REST & RESTful WEB SERVICES

In a Nutshell

- REST is about resources and how to represent resources in different ways.
- REST is about client-server communication.
- REST is about how to manipulate resources.
- REST offers a simple, interoperable and flexible way of writing web services that can be very different from other techniques.
- Comes from Roy Fielding's Thesis study.

REST is NOT!

- A protocol.
- A standard.
- A replacement for SOAP.

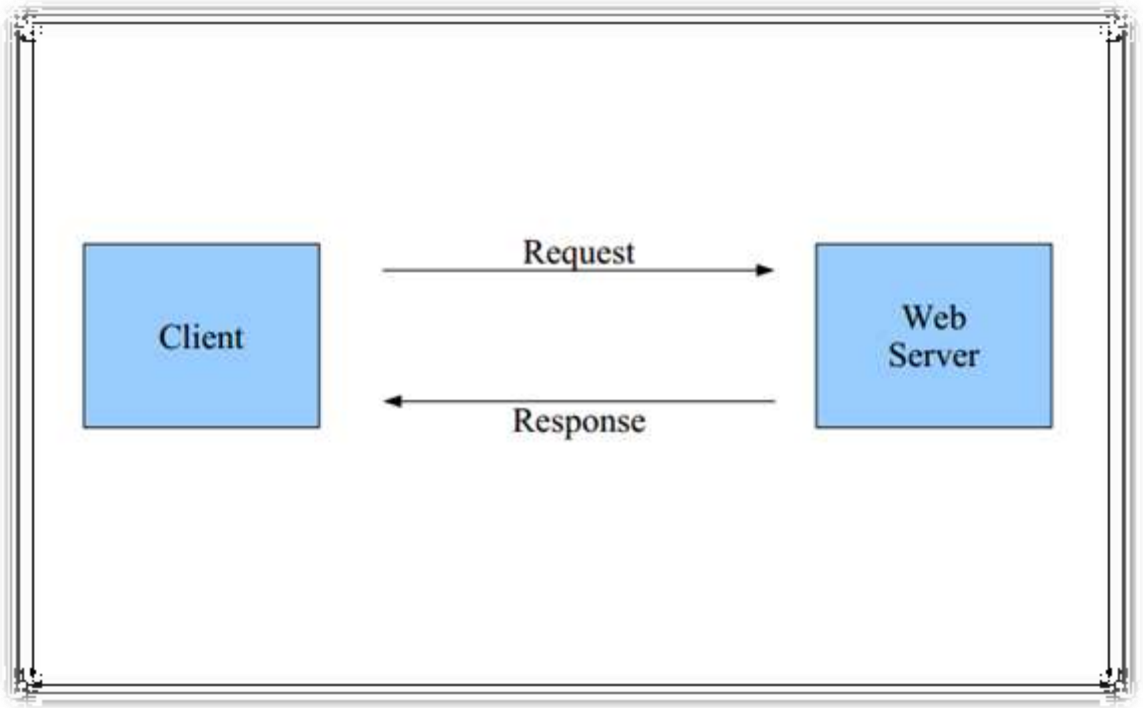
REST

- **Re**presentational **S**tate **T**ransfer
- Architectural style (technically not a standard)
- Idea: a network of web pages where the client progresses through an application by selecting links
- When client traverses link, accesses new resource (i.e., transfers state)
- Uses existing standards, e.g., HTTP
- REST is an architecture all about the Client-Server communication.

An Architectural Style

- REST is the architecture of the Web as it works today and, so it is already used in the **web**!
- It is an software architectural model which is used to describe distributed systems like **WWW** (**W**orld **W**ide **W**eb).
- It has been developed in parallel with **HTTP** protocol.

THE WEB



REST

- Client **request**s a specific **resource** from the server.
- The server **respond**s to that request by delivering the requested resource.
- Server does not have any information about any client.
- So, there is no difference between the two requests of the same client.
- A model which the representations of the resources are transferred between the client and the server.
- The Web as we know is already in this form!

Resources

- Resources are just consistent mappings from an identifier [such as a URL path] to some set of views on server-side state.
- Every resource must be uniquely addressable via a URI.
- “If one view doesn’t suit your needs, then feel free to create a different resource that provides a better view. ”
- “These views need not have anything to do with how the information is stored on the server … They just need to be understandable (and actionable) by the recipient.”
  Roy T. Fielding

Requests & Responses

- REQUEST

  GET /news/ HTTP/1.1

  Host: example.org

  Accept-Encoding: compress, gzip

  User-Agent: Python-httplib2

Here is a **GET** request to «http://example.org/news/»

Method = **GET**

Requests & Responses

- And here is the response…
- RESPONSE

HTTP/1.1 200 Ok

Date: Thu, 07 Aug 2008 15:06:24 GMT

Server: Apache

ETag: "85a1b765e8c01dbf872651d7a5"

Content-Type: text/html

Cache-Control: max-age=3600

<!DOCTYPE HTML>

…

Requests & Responses

- The request is to a resource identified by a **URI** (**URI** = **U**nified **R**esource **I**dentifier).
- In this case, the resource is «http://example.org/news/»
- Resources, or addressability is very important.
- Every resource is URL-addressable.
- To change system state, simply change a resource.

