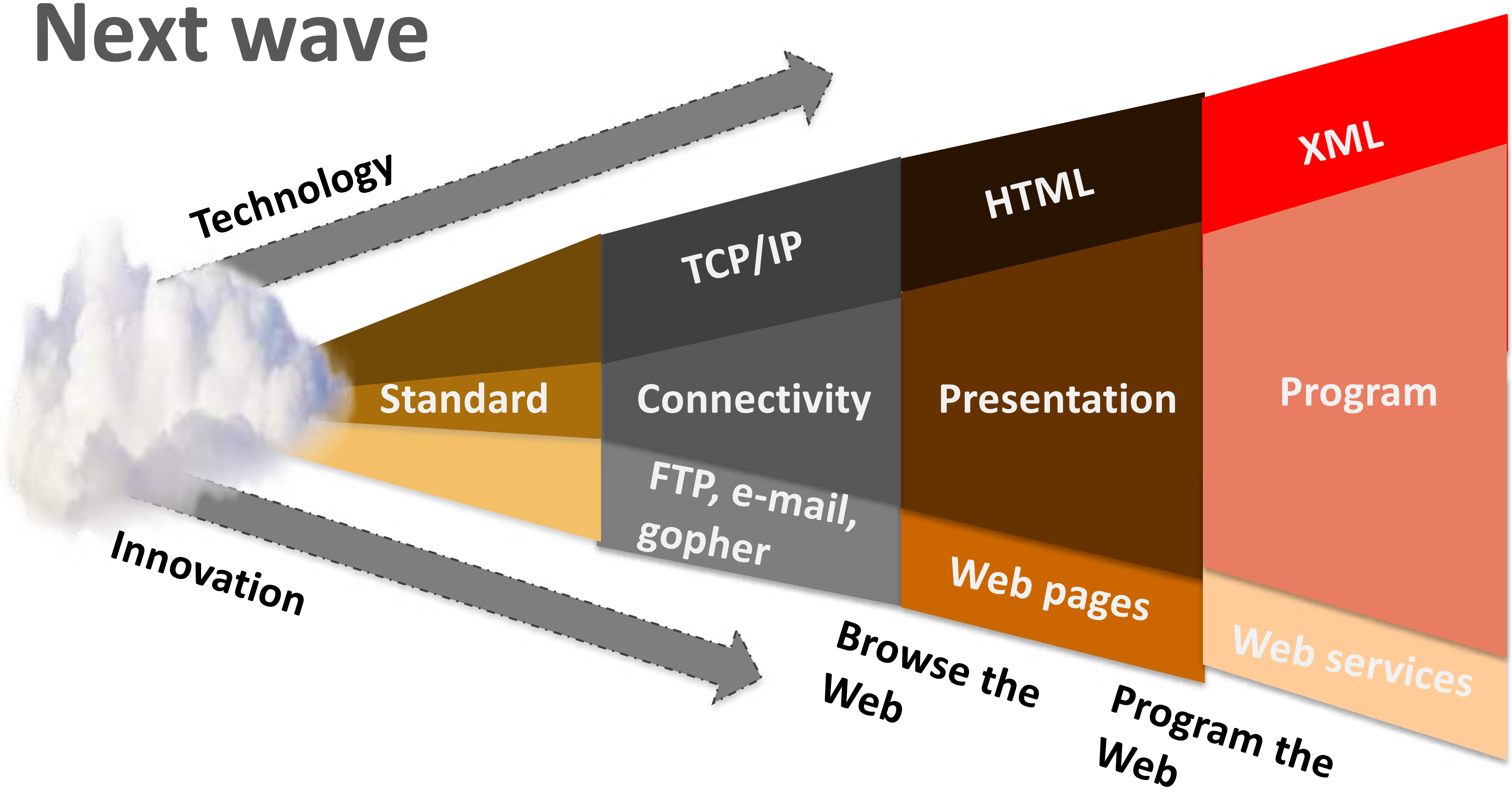
- Principle:
  - □ Encodes non ASCII-symbols and reserved-characters using the triple **% HEXADIG HEXADIG**
- Example:
  - □ Hi Zoé ! → Hi %20 Zo %C3 %A9

space é !

Plan

- ✓ Context and motivation
- ✓ The Web as a content provider
- The Web as a service provider
- Distributed architecture models

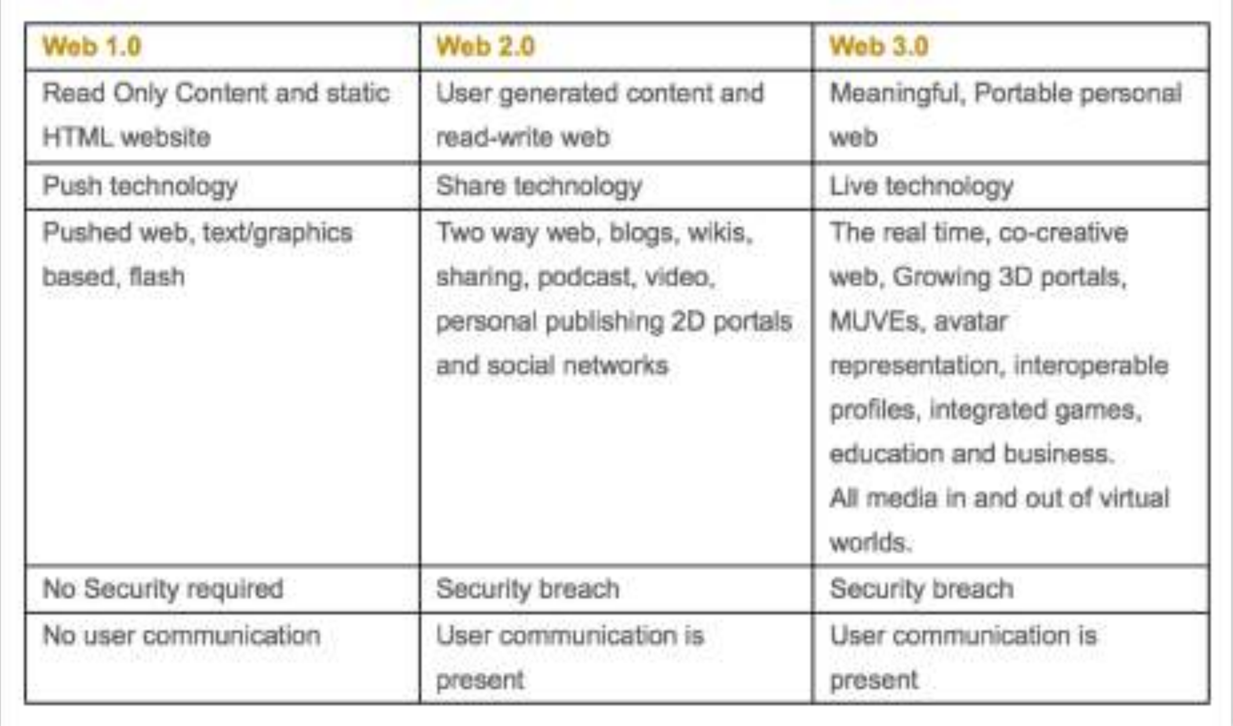
Next wave


Technology
Innovation
Standard
TCP/IP
Connectivity
FTP, e-mail, gopher
HTML
Presentation
Web pages
Browse the Web
XML
Program
Web services
Program the Web

Web 2.0

- **Folksonomy** - free classification of information; allows users to collectively classify and find information (e.g. tagging)
- **Rich User Experience** - dynamic content; responsive to user input
- **User Participation** - information flows two ways between site owner and site user by means of evaluation, review, and commenting; site users add content for others to see
- **Software as a service** - Web 2.0 sites developed APIs to allow automated usage, such as by an app or mashup

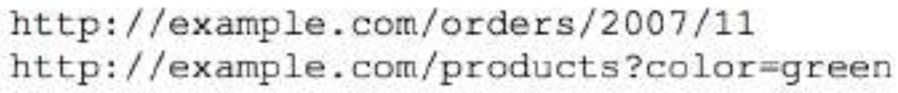
Evolution of the Web: Synthesis



Web 1.0	Web 2.0	Web 3.0
Read Only Content and static HTML website	User generated content and read-write web	Meaningful, Portable personal web
Push technology	Share technology	Live technology
Pushed web, text/graphics based, flash	Two way web, blogs, wikis, sharing, podcast, video, personal publishing 2D portals and social networks	The real time, co-creative web, Growing 3D portals, MUVEs, avatar representation, interoperable profiles, integrated games, education and business. All media in and out of virtual worlds.
No Security required	Security breach	Security breach
No user communication	User communication is present	User communication is present

Resource

- Key **abstraction** of information, data and operations:
  - Everything object (or "thing") in a system can be a resource
- Each resource is **addressable via a URI** (Uniform Resource Identifier):
  - Extensible naming schema
  - Works pretty well on global scale and is understood by practically everybody
  - Can be human-readable
- Examples:



```
http://example.com/orders/2007/11
http://example.com/products?color=green
```

http://www.facebook.com:80/joseluis.zechinelli?sk=info

URI Types

- **Uniform Resource Locator** (URL)
  - □ Identifies and provides means for locating a resource
- **Uniform Resource Name** (URN)
  - □ Persistent even if the resource ceases to exit or is unavailable

