# Chapter 5: SOA and RESTFUL Web services

Service-oriented architecture (SOA) and development is a paradigm where software components are created with concise interfaces, and each component performs a discrete set of related functions. With its well-defined interface and contract for usage, each component, provides a service to other software components. This is analogous to an accountant who provides a service to a business, even though that service consists of many related functions—bookkeeping, tax filing, investment management, and so on.

There are no technology requirements or restrictions with SOA. You can build a service in any language with standards such as CORBA, remote procedure calls (RPC), or XML. Although SOA has been around as a concept for years, its vague definition makes it difficult to identify. The client/server development model of the early '90s was a simple example of an SOA-based approach to software development.

A web service is an example of an SOA with a well-defined set of implementation choices. In general, the technology choices are SOAP and the Web Service Definition Language (WSDL); both XML-based. WSDL describes the interface (the "contract"), while SOAP describes the data that is transferred. Because of the platform-neutral nature of XML, SOAP, and WSDL, Java is a popular choice for web-service implementation due to its OS-neutrality.

Web-service systems are an improvement of client/server systems, and proprietary object models such as CORBA or COM, because they're standardized and free of many platform constraints. Additionally, the standards, languages, and protocols typically used to implement web services helps systems built around them to scale better.

## Representational State Transfer (REST)

However, there exists an even less restrictive form of SOA than a web service—representational state transfer (REST). Described by Roy Fielding in his doctoral dissertation, REST is a collection of principals that are technology independent, except for the requirement that it be based on HTTP.

A system that conforms to the following set of principals is said to be "RESTful":

- All components of the system communicate through interfaces with clearly defined methods and dynamic code.
- Each component is uniquely identified through a hypermedia link (URL).
- A client/server architecture is followed (web browser and web server).
- All communication is stateless.
- The architecture is tiered, and data can be cached at any layer.

These principals map directly to those used in the development of the Web, and according to Fielding, account for much of the Web's success. HTTP protocol, its interface of methods (GET, POST, HEAD, and so on), the use of URLs, HTML, and JavaScript, as well as the clear distinction between what is a web server and web browser, all map directly to the first four principals. The final principal (involving tiers) allows for the common network technology found in most website implementations: load balancers, in-memory caches, firewalls, routers, and so on. These devices are acceptable because they don't affect the interfaces between the components; they merely enhance their performance and communication.

The Web is the premier example of a RESTful system, which makes sense since much of the Web's architecture preceded the definition of REST. What the Web makes clear, however, is that complex remote procedure call protocols are not needed to create successful, scalable, understandable, and reliable distributed software systems. Instead, the principals of REST are all you need.

Overall, REST can be described as a technology and platform-independent architecture where loosely coupled components communicate via interfaces over standard web protocols. Software, hardware, and data-centric designs maximize system efficiency, scalability, and network throughput. The underlying principal is simplicity.

Figure 1 illustrates the REST architecture, combining both logical software architecture and physical network elements. Communication is performed over HTTP, clients contain optional server caches for efficiency, services employ caches to backend databases, there are no restrictions on maximum clients, or maximum services per client, services can call services, load-balancing hardware is used for scalability, and firewalls can be used for security.