URI Examples

- http://localhost:9999/restapi/books
  - GET – get all books
  - POST – add a new book
- http://localhost:9999/restapi/books/{id}
  - GET – get the book whose id is provided
  - POST – update the book whose id is provided
  - DELETE – delete the book whose id is provided

REST Characteristics

- Resources: Application state and functionality are abstracted into resources.
  - *URI*: Every resource is **uniquely addressable** using URIs.
  - *Uniform Interface*: All resources share a uniform interface for the transfer of state between client and resource, consisting of
    - Methods: Use only HTTP methods such as **GET**, **PUT**, **POST**, **DELETE**, **HEAD**
    - Representation
- Protocol (The constraints and the principles)
  - *Client-Server*
  - *Stateless*
  - *Cacheable*
  - *Layered*

HTTP Methods

- GET – *safe*, *idempotent*, *cacheable*
- PUT - *idempotent*
- POST
- DELETE - *idempotent*
- HEAD
- OPTIONS

CRUD Operations Mapped to HTTP Methods in RESTful Web Services

OPERATION	HTTP METHOD
Create	POST
Read	GET
Update	PUT or POST
Delete	DELETE

Status Codes

HTTP status codes returned in the response header:

- **200 OK** The resource was read, updated, or deleted.
- **201 Created** The resource was created.
- **400 Bad Request** The data sent in the request was bad.
- **403 Not Authorized** The Principal named in the request was not authorized to perform this action.
- **404 Not Found** The resource does not exist.
- **409 Conflict** A duplicate resource could not be created.
- **500 Internal Server Error** A service error occurred.

RESTful Web Services

- RESTful web services are web services which are REST based.
- **Stateless** & **cacheable**.
- Uses **URI** & **HTTP** methods.
- Frequently used with **SOA** projects.
- Quiet light, extensible and simple services.
- The reason behind the **popularity of REST** is that the applications we use are **browser-based** nowadays and top it all, REST is built on **HTTP**.
- Main idea: Providing the communication between **client** and **server** over **HTTP** protocol rather than other complex architectures like SOAP and RPC etc.

RESTful Web Services

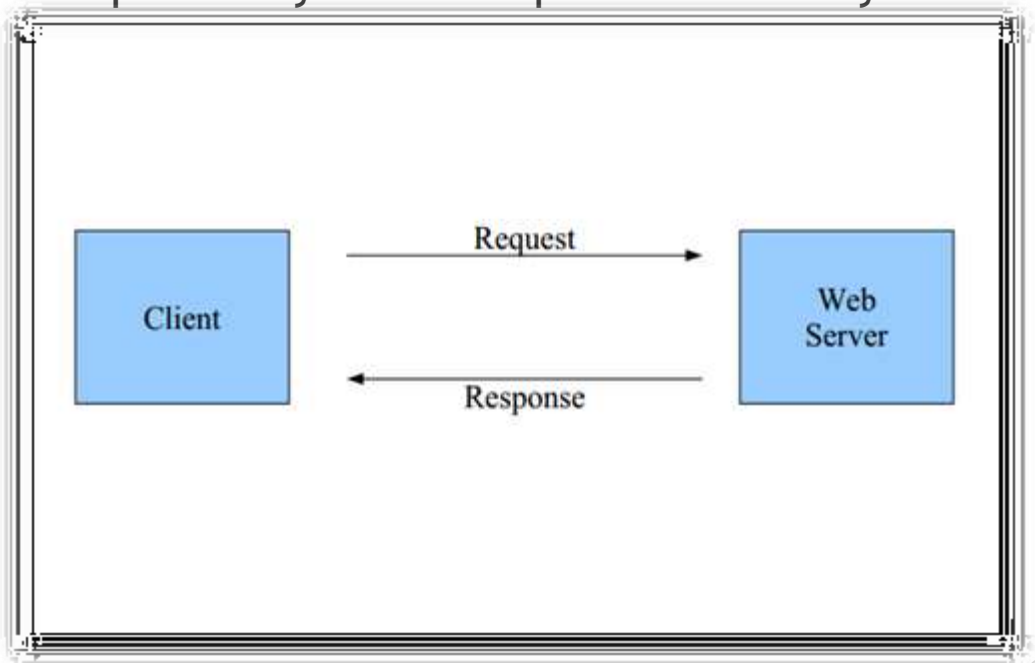
- You can do anything you already do with normal web services.
- No severe restrictions on how the architectural model will be and what properties it will have.
- Models like SOAP have severe rules, REST does not.
- There are lots of frameworks to develop RESTful web services on platforms like C# and Java, but you can write one easily using some standard libraries.
- Inspite of the low bandwidth, large data have been transfered with methods even inflating the size of the data, like XML with SOAP. Why?
- Nowadays, the bandwidth is amazingly large, but we still use JSON and it shrinks the size of our data.