URI Schema

- Defines a **set of rules** for identifying a resource
- **Examples**

HTTP

http://vargas-solar.com

MAIL

mailto:joseluis.zechinelli@udlap.mx

GEO

geo:48.890172,2.249922

Spotify

spotify:user:jlzechinelli

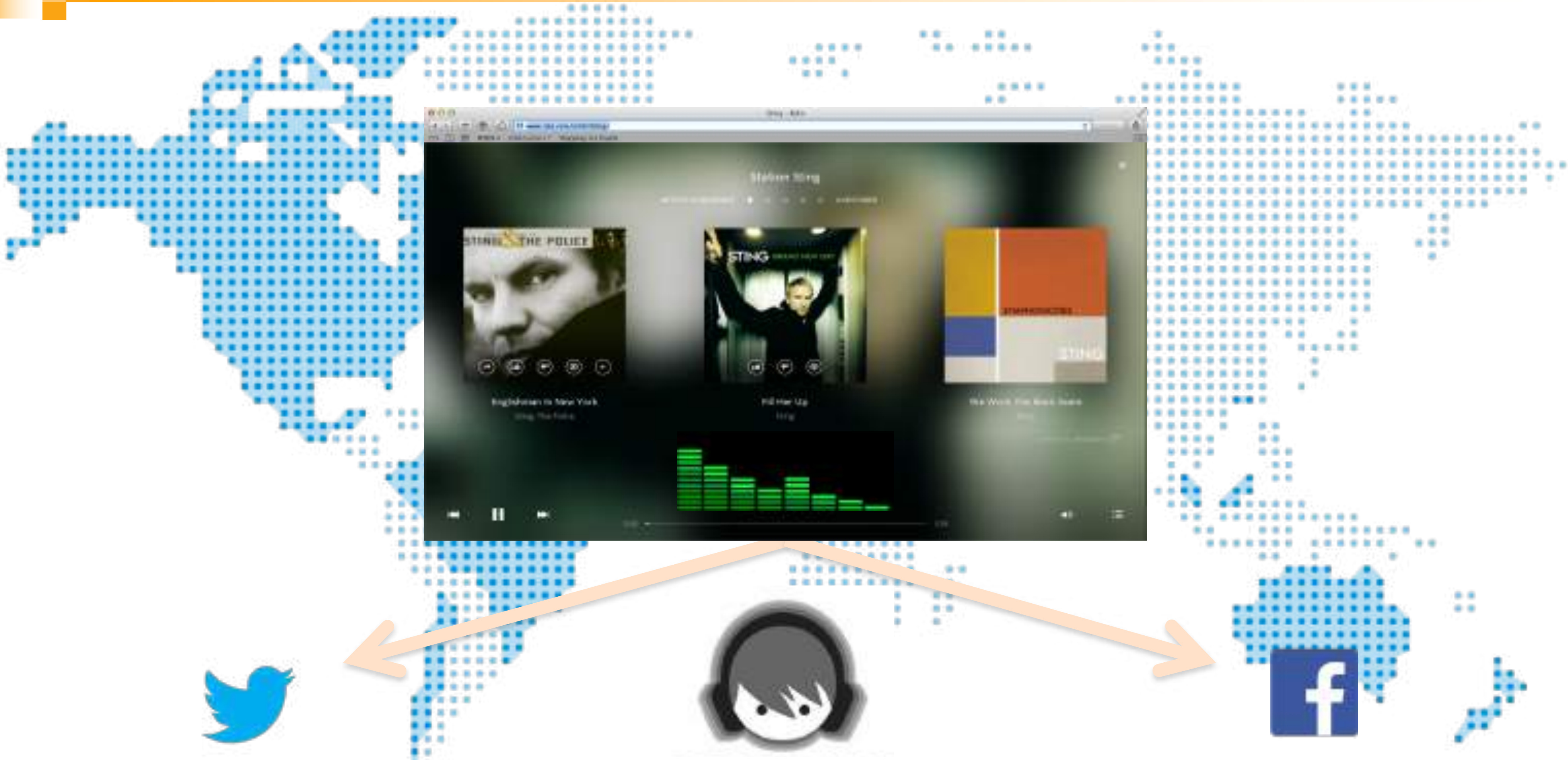
Skype

skype:joseluis.zechinelli

LastFM

lastfm://user/jlzechinelli

Spotify
DEEZER
rdio
YQL
programmableweb
TEMBOO
bing
Google



Fill Her Up - Sting

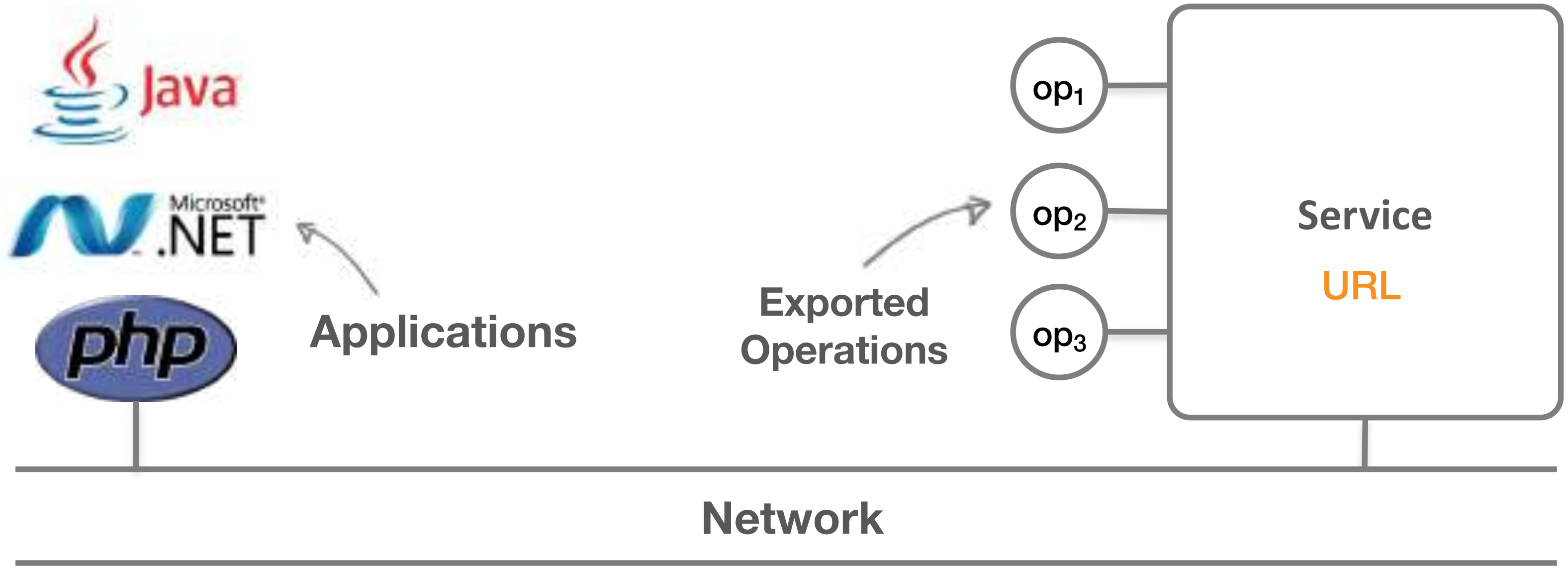
Fill Her Up - Sting

Interacting with operations

Web services use case

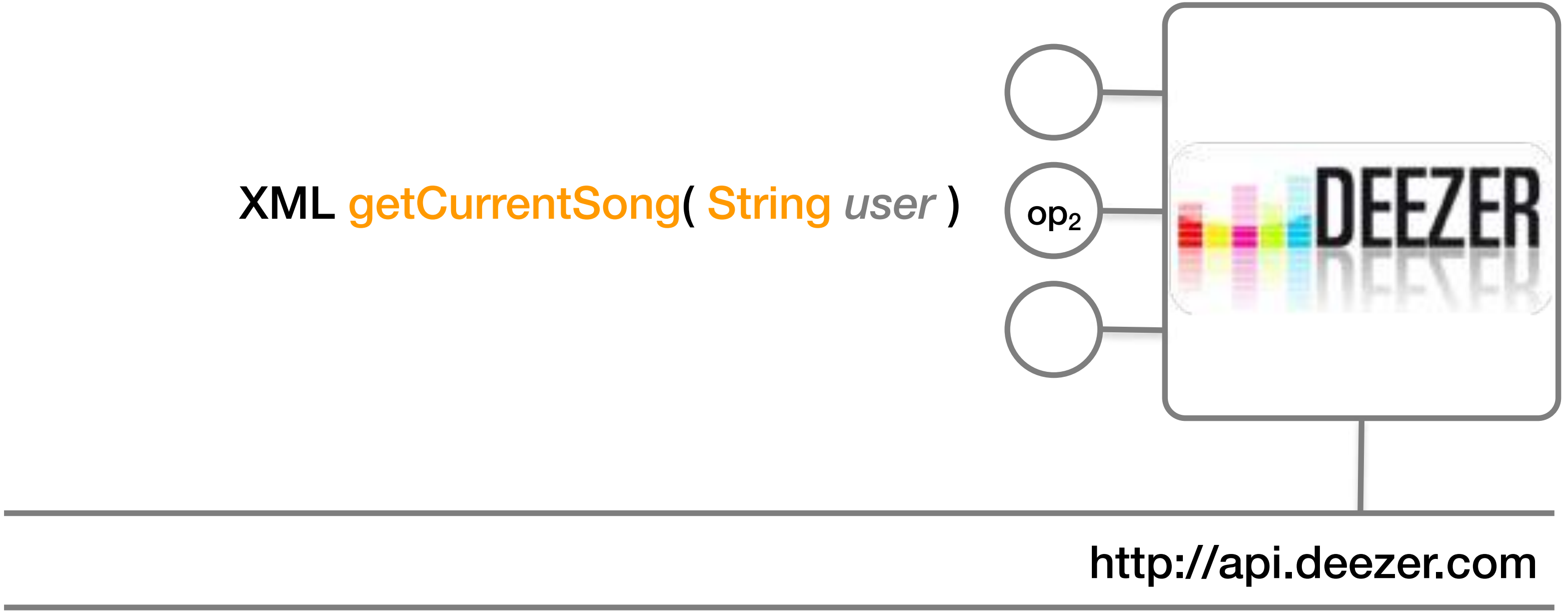
Service

- **Component** exporting **operations** accessible via a **network**



Service Example

- " Retrieve the **current song** listened by a **Deezer user** "

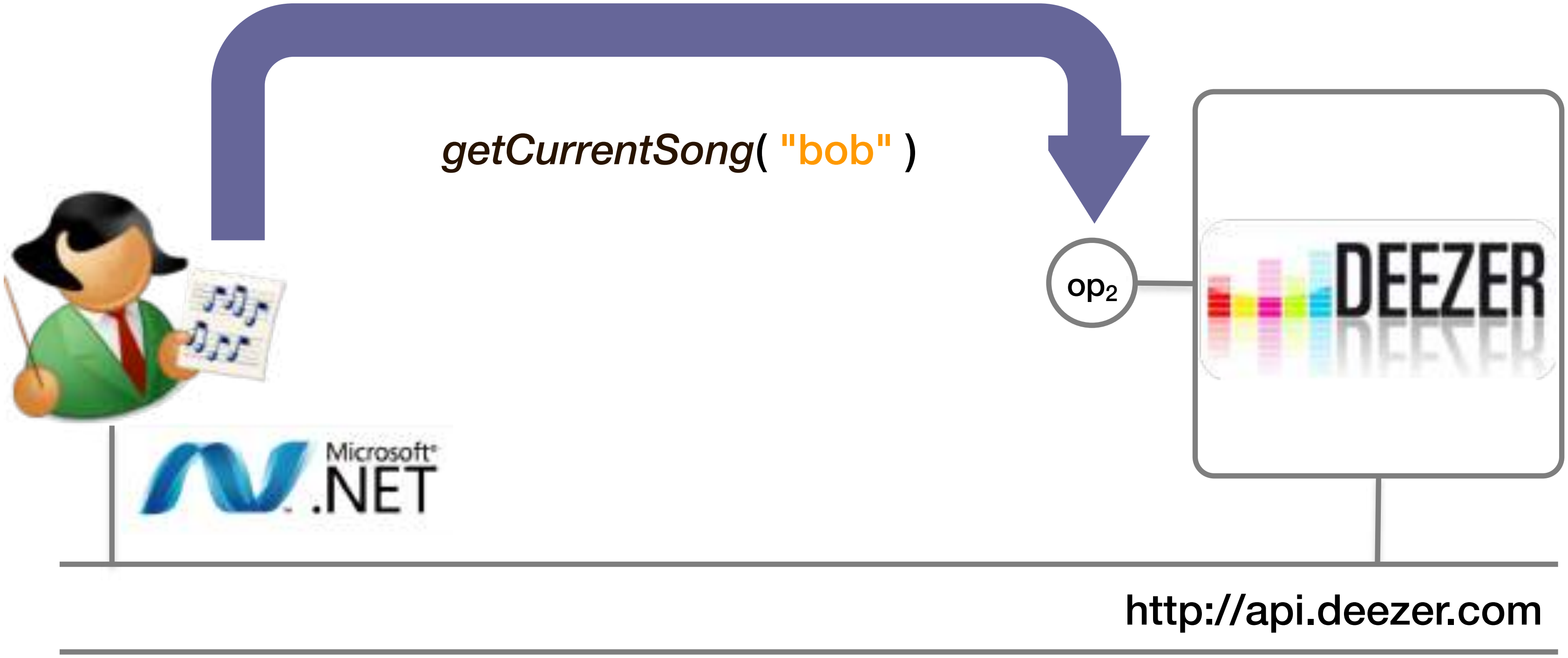


XML getCurrentSong(String *user*)