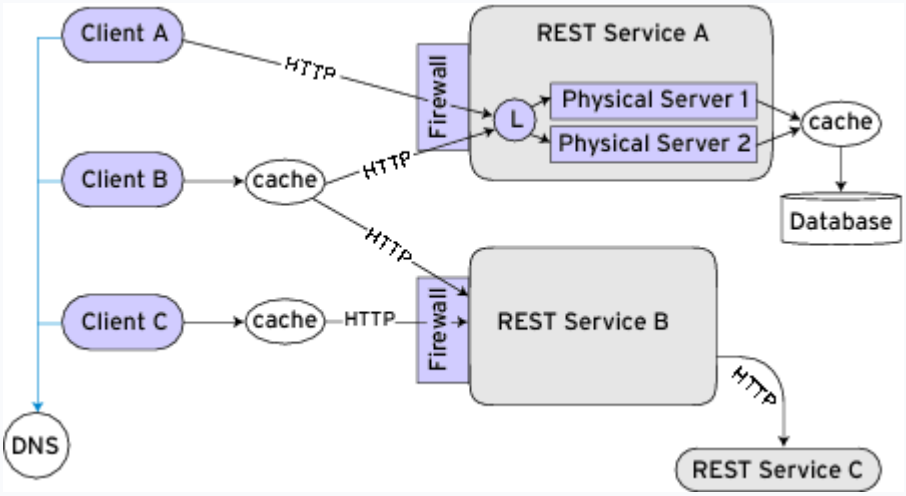


**Figure 1: This architectural diagram provides a visual overview of the REST principals.**

There are some interesting points on data caching that need to be made. First, data must be marked, either implicitly or explicitly, as cacheable, or noncacheable. Second, although specialized caches may be used (custom, in-memory data structures), general-purpose caches, such as web browser caches or third-party web caches (such as Akamai), can also be used.

## Resource Oriented Architectures

### Main Concepts

Resource Oriented Architectures (ROA) are based upon the concept of resource; each resource is a directly accessible distributed component that is handled through a standard, common interface making possible resources handling. RESTFul platforms based on REST development technology enable the creation of ROA. The main ROA concepts are the following:

- Resource – anything that's important enough to be referenced as a thing itself
- Resource name – unique identification of the resource

• Resource representation – useful information about the current state of a resource
• Resource link – link to another representation of the same or another resource
• Resource interface – uniform interface for accessing the resource and manipulating its state .
The resource interface semantics is based on the one of HTTP operations. The following table summarizes the resource methods and how they could be implemented1 using the HTTP protocol:

| ***Resource method*** | ***Description*** | ***HTTP operation*** |
|---|---|---|
| createResource | Create a new resource (and the corresponding unique identifier) | PUT |
| getResourceRepresentation | Retrieve the representation of the resource | GET |
| deleteResource | Delete the resource (optionally including linked resources) | DELETE (referred resource only), POST (can be used if the delete is including linked resources) |
| modifyResource | Modify the resource | POST |
| getMetaInformation | Obtain meta information about the resource | HEAD |

*Table 1*

In terms of platform implementation specification, each resource must be associated to a unique identifier that usually consists of the URL exhibiting the resource interface.

***Designing Resource Oriented Architectures***

One of the most important decisions that have to be taken in the design of a Resource Oriented Architecture is what must be considered a resource (by definition, each component deserving to be directly represented and accessed). This is one of the main differences between ROA and SOA where in the latter one the single, directly accessible distributed component, represents one or more business functionalities that often process different potential resources. Such resources are credited candidates for being considered resources in a ROA, thus deserving to be represented as distributed components (e.g. features offered by WFS, registers or items of registries/catalogues, WFS and Registry/Catalogue services functionalities).

Once having defined the granularity of the resources composing the ROA it is necessary to define, for each resource type, the content of the messages for invoking the methods as well as the corresponding responses. More in detail, beside the definition of resource types (and sub types), an addressing schema for accessing instances of those resource types, a response schema (response is not binary) and a mapping of logical functions to the HTTP operations. All resources in a ROA are accessed via the same common interface which is plain HTTP. It is worth

noting that the usage of a common interface does not necessarily mean that all the necessary information enabling interoperability and collaboration among resources are available. The necessity of integrating a standard common interface description with some specific service instance aspects has been already addressed in other existing solutions such as, for instance, for the OGC WPS execute operation [7] where the specific processing detailed information are offered through the describe Process operation. In a ROA, this information completes the description of the resources and their interface thus enabling: i) system integration and interoperability through tools for the creation of resource clients, ii) message validation processes, and iii) model driven development (top-down approach) for which a typed description of the interface and its operation is necessary. In the REST technology such a description is based on WADL documents that play the same role of WSDL in the W3C Web Services platform; both languages use XML schema for expressing the structure of exchanged messages.

