# RESTFUL WEB SERVICES

## INTRODUCTION

### WHAT IS REST?

REST: Representational State Transfer.

REST is an architectural style which is based on web-standards and the HTTP protocol.

In a REST based architecture everything is a resource.
A resource is accessed via a common interface based on the HTTP standard methods.

In a REST based architecture you have a REST server which provides access to the resources.

A REST client can access and modify the REST resources.
Every resource should support the HTTP common operations. Resources are identified by global IDs (which are typically URIs).

REST allows that resources have different representations, e.g., text, XML, JSON etc. The REST client can ask for a specific representation via the HTTP protocol (content negotiation).

### RESTFUL WEB SERVICES

RESTFul web services are based on HTTP methods and the concept of REST.
A RESTFul web service defines the base URI for the services, the supported MIME-types (XML, text, JSON, user-defined, …).
It also defines the set of operations (POST, GET, PUT, DELETE) which are supported.

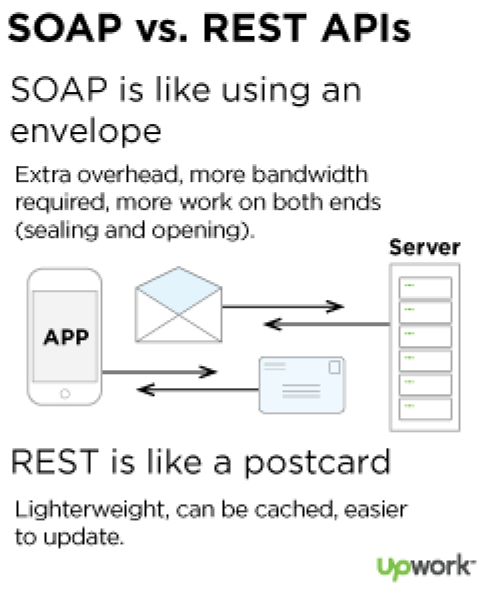
### IMPORTANCE OF REST

If measured by the number of Web services that use it, REST has emerged in the last few years alone as a predominant Web service design model.

In fact, REST has had such a large impact on the Web that it has mostly displaced SOAP-and WSDL-based interface design because it's a considerably simpler style to use.

## SOAP VS REST

- SOAP is a protocol. REST is an architectural style.
  An API is designed to expose certain aspects of an application's business logic on a server, and SOAP uses a service interface to do this while REST uses URIs.
- REST APIs access a resource for data (a URI); SOAP APIs perform an operation.
  REST is an architecture that's more data-driven; SOAP is a standardized protocol for transferring structured information that's more function-driven.
- REST permits many different data formats including plain text, HTML, XML, and JSON, which is a great fit for data and yields more browser compatibility; SOAP only uses XML.
- Security is handled differently. SOAP supports WS-Security, which is great at the transport level and a bit more comprehensive than SSL, and more ideal for integration with enterprise-level security tools. Both support SSL for end-to-end security, and REST can use the secure version of the HTTP protocol, HTTPS.
- SOAP requires more bandwidth; REST requires fewer resources (depending on the API). There's a little more overhead with SOAP out of the gate, on account of the envelope-style of payload transport. Because REST is used primarily for web services, its being lightweight is an advantage in those scenarios.
- REST calls can be cached, SOAP-based calls cannot be cached. Data can be marked as cacheable, which means it can be reused by the browser later without having to initiate another request back to the server. This saves time and resources.
- An API is built to handle your app's payload, and REST and SOAP do this differently. A payload is data sent over the internet, and when a payload is "heavy," it requires more resources. REST tends to use HTTP and JSON, which lighten the payload; SOAP relies more on XML.
- SOAP is tightly coupled with the server; REST is coupled to a lesser degree. In programming, the more layers of abstraction between two pieces of technology, the less control you have over their interaction, but there's also less complexity and it's easier to make updates to one or the other without blowing up the whole relationship. The same goes for APIs and how closely they interact with a server. This is a key difference between SOAP and REST to consider. SOAP is very closely coupled with the server, having a strict communication contract with it that makes it more difficult to make changes or updates. A client interacting with a REST API needs no knowledge of the API, but a client interacting with a SOAP API needs knowledge about everything it will be using before it can even initiate an interaction.