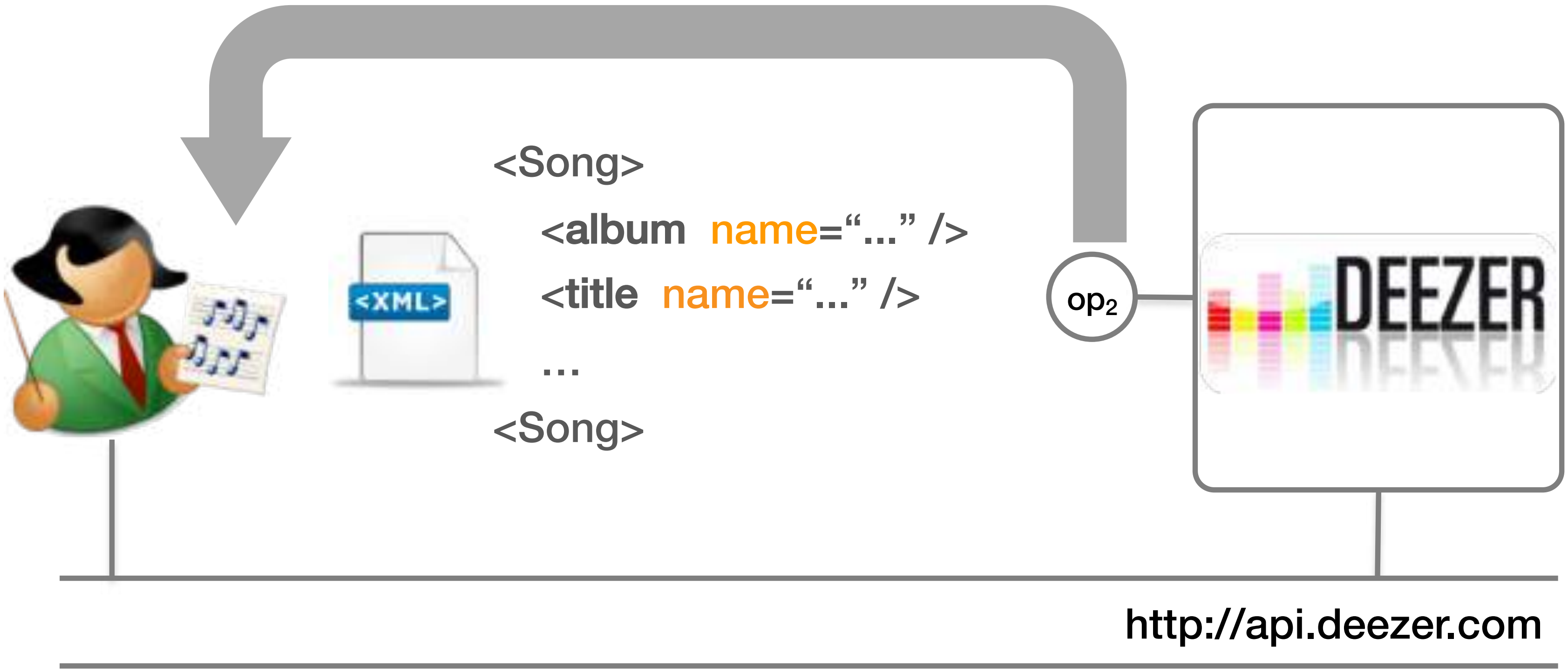
op_2

http://api.deezer.com

Operation Call Example



Operation Call Example



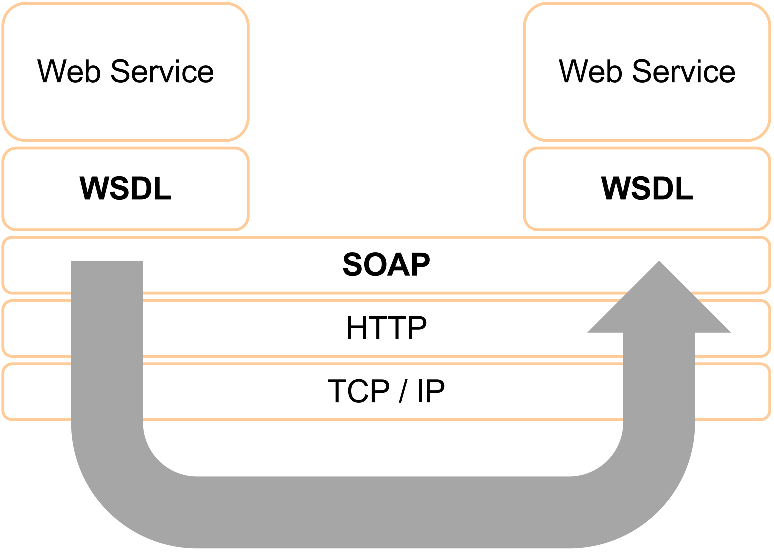
```
<Song>
   <album  name=“...” />
   <title  name=“...” />
   ...
<Song>
```

op$_2$

http://api.deezer.com

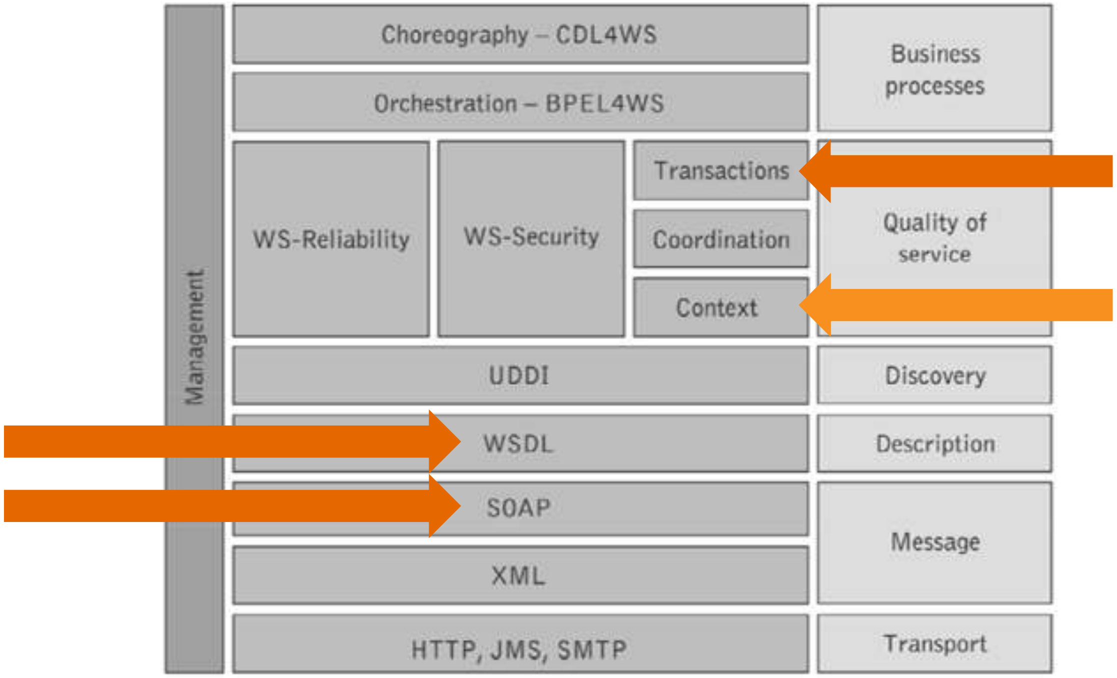
Web Service

- Provides a standard means of **interoperating** between applications running on **different platforms**
- Exposes an **API** described in a **machine-processable format** (WSDL)
- Other systems interact with it using SOAP messages that are conveyed using HTTP and other **web-based technologies**



Web Service Protocol Stack

- Set of **standards** addressing interoperability aspects



Management				
	Choreography – CDL4WS			Business processes
	Orchestration – BPEL4WS			
	WS-Reliability	WS-Security	Transactions	Quality of service
			Coordination	
			Context	
	UDDI			Discovery
	WSDL			Description
	SOAP			Message
	XML			
	HTTP, JMS, SMTP			Transport

Basic Standards

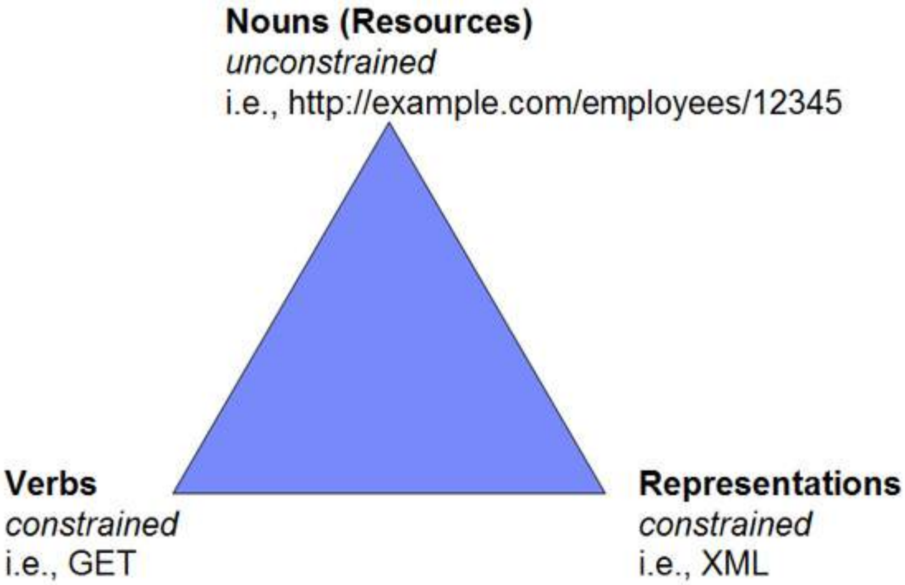
Used for managing Service state

Interacting with resources

Restful services use case

REST in a Nutshell

- REST is all about:
  - → Resources
  - → How to manipulate the resource
  - → How to represent the resource in different ways

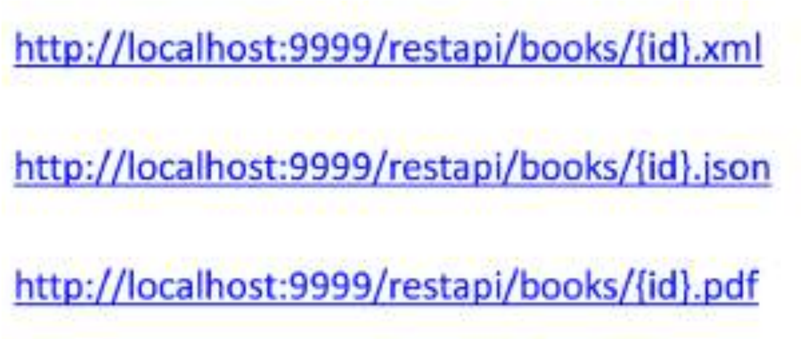


Designing a Resource Representation

- **Understandability** - Both Server and Client should be able to understand and utilize the representation format of the resource.
- **Completeness** - Format should be able to represent a resource completely. For example, a resource can contain another resource. Format should be able to represent simple as well as complex structures of resources.
- **Linkability** - A resource can have a linkage to another resource, a format should be able to handles such situations.