RESTful Web Services

- Platform independent.
- Language independent.
- Work on HTTP protocol.
- Flexible and easily extendible.
- They also have some constraints or principles.
  - *Client-Server*
  - *Stateless*
  - *Cacheable*
  - *Uniform Interface*
  - *Layered System*
  - *Code on Demand*

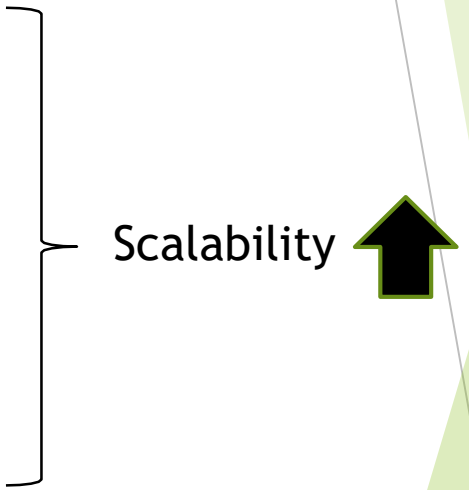
Client-Server

- **Seperation of concerns.**
- Client and server are independent from eachother.
- Client doesn’t know anything about the resource which is kept in the server.
- Server responds as long as the right requests come in.
- **Goal**: Platform independency and to improve scalability.



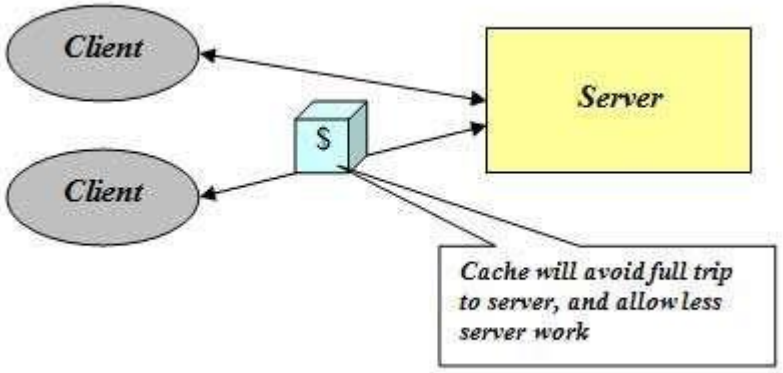
Stateless

- Each request is independent from other requests.
- No client session data or any context stored on the server.
- Every request from client stores the required information, so that the server can respond.
- If there are needs for session-specific data, it should be held and maintained by the client and transferred to the server with each request as needed.
- A service layer which doesn't have to maintain client sessions is much easier to scale.
- Of course there may be cumbersome situations:
  - The client must load the required information to every request. And this increases the network traffic.
  - Server might be loaded with heavy work of «validation» of requests.



Cacheable

- HTTP responses must be cacheable by the clients.
- Important for performance.
- If a new request for the resources comes within a while, then the cached response will be returned.



Uniform Interface

- All resources are accessed with a generic interface (HTTP-based).
- This makes it easier to manage the communication.
- By the help of a uniform interface, client and server evolve independently from eachother.
- E.g. LEGO; eventhough there are different shaped pieces, there are only a few ways to pair up these pieces.

Layered System

- There can be many intermediaries between you and the server you are connecting to.
- Actually, a client does not know if it is connected to the last server or an intermediary server.
- Intermediaries may improve performance by caching and message passing.
- Intermediary servers can increase scalability by load-balancing and can force clients to form some sort of security policies.
- This structure can be used when encapsulation is needed.

Code on Demand

- Servers can send some kind of **executable scripts** to the client-side in order to increase or change the functionality on the client side.
- This may cause **low visibility**, so it is the only **optional** constraint.
- *EXAMPLE*

```
...
<head>
    <script src="utility.js"
        type="text/javascript">
    </script>
...
```

more

- If a service does not include all constraints out of «Code on Demand», it is not a RESTful web service.
- The most epic constraint is «**Stateless**».

What REST actually aims for?

