## SOA(Service Oriented Architecture) Core ideas:

1. A set of services that a business wants to provide to their

customers, partners, or other areas of an organization

1. An architectural pattern that requires a service provider,

mediation, and service requestor with a service description

1. A set of architectural principles, patterns and criteria that address

characteristics such as modularity, encapsulation, loose coupling,s

eparation of concerns, reuse and composability

1. A programming model complete with standards, tools and

technologies that supports web services, ReST services or other

kinds of services

1. A middleware solution optimized for service assembly,

orchestration, monitoring, and management

## SOA principle :

1. Standardized service contract: Services adhere to a communications

agreement, as defined collectively by one or more service-description

documents.

1. Service loose coupling: Services maintain a relationship that minimizes

dependencies and only requires that they maintain an awareness of each other.

1. Service abstraction: Beyond descriptions in the service contract, services hide

logic from the outside world.

1. Service reusability: Logic is divided into services with the intention of

promoting reuse.

1. Service autonomy: Services have control over the logic they encapsulate.
2. Service statelessness: Services minimize resource consumption by deferring

the management of state information when necessary.

1. Service discoverability: Services are supplemented with communicative meta

data by which they can be effectively discovered and interpreted.

1. Service composability: Services are effective composition participants,

regardless of the size and complexity of the composition.

1. Service granularity: A design consideration to provide optimal

scope at the right granular level of the business functionality in a service operation.

1. Service normalization: Services are decomposed and/or

consolidated to a level of normal form to minimize redundancy.

In some cases, services are denormalized for specific

purposes, such as performance optimization, access, and aggregation.

1. Service location transparency: The ability of a service

consumer to invoke a service regardless of its actual location in the network.

## SOAP vs REST and other WS flavours (geospatial)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Protocol Support | Architectural  Design over http call | Use pattern | HTTP method |
| SOAP | HTTP and other protocols | Upon the paradigm of the remote procedure call | a stack of protocols that  covers every aspect of using a remote service | POST/GET |
|  |  |  |  |  |
| REST | HTTP only | Centered about resources, and its manipulated way | more of a style of using HTTP than a separate  protocol, | GET/POST/PUT/DELETE |

## Other WS flavours(geospatial)

Geo-spatial data:

1. Finished maps (as the ones you see on Google Maps): WMS,WMTS,SLD
2. Vector data(just the geometry, in practice a collection of points): WFS,FE
3. Raster data(a mathematical matrix): WCS
4. Metadata(information about the geo-spatial data available): CSW
5. Processing(remote execution/computation on data):WPS

WFS data source interactions, entails following steps:

1. Retrieving the list of feature types available from that service

2. Retrieving information (metadata) about a feature type

3. Retrieving the actual data

## Resource-oriented architectures and do’s don’ts

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

Procedure:

1. Figure out the data set

2. Split the data set into resources and for each kind of resource:

3. Name the resources with URIs

4. Expose a subset of the uniform interface

5. Design the representation(s) accepted from the client

6. Design the representation(s) served to the client

7. Integrate this resource into existing resources, using hypermedia links and forms

8. Consider the typical course of events: what’s supposed to happen?

9. Consider error conditions: what might go wrong?

Rest Best Practice

1) Keep your 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". That is, the response

itself contains info about "what's the next step to take". Contrast this to responses that do not contain links. Thus, the decision of "what's the next step to take" must be made out-of-band.

6) Minimize the use of query strings.

7) Use HTTP status codes to convey errors/success

8) In general, keep the REST principles in mind. In particular:

• Addressability

• Uniform Interface

• Resources and Representations instead of RPC

• HATEOAS

## ACTION HTTP METHOD

|  |  |
| --- | --- |
| Create Resource | PUT to a new URI  POST to an existing URI |
| Retrieve Resource | GET |
| Update Resource | POST to an existing URI |
| Delete | Resource DELETE |

## Code versioning and Git demo



Version control system

* Local (RCS)
* Centrialiszed(SVN,CZS)
* Decentrailized(Git,Mercruial,Bitbucket)