Representation of Resources

- A resource referenced by one URI can have **different representations**:
  - □ HTML (for browsers), XML (for application), JSON (for JavaScript)



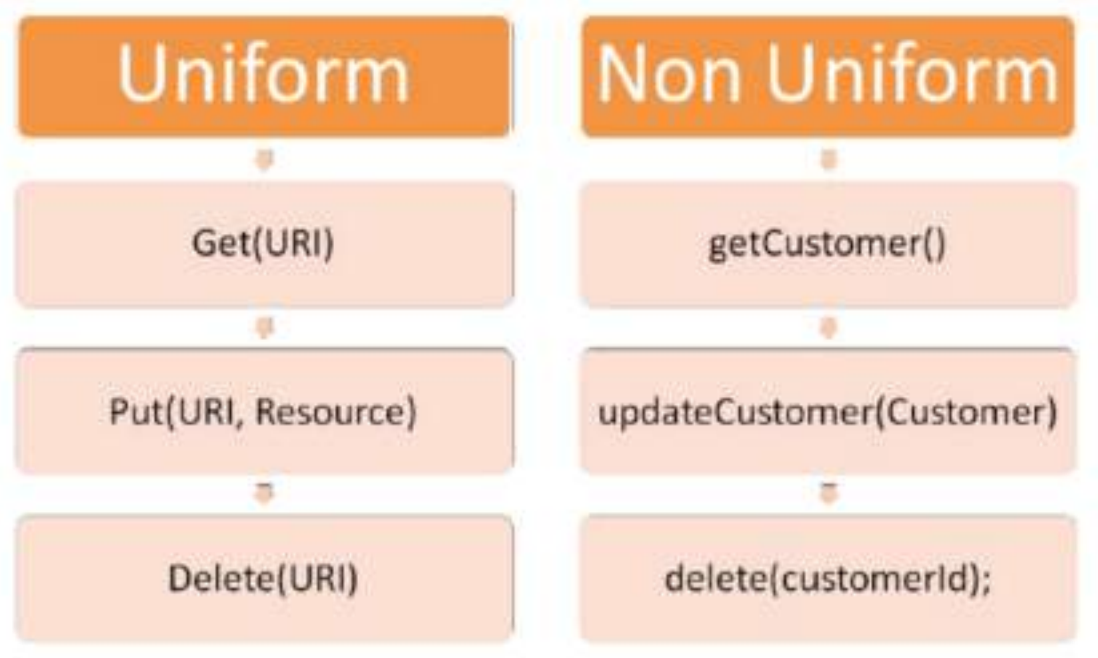
http://localhost:9999/restapi/books/{id}.xml

http://localhost:9999/restapi/books/{id}.json

http://localhost:9999/restapi/books/{id}.pdf

- If the client "knows" both the HTTP application protocol and a set of data formats then it can interact with any RESTful service in the world

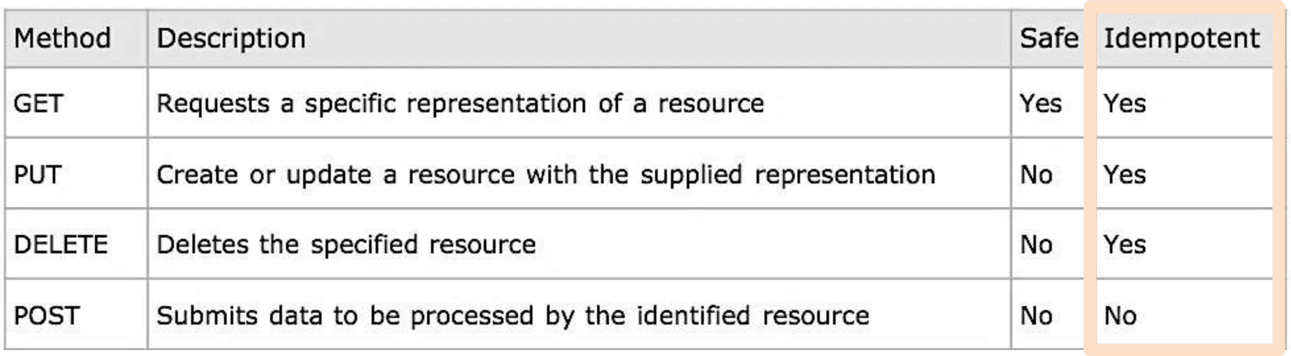
Resource Oriented VS Operation Oriented



Uniform	Non Uniform
Get(URI)	getCustomer()
Put(URI, Resource)	updateCustomer(Customer)
Delete(URI)	delete(customerId);

Interacting with Resources (i)

- All resources supports the same API (i.e. HTTP operations)
  - Each operation has a specific purpose and meaning

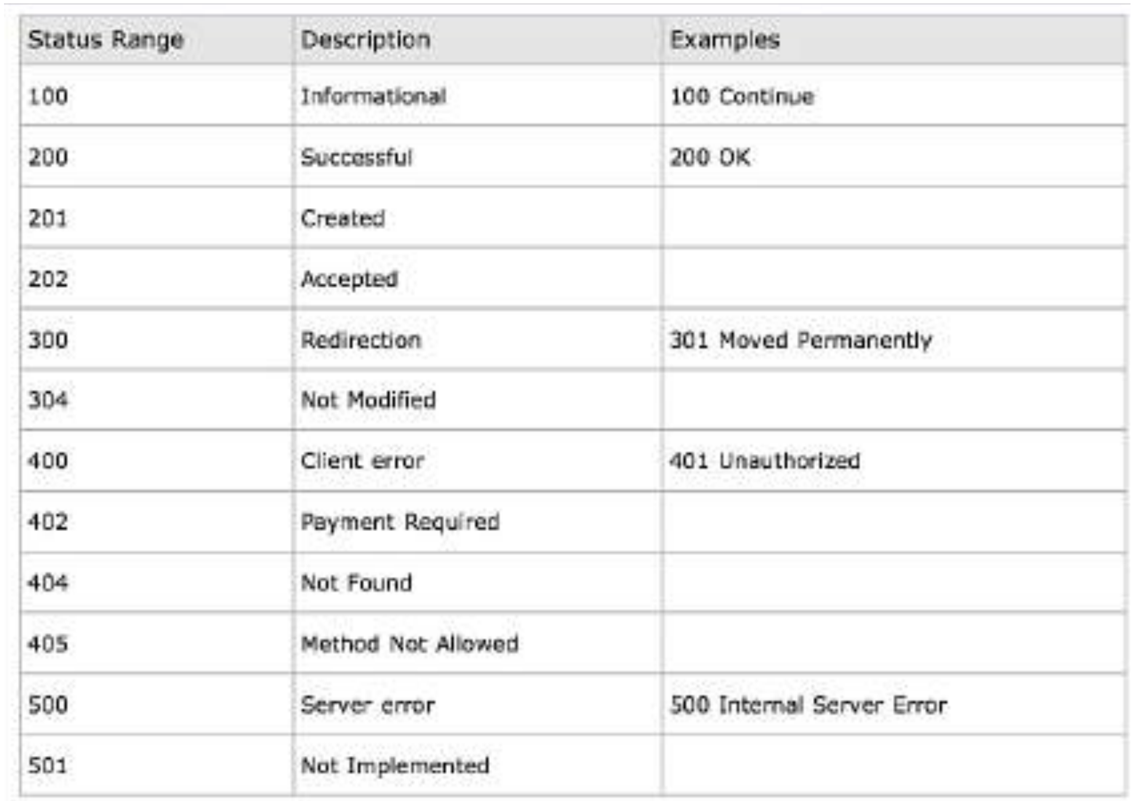


Method	Description	Safe	Idempotent
GET	Requests a specific representation of a resource	Yes	Yes
PUT	Create or update a resource with the supplied representation	No	Yes
DELETE	Deletes the specified resource	No	Yes
POST	Submits data to be processed by the identified resource	No	No

- **Note**: The actual semantics of **POST** are defined by the server

Interacting with Resources (ii)

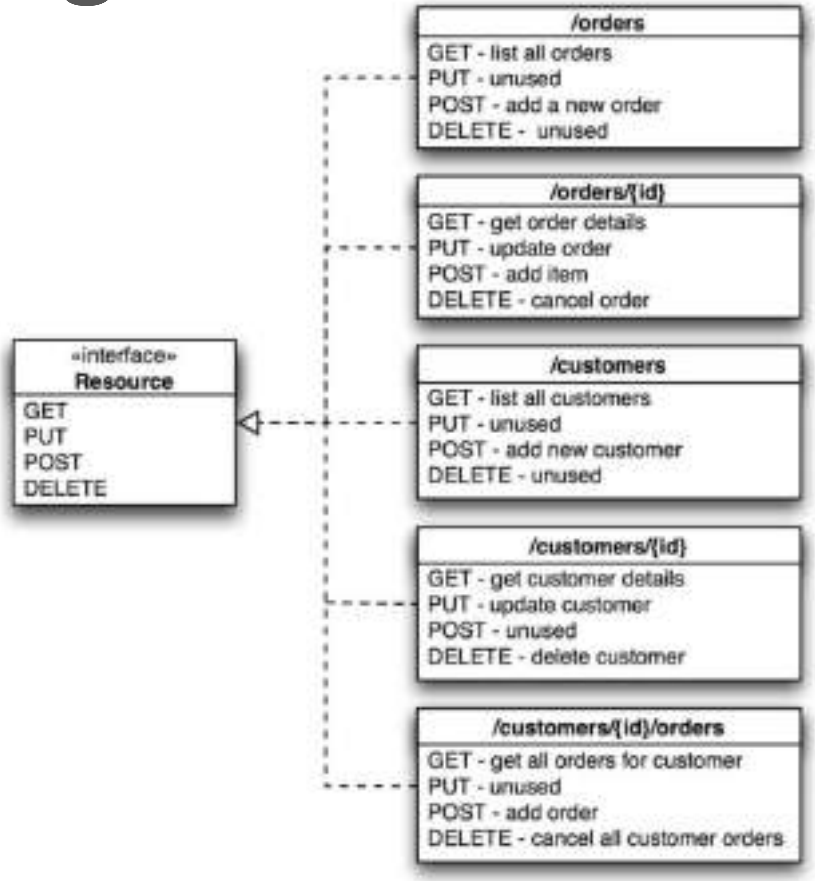
- HTTP Codes



Status Range	Description	Examples
100	Informational	100 Continue
200	Successful	200 OK
201	Created	
202	Accepted	
300	Redirection	301 Moved Permanently
304	Not Modified	
400	Client error	401 Unauthorized
402	Payment Required	
404	Not Found	
405	Method Not Allowed	
500	Server error	500 Internal Server Error
501	Not Implemented	

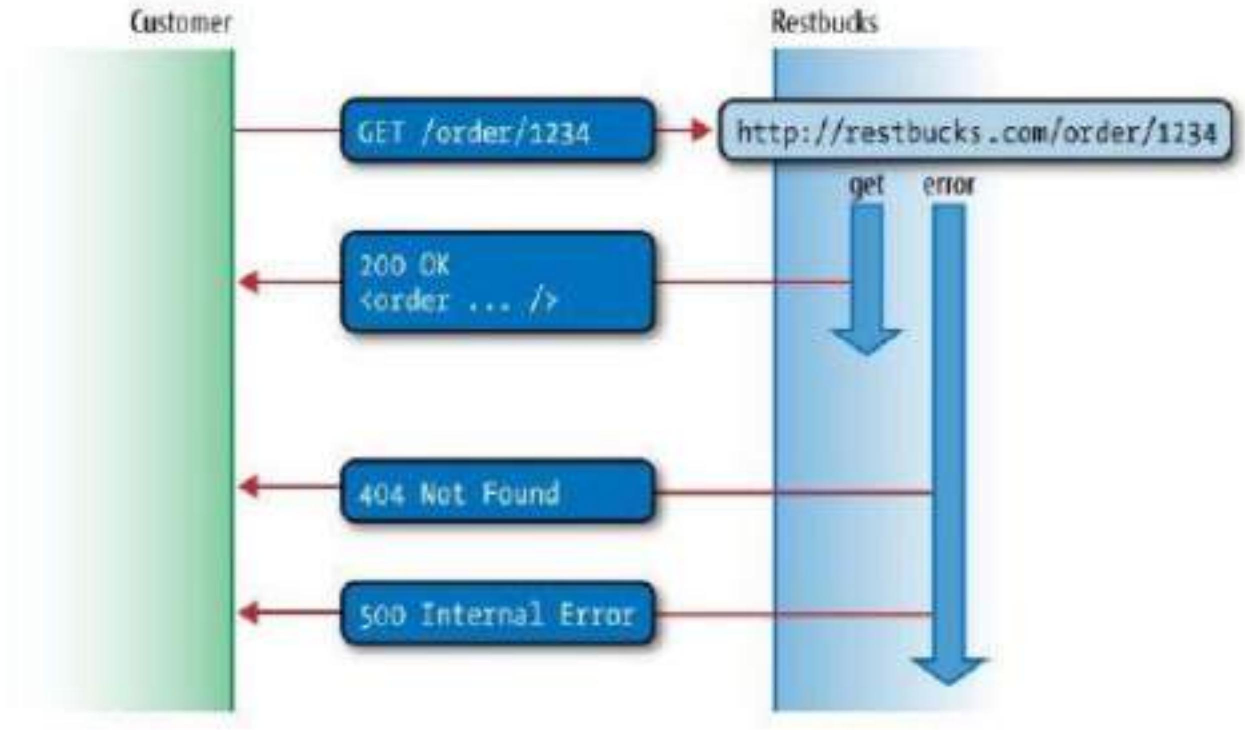
Interacting with Resources (iii)