## Designing Read-Only Resource-Oriented Services:

Once you have an idea of what you want your program to do. It produces a set of resources that respond to a read-only subset of HTTP's uniform interface: GET and possibly HEAD. Once you get to the end of this procedure, you should be ready to implement your resources in whatever language and framework you like. The following steps are to be done to design read-only resource-oriented services:

1. Figure out the data set
2. Split the data set into resources

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?

In the sections to come, I'll show, step by step, how following this procedure results in a RESTful web service that works like the Web.

### Figure Out the Data Set:

A web service starts with a data set, or at least an idea for one. This is the data set you'll be exposing and/or getting your users to build.

### Split the Data Set into Resources:

Once you have a data set in mind, the next step is to decide how to expose the data as HTTP resources. Remember that a resource is *anything interesting enough to be the target of a hypertext link*. Anything that might be refered to by name ought to have a name. Web services commonly expose three kinds of resources:

*Predefined one-off resources for especially important aspects of the application.*
This includes top-level directories of other available resources. Most services expose few or no one-off resources.
*Example:* A web site's homepage. It's a one-of-a-kind resource, at a well-known URI, which acts as a portal to other resources.
*A resource for every object exposed through the service.*
One service may expose many kinds of objects, each with its own resource set. Most services expose a large or infinite number of these resources.
*Resources representing the results of algorithms applied to the data set.*
This includes collection resources, which are usually the results of queries. Most services either expose an infinite number of algorithmic resources, or they don't expose any.
*Example:* A search engine exposes an infinite number of algorithmic resources. There's one for every search request you might possibly make. The Google search engine exposes one resource at *http://google.com/search?q=jellyfish* (that'd be "a directory of resources about jellyfish") and another at *http://google.com/search?q=chocolate* ("a directory of resources about chocolate"). Neither of these resources were explicitly defined ahead of time: Google translates *any* URI of the form *http://google.com/search?q={query}* into an algorithmic resource "a directory of resources about *{query}*."

**Name the Resources:**

Now the resource need names. Resources are named with URIs, so let's pick some. Remember, in a resource-oriented service the URI contains all the scoping information. Our URIs need to answer questions like: "Why should the server operate on this map instead of that map?" and "Why should the server operate on this place instead of that place?"
Now let's consider the resources. The most basic resource is the list of planets. It makes sense to put this at the root URI, *http://maps.example.com/*. Since the list of planets encompasses the entire service, there's no scoping information at all for this resource (unless you count the service version as scoping information).
For the other resources I'd like to pick URIs that organize the scoping information in a natural way. There are three basic rules for URI design, born of collective experience:

1. Use path variables to encode hierarchy: /parent/child
2. Put punctuation characters in path variables to avoid implying hierarchy where none exists: /parent/child1;child2
3. Use query variables to imply inputs into an algorithm, for example: /search?q=jellyfish&start=20

**Map URIs:**

Now that I've designed the URI to a geographic point on a planet, what about the corresponding point on a road map or satellite map? After all, the main point of this service is to serve maps.
Earlier I said I'd expose a resource for every point on a map. For simplicity's sake, I'm not exposing maps of named places, only points of latitude and longitude. In addition to a set of coordinates or the name of a place, I need the name of the planet and the type of map (satellite map, road map, or whatever). Here are some URIs to maps of planets, places, and points:

- *http://maps.example.com/radar/Venus*
- *http://maps.example.com/radar/Venus/65.9,7.00*

- *http://maps.example.com/geologic/Earth/43.9,-103.46*

**Design Your Representations:**

I've decided which resources I'm exposing, and what their URIs will look like. Now I need to decide what data to send when a client requests a resource, and what data format to use.

**Integrate this resource in to existing resources, using hypermedia links and forms(Link the Resources to Each Other):**

Since I designed all my resources in parallel, they're already full of links to each other (see Figure 5-3). A client can get the service's "home page" (the planet list), follow a link to a specific planet, follow another link to a specific map, and then follow navigation and zoom links to jump around the map. A client can do a search for places that meet certain criteria, click one of the search results to find out more about the place, then follow another link to locate the place on a map.