- Scalability
- Simplicity
- Modifiability
- Useability
- Portability
- Reliability

Benefits

Network Performance

- Efficiency
- Scalability
- User Perceived Performance

Other Benefits

- Simplicity
- Evolvability
- Reuseability
- Visibility
- Extensibility
- Configuration
- Customizability

Benefits ctd.

- HTTP is **efficient** because of all those **caches**, your request may not have to reach all the way back to the origin server.
- **Scalability** comes from many areas. The use of layers allows you to distribute traffic among a large set of origin servers based on method, URI or content-type, or any other visible control data or meta-data in the request headers.
- Caching also helps **scalability** as it reduces the actual number of requests that hit the origin server.
- Statelessness allows requests to be routed through different gateways and proxies, thus avoiding introducing bottlenecks, allowing more intermediaries to be added as needed. [**Scalability**]

Will REST Replace Other Technologies ?

- There is actually an untold story that both technologies can be mixed and matched. REST is very easy to understand and is extremely approachable, but does **lack standards** and is considered an **architectural approach**. In comparison, SOAP is an industry standard with a **well-defined protocol** and a set of **well-established rules** to be implemented, and it has been used in systems both big and small.

REST vs SOAP

- SOAP is a XML-based message protocol, while REST is an architectural style.
- SOAP uses WSDL for communication between consumer and provider, whereas REST just uses XML or JSON to send and receive data.
- SOAP invokes services by calling RPC method, REST just simply calls services via URL path.
- SOAP doesn't return human readable result, whilst REST result is readable with is just plain XML or JSON.
- SOAP is not just over HTTP, it also uses other protocols such as SMTP, FTP, etc, REST is over only HTTP.

SOAP uses WSDL for communication between consumer and provider, whereas REST just uses XML or JSON to send and receive data.

- WSDL defines contract between client and service and is static by its nature. In case of REST contract is somewhat complicated and is defined by HTTP, URI, Media Formats and Application Specific Coordination Protocol. It's highly dynamic unlike WSDL.

Performance is broad topic!-1

- If you mean the load of the server, REST has a bit better performance because it bears minimal overhead on top of HTTP. Usually SOAP brings with it a stack of different (generated) handlers and parsers. Anyway, the performance difference itself is not that big, but RESTful service is more easy to scale up since you don't have any server side sessions.
- If you mean the performance of the network (i.e. bandwidth), REST has much better performance. Basically, it's just HTTP. No overhead. So, if your service runs on top of HTTP anyway, you can't get much leaner than REST. Furthermore if you encode your representations in JSON (as opposed to XML), you'll save many more bytes.
- In short, I would say 'yes', you'll be more performant with REST. Also, it (in my opinion) will make your interface easier to consume for your clients. So, not only your server becomes leaner but the client too.

Performance is broad topic!-2 *However, couple of things to consider!

- RESTful interfaces tend to be a bit more "chatty", so depending on your domain and how you design your resources, you may end up doing more HTTP requests.
- SOAP has a very wide tool support. For example, consultants love it because they can use tools to define the interface and generate the WSDL file and developers love it because they can use another set of tools to generate all the networking code from that WSDL file. Moreover, XML as representation has schemas and validators, which in some cases may be a key issue. (JSON and REST do have similar stuff coming but the tool support is far behind)

Complements or Competitors ?

- There is some competition between proponents of each approach.
- Yet both have value.
  - The challenge is to determine when to use each one !

So the areas that REST works really well are:

- **Limited bandwidth and resources;** remember the return structure is really in any format (developer defined). Plus, any browser can be used because the REST approach uses the standard *GET*, *PUT*, *POST*, and *DELETE* verbs. Again, remember that REST can also use the *XMLHttpRequest* object that most modern browsers support today, which adds an extra bonus of AJAX.
- **Totally stateless operations;** if an operation needs to be continued, then REST is not the best approach and SOAP may fit it better. However, if you need stateless CRUD (Create, Read, Update, and Delete) operations, then REST is it.
- **Caching situations;** if the information can be cached because of the totally stateless operation of the REST approach, this is perfect.

So if you have the following then SOAP is a great solution:

- **Asynchronous processing and invocation;** if your application needs a guaranteed level of reliability and security then SOAP 1.2 offers additional standards to ensure this type of operation. Things like WSRM – WS-Reliable Messaging.
- **Formal contracts;** if both sides (provider and consumer) have to agree on the exchange format then SOAP 1.2 gives the rigid specifications for this type of interaction.
- **Stateful operations**; if the application needs contextual information and conversational state management then SOAP 1.2 has the additional specification in the WS* structure to support those things (Security, Transactions, Coordination, etc). Comparatively, the REST approach would make the developers build this custom plumbing.