- Example



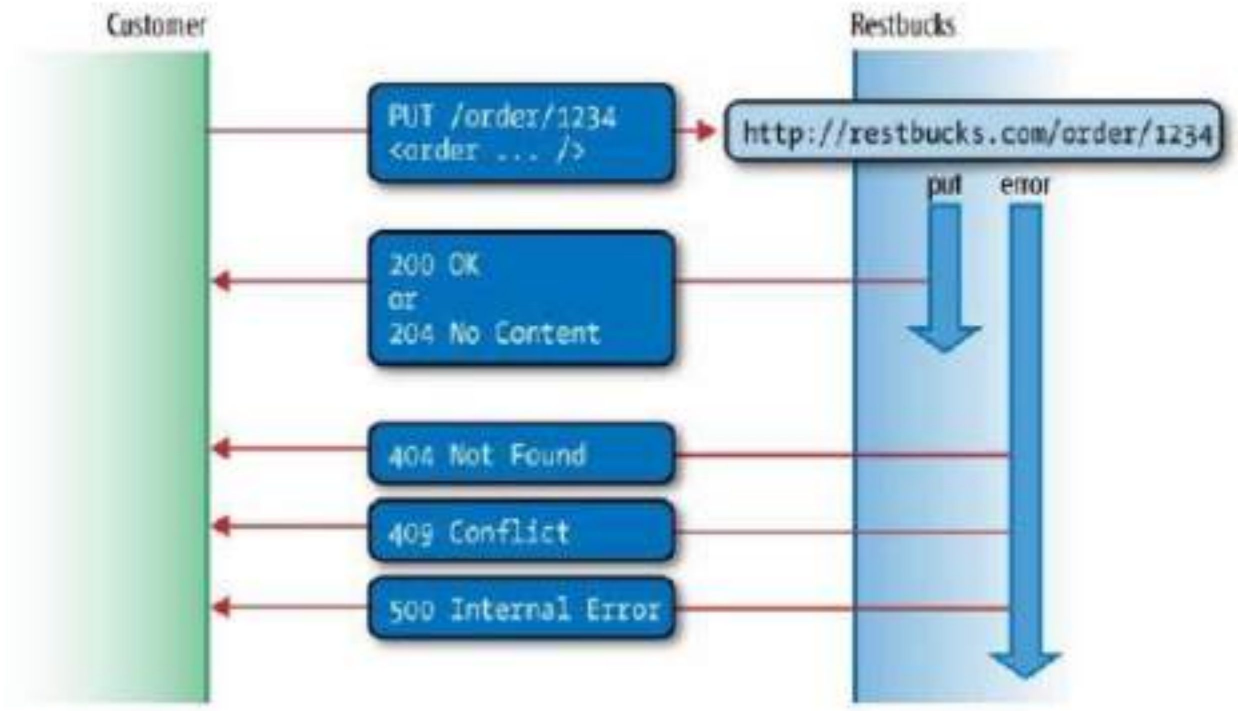
Interacting with Resources (iv)

- **Get** a resource



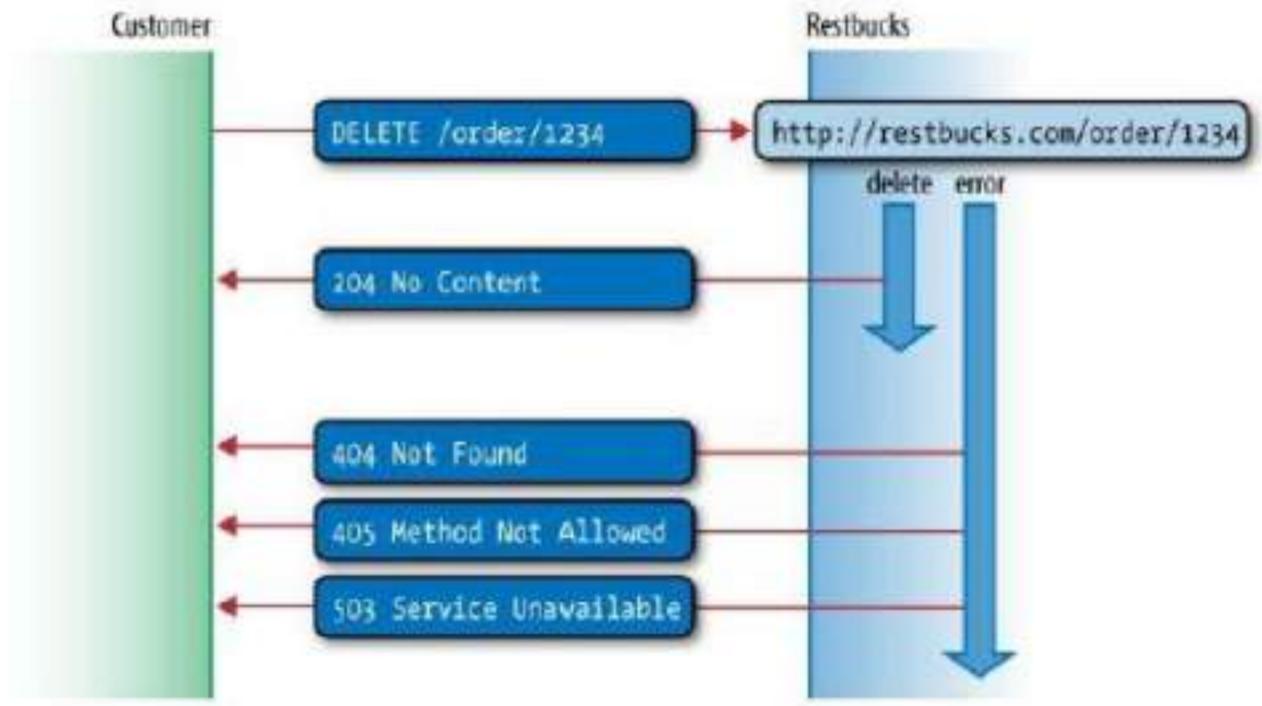
Interacting with Resources (v)

- **Create/Update** a resource



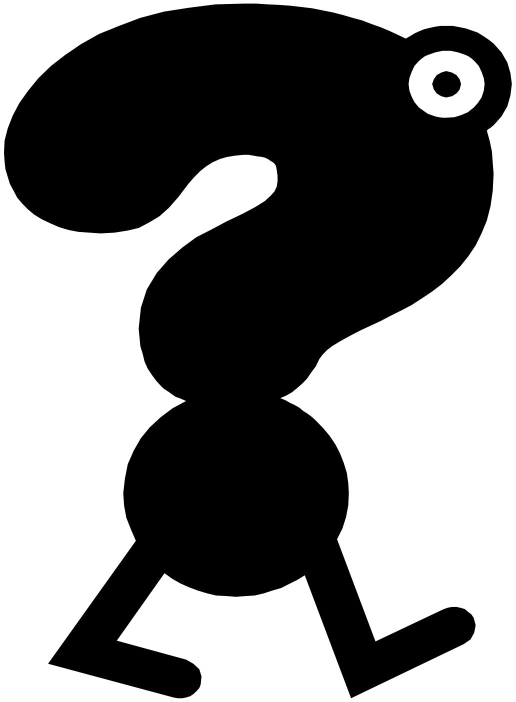
Interacting with Resources (vi)

- **Delete** a resource



To read

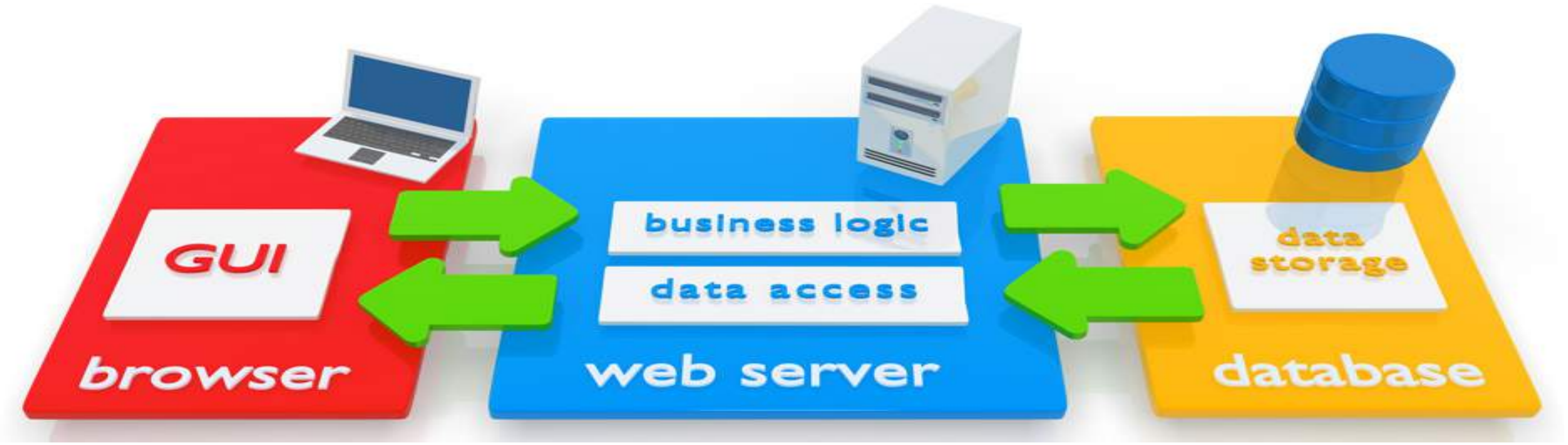
Principled Design of the Modern Web Architecture



Plan

- ✓ Content and motivation
- ✓ The Web as a content provider
- ✓ The Web as a service provider
- Distributed architecture models:
  - □ Client-Server
  - □ Design aspects
  - □ N-tier architectures

General Architecture



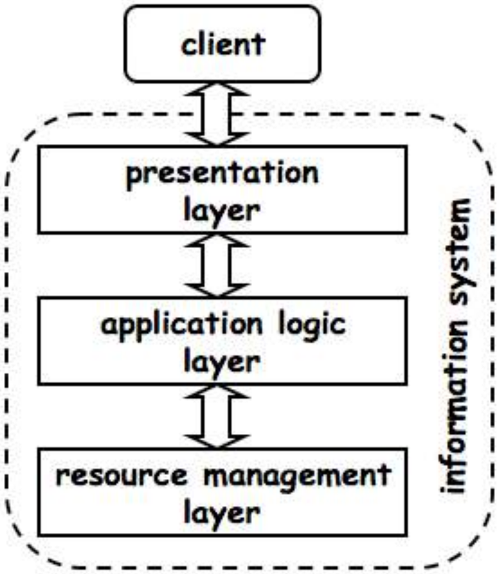
To read

Web design across different generations

https://econsultancy.com/blog/65354-web-design-across-different-generations/

System Layers

- **Presentation**: Offers operations to a client for interacting with the system
- **Application Logic**: Determines what the system actually does; enforces the business rules and establishes the business process
- **Resource Manager**: Deals with the business logic data (e.g., storage, indexing, and retrieval); it can be any system providing querying capabilities and persistence (e.g. DBMS)