One thing is still missing, though. How is the client supposed to get to a list of search results? I've set up rules for what the URI to a set of search results looks like, but if clients have to follow rules to generate URIs, my service isn't well connected. HTML solves this problem with forms. By sending an appropriate form in a representation, I can tell the client how to plug variables into a query string. The form represents infinitely many URIs, all of which follow a certain pattern. I'm going to extend my representations of places by including this HTML form.

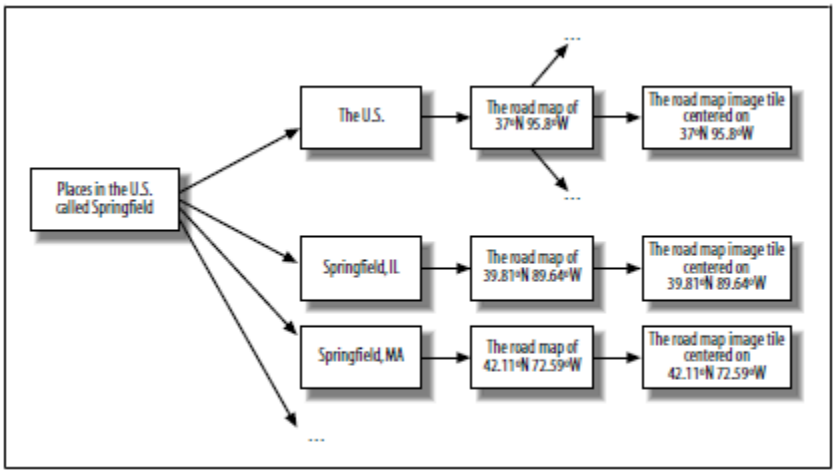


*Figure 5-3. Where can you go from the list of search results?*

**What's Supposed to Happen?**

This step of the design is conceptually simple, but it's the gateway to where you're going to spend much of your implementation time: making sure that client requests are correctly turned into responses. Most read-only resources have a pretty simple typical course of events. The user sends a GET request to a URI, and the server sends back a happy response code like 200 ("OK"), some HTTP headers, and a representation. A HEAD request works the same way, but the server omits the representation. The only main question is which HTTP headers the client should send in the request, and which ones the server should send in the response.

**What Might Go Wrong?**

I also need to plan for requests I can't fulfill. When I hit an error condition I'll send a response code in the 3xx, 4xx, or 5xx range, and I may provide supplementary data in HTTP headers. If

they provide an entity-body, it'll be a document describing an error condition, not a representation of the requested resource (which, after all, couldn't be served).
Here are some likely error conditions for the map application:
• The client may try to access a map that doesn't exist, like /road/Saturn. I understand what the client is asking for, but I don't have the data. The proper response code in this situation is 404 ("Not Found"). I don't need to send an entity-body along with this response code, though it's helpful for debugging.

## Designing Read/Write Resource-Oriented Services:

- Same process, but now we examine full range of uniform interface operations
    - Build matrix with resource types as rows, and operations as columns
        - Indicate what operations apply to which types
        - provide example URIs and discussion of what will happen
            - especially in the case of POST and PUT
                - PUT: create or modify resource
                - POST: append content to existing resource OR append child resource to parent resource (blog entries)
        - Two questions to help
            - Will clients be creating new resources of this type?
            - Who's in charge of determining the new resource's URI? Client or Server? If the former, then PUT. If the latter, then POST.

**New Issues: Authentication and Authorization**

- Now that we are allowing a client to change stuff on our server, we need
    - Authentication: problem of tying a request to a user
    - Authorization: problem of determining which requests to let through for a given user
- HTTP provides mechanisms to enable this (HTTP Basic/Digest) and other web services roll their own (Amazon's public/private key on subset of request)
- Another Issue: Privacy
    - Can't transmit "private information" in the clear; need to use HTTPS
- Another Issue: Trust
    - How do you trust your client software to do the right thing?
        - Especially in today's environment with malware becoming harder and harder to discern