SOAP vs. REST APIs
SOAP is like using an envelope
Extra overhead, more bandwidth required, more work on both ends (sealing and opening).
Server
APP
REST is like a postcard
Lighterweight, can be cached, easier to update.
upwork

## HTTP METHODS

The PUT, GET, POST and DELETE methods are typical used in REST based architectures.
The following text gives an explanation of these operations.

GET defines a reading access of the resource without side-effects.
The resource is never changed via a GET request, e.g., the request has no side effects (idempotent).

PUT creates a new resource.
It must also be idempotent.

DELETE removes the resources.
The operations are idempotent.
They can get repeated without leading to different results.

POST updates an existing resource or creates a new resource.

## JAX-RS WITH JERSEY

### JAX-RS

Java defines REST support via the Java Specification Request (JSR) 311. This specification is called JAX-RS (The Java API for RESTful Web Services). JAX-RS uses annotations to define the REST relevance of Java classes.

### JERSEY

The Jersey implementation provides a library to implement Restful webservices in a Java servlet container. It is the reference implementation for the JSR 311 specification.

Jersey provides a servlet implementation which scans predefined classes to identify RESTful resources. You use annotations in the class and methods to define their responsibility.

The Jersey implementation also provides a client library to communicate with a RESTful webservice.

Retrofit is a very powerful REST client.
The base URL of this servlet is: http://your_domain:port/context-root/url-pattern/path_from_rest_class
This servlet analyzes the incoming HTTP request. It selects the correct class and method to respond to this request.

These two types are typically maintained in different packages.
JAX-RS supports the creation of XML and JSON via the Java Architecture for XML Binding (JAXB).

### WEB CONTAINER

For implementing RESTful web services, can use any web container, for example Tomcat or the Google App Engine.

If you want to use Tomcat as servlet container please see Eclipse WTP and Apache Tomcat for instructions on how to install and use Eclipse WTP and Apache Tomcat.

Alternative you could also use the Google App Engine for running the server part of the following REST examples.
If you use the Google App Engine, you do not have to install and configure Tomcat.

If you are using GAE/J, you have to create App Engine projects instead of Dynamic Web Project.
The following description is based on Apache Tomcat.

## PRINCIPLES/CONSTRAINTS APPLIED TO REST ARCITECTURE

REST-style architectures consist of clients and servers. Clients initiate requests to servers who process these requests and return responses based on these requests. These requests and responses are built around the transfer of representations of these resources. A resource can be any coherent and meaningful concept that can be addressed, while a representation of a resource is a document that captures the intended state of a resource. Fundamentally in REST each resource is first identified using a URL

REST architectural style describes six constraints applied to architecture:

1. Uniform Interface
Individual resources are identified using URLS. The resources (database) are themselves different from the representation (XML, JSON, HTML) sent to the client. The client can manipulate the resource through the representations provided they have the permissions. Each message sent between the client and the server is self-descriptive and includes enough information to describe how it is to be processed. The hypermedia comprises of hyperlinks and hypertext. These act as the engine for state transfer.

2. Stateless Interactions
none of the clients context is to be stored on the server side between the request. All of the information necessary to service the request is contained in the URL, query parameters, body or headers.

3. Cacheable
Clients can cache the responses. The responses must define themselves as cacheable or not to prevent the client from sending the inappropriate data in response to further requests.

4. Client-Server
The clients and the server are separated from each other thus the client is not concerned with the data storage thus the portability of the client code is improved while on the server side the server is not concerned with the client interference thus the server is simpler and easy to scale.

5. Layered System
At any time client cannot tell if it is connected to the end server or to an intermediate. The intermediate layer helps to enforce the security policies and improve the system scalability by enabling load-balancing

6. Code on Demand
an optional constraint where the server temporarily extends the functionality of a client by the transfer of executable code.