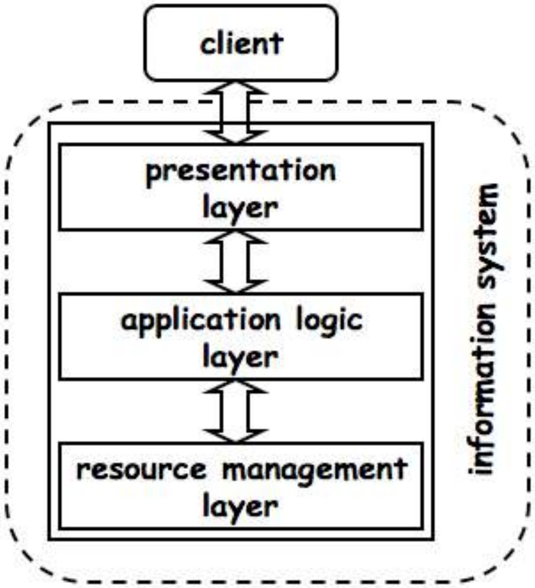


N-Tier Architecture Model

- Organizes the layers of a system based on their distribution
  - **1-Tier** (Monolithic)
  - **2-Tiers** (Client-Server)
  - **3-Tiers** (Middleware)
  - **N-Tiers**
- System architectures are represented using blocks and arrows
  - Blocks represent tiers and/or layers
  - Arrows represent communication among blocks

Monolithic (1-Tier)

- All the layers are centralized in a single place
- Managing and controlling resources is easier
- Can be optimized by blurring the separation between layers



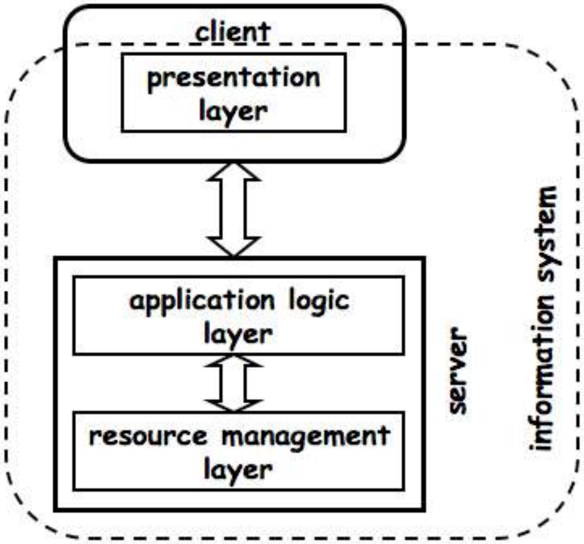
Client-Server (2-Tiers) (i)

- Several presentation layers can be defined depending on what each client needs to do
- Takes advantage of clients computing power for creating more sophisticated presentation layers

→ Saves computer resources on the server

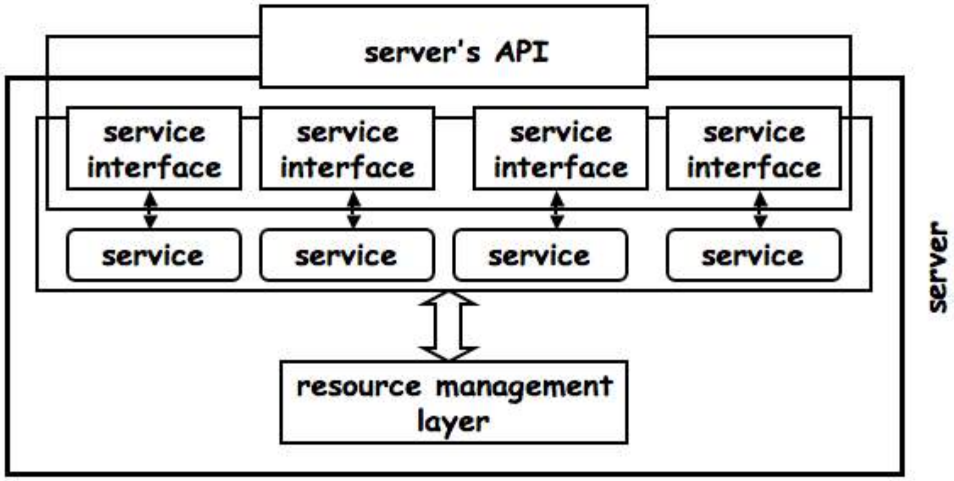
- The resource manager only sees one client: the application logic

→ Helps with performance since no extra sessions are maintained



Client-Server (2-Tiers) (ii)

- Introduces the notion of service and service interface
  - The client invokes a service implemented by a server through an interface
- All the services provided by a server define its API (Application Programing Interface)



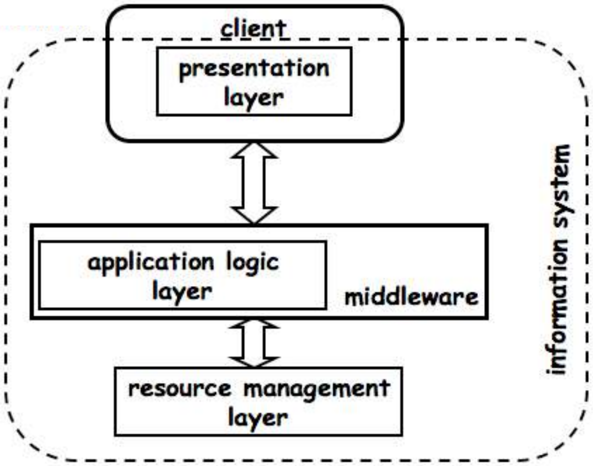
Client-Server (2-Tiers) (iii)

- Advantages
  - Can **off-load work** from server to **clients**
  - Server design is still tightly coupled and can be optimized by **ignoring presentation issues**
  - Relatively **easy to manage** from a software engineering point of view

- Disadvantages
  - A single server can only manage a **limited number** of clients
  - There is **no failure encapsulation**; if a server fails, no clients can work
  - The load created by a client will directly **affect other clients** since they compete for the same resources

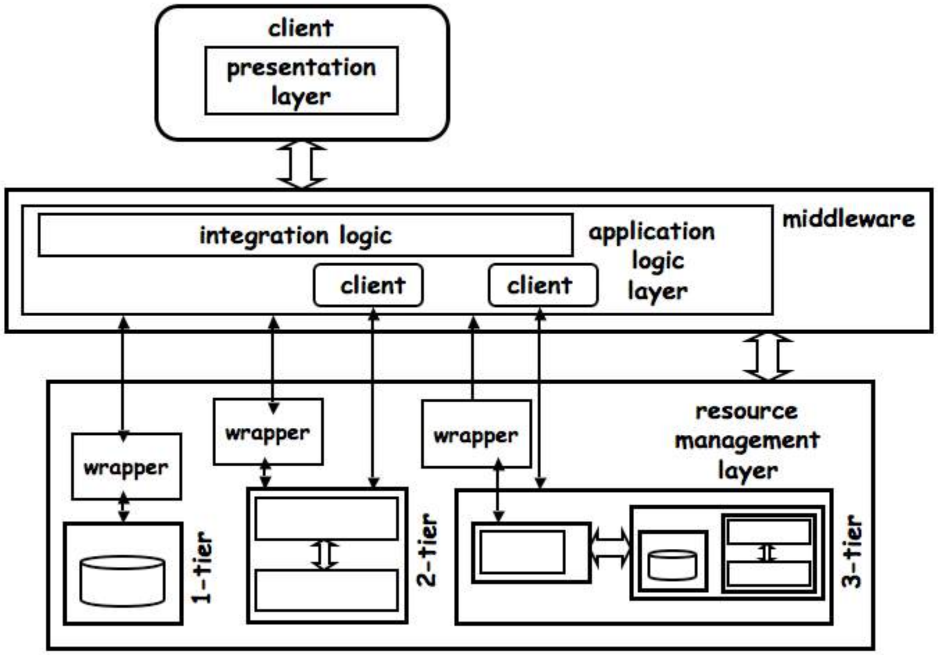
3-Tiers

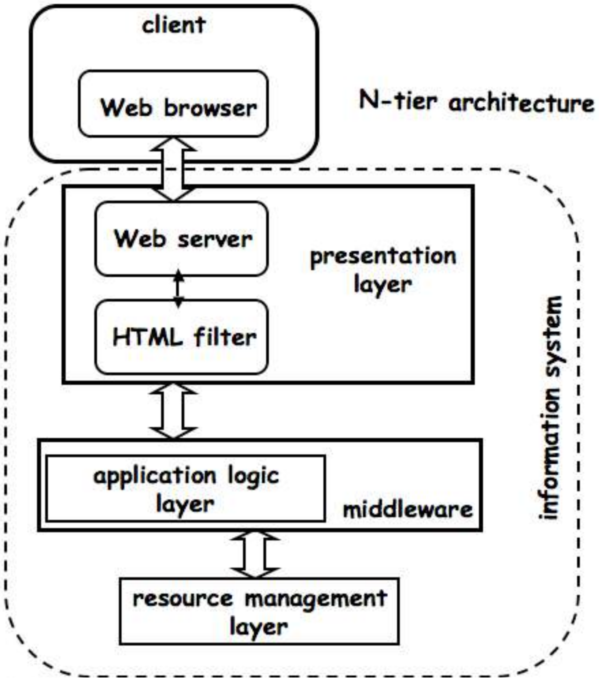
- Fully separates the three layers
- Introduces an additional layer of business logic called middleware:
  - **Simplifies the design of clients** by reducing the number of interfaces it needs to know
  - Provides **transparent access** to the underlying systems
  - Acts as a **platform for inter-system functionality** and high level application logic
  - Takes care of locating resources, accessing them, and **gathering** results



N-Tier

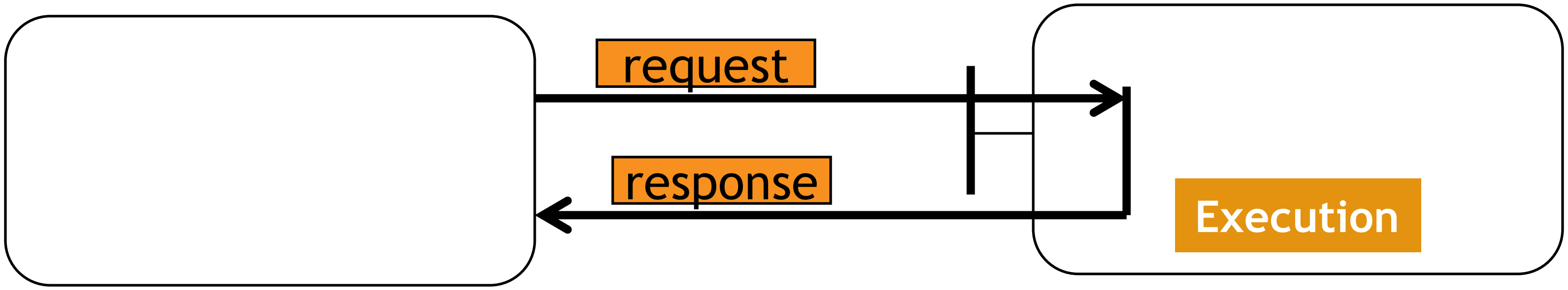
- Architecture resulting from connecting several 3-tier systems to each other.





Client – Server Model

Client-Server Abstraction



Client-Server Characteristics

- **State Management**
  - Server-side: persistent or not
  - Client-side: stateful or stateless
- **Communication Model**
  - Connected or disconnected mode (datagrams)
  - Synchronous or asynchronous
- **Server-side Execution Model**
  - One or more processes
  - Pool of processes or processes on-demand

Server without Persistent Data

- The execution only uses the input parameters:
  - Does not modify the state of the server
- Ideal situation for:
  - Fault tolerance
  - Controlling concurrency
- Example:
  - A service for computing mathematical functions

Server with Persistent Data

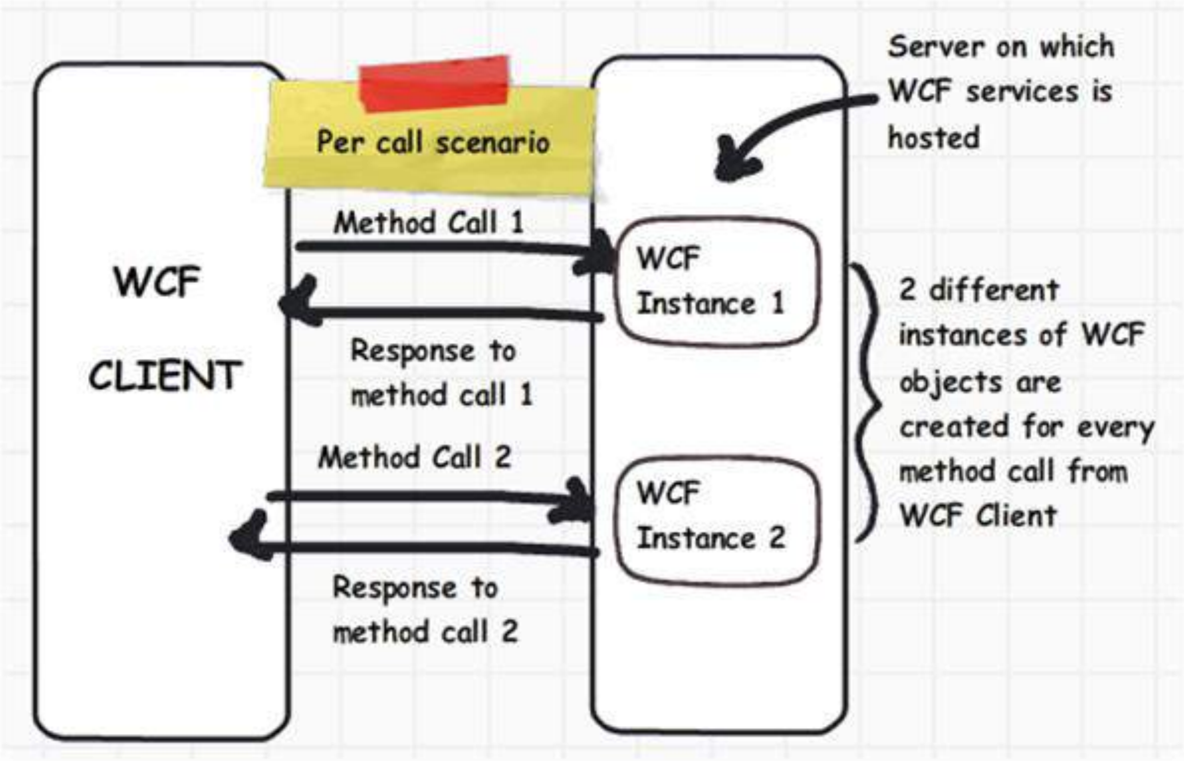
- Successive executions manipulates persistent data:
  - Modifies the execution context
  - Introduces problems for controlling concurrent access to resources
  - Fault tolerance is not guaranteed

- Examples:
  - Database Server
  - Distributed File System

Stateless Service (i)

- The server does not keep track of client requests
- Successive request are independents:
  - □ Even if global data is modified, the current request dost not have any relation with previews ones
  - □ The order among request is not important
- Example: The service of *clock synchronization* of a network
  - ➢ NTP service (Network Time Protocol)

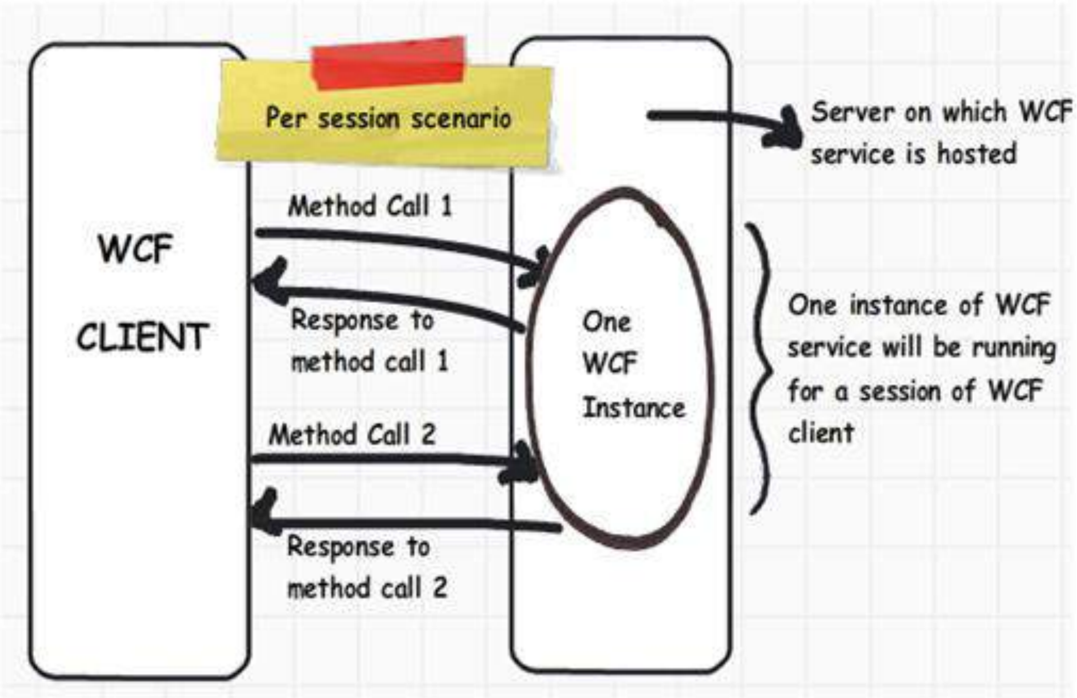
Stateless Service (ii)



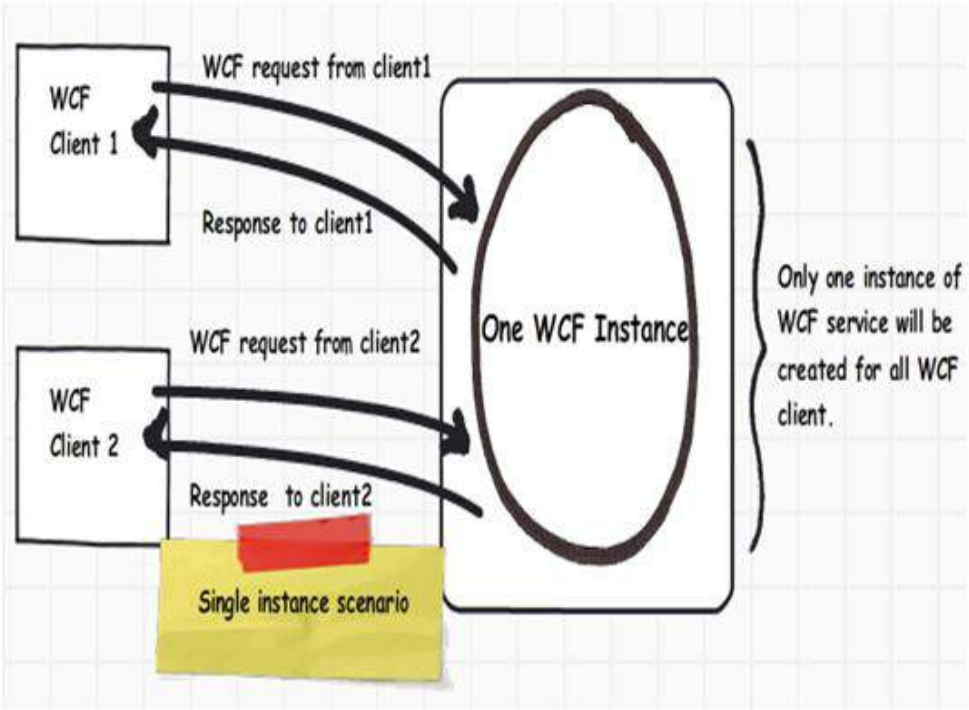
Stateful Service (i)

- Requests are executed based on the state produced by previews requests
- Order among requests is important
- Examples:
  - Sequential access to the content of a file
    - → depends on the file's pointer position
  - Calling a remote method
    - → the result of the call depends on the state of the object

Stateful Service (ii)



Stateful Service (iii)



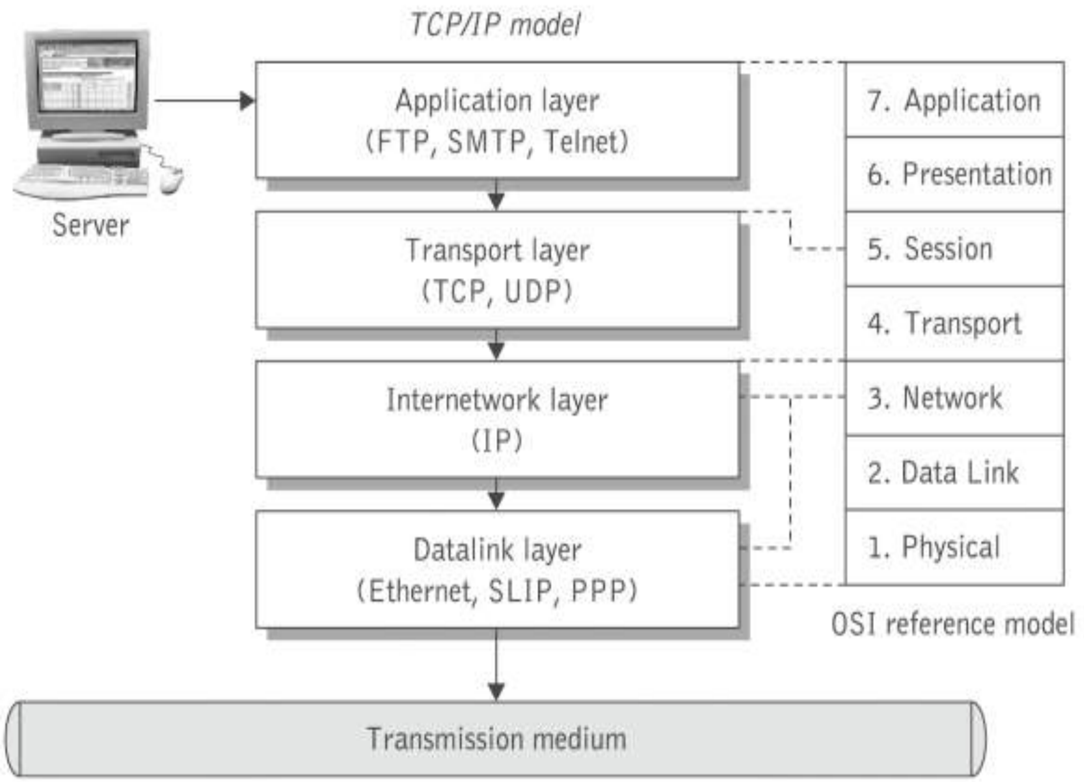
Client-Server Characteristics

- ✓ **State Management**
  - ✓ Server-side: persistent or not
  - ✓ Client-side: stateful or stateless
- **Communication Model**
  - Connected or disconnected mode (datagrams)
  - Synchronous or asynchronous
- **Server-side Execution Model**
  - One or more processes
  - Pool of processes or processes on-demand

Connection Modes

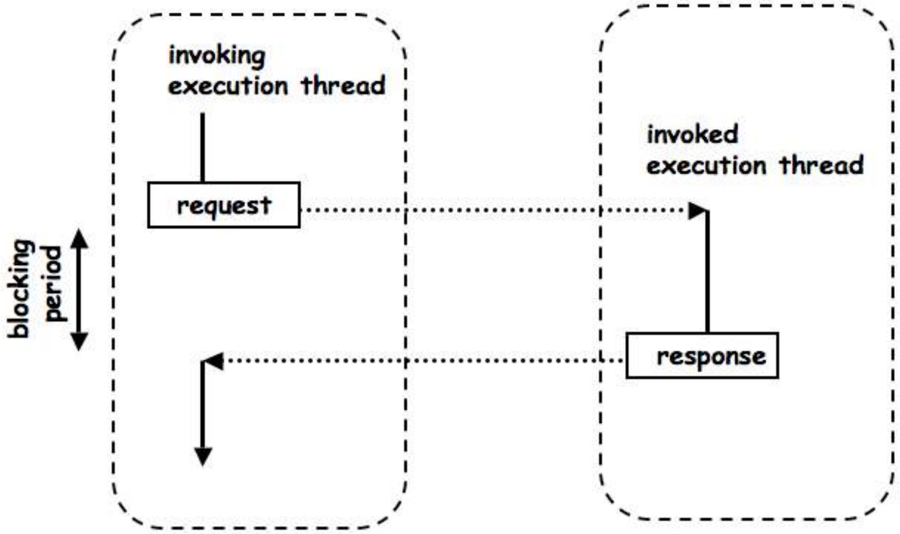
- The main difference resides in the **reliability** of message delivery
- **Connection oriented**
  - □ Message delivery is guaranteed
  - □ Order among messages is respected
  - □ Free of error (delivery is retried when necessary)
- **Datagram oriented**
  - □ Follows the "best-effort" approach (i.e., there is not guarantees of message delivery)
  - □ Message can arrived duplicated
  - □ Order is not respected

Communication Protocols



Synchronous Interaction

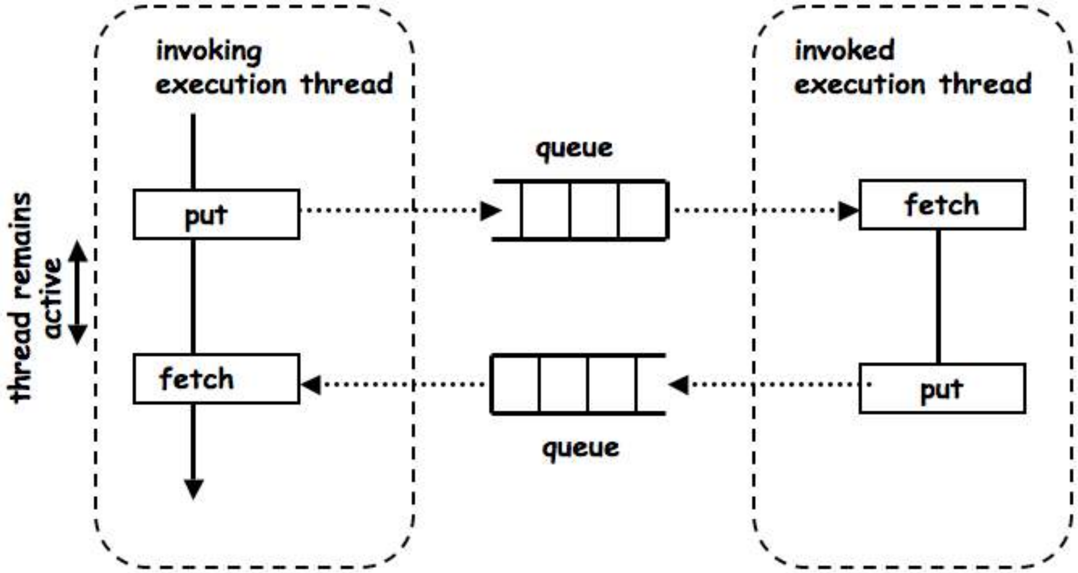
- Traditionally used for developing distributed systems:
  - Client waits while server processes a request (blocking call)
  - Requires both parties to be on-line
- **Advantage**
  - Simple to understand and implement
  - Failures are simple to manage
- **Disadvantages**
  - Connection overhead
  - Higher probability of failures
  - Solutions:
    - → Transactions
    - → Asynchronous interactions



Asynchronous Interaction (i)

- Calls to servers are non-blocking thus clients can continue running:
  - Clients checks at different times to see if a response is ready
  - Typically implemented via message queues
- **Disadvantage:**
  - Adds complexity to client architecture
- **Advantages:**
  - More modular
  - More distribution modes (multicast, replication, message coalescing, etc.)
  - More natural way to implement complex interactions between heterogeneous systems

Asynchronous Interaction (ii)



Client-Server Characteristics

- ✓ State Management
  - ✓ Server-side: persistent or not
  - ✓ Client-side: stateful or stateless
- ✓ Communication Model
  - ✓ Connected or disconnected mode (datagrams)
  - ✓ Synchronous or asynchronous
- Server-side Execution Model
  - □ One or more processes
  - □ Pool of processes or processes on-demand

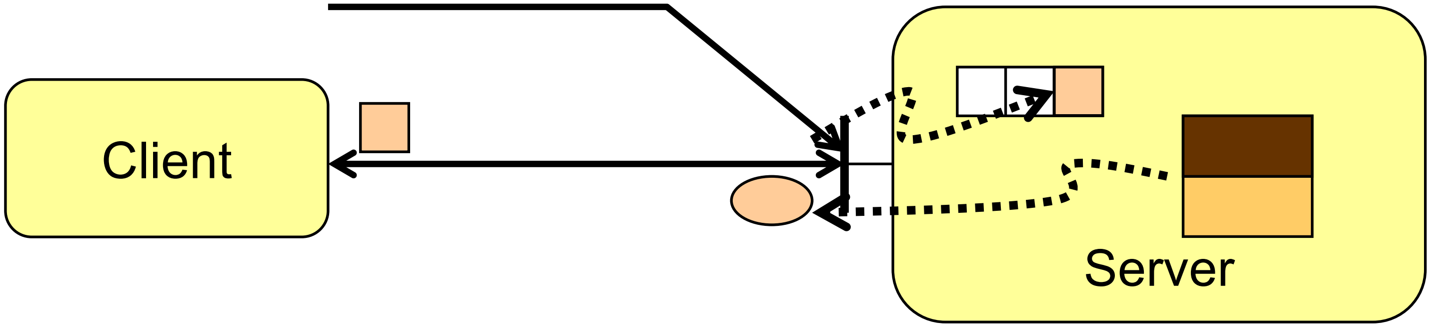
Execution models

- **Iterative execution**:
  - Based on a single process

- **Concurrent execution:**
  - Based on multiple processes or threads
    - Processes are created on-demand
    - Processes are selected from a "pool of processes"

Single Process Execution

```
while (true) {
   receive(client_id, message);
   extract(message, service_id, params);
   result = do_service[service_id](params);
   send(client_id, result);
}
```



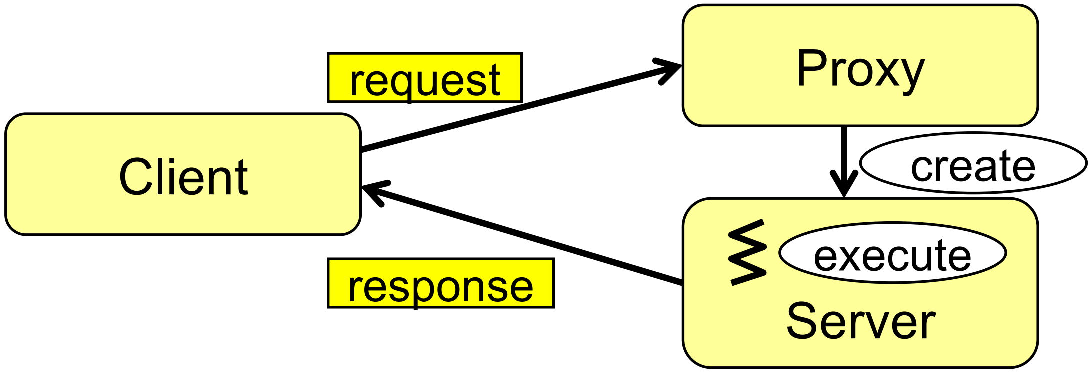
Processes Created On-Demand

Proxy

```
while (true) {
  receive(client_id, message);
  extract(message, service_id, params);
  create_process(client_id, service_id, params);
}
```

Server

```
// código a ejecutar
result = do_service[service_id](params);
send(client_id, result);
exit;
```



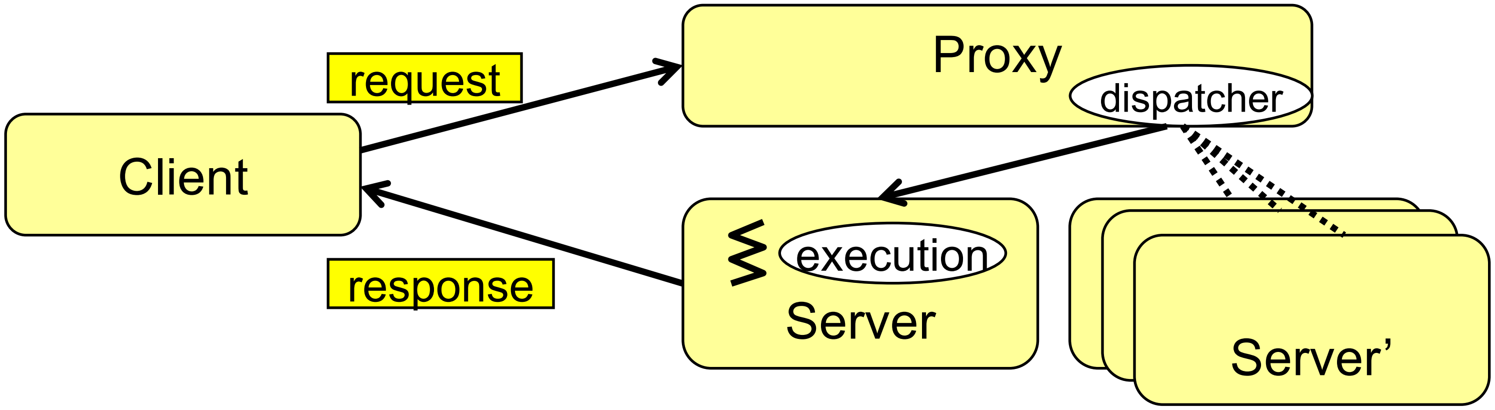
Pool of Processes

Proxy

```
while (true) {
  receive(client_id, message);
  extract(message, service_id, params);
  dispatch(client_id, service_id, params);
}
```

Servicio

```
// código a ejecutar
result = do_service[service_id](params);
send(client_id, result);
exit;
```


request
Proxy
dispatcher
Client
execution
Server
Server'
response