Web Services Architecture

- The simplest Web service system has two participants:
- **Provider:** a service producer.
  - The provider represents the interface and implementation of the service.
- **Requester:** a service consumer.
  - A client application that consumes the provided service.
- Fundamentally, a Web service and the clients that utilize it form a distributed system.
- Web Services rely on the **client-server model**: client applications can access Web services over a network.

Benefits of Web Services

- **Loosely coupled:**
  - Each service exists independently of the other services that make up the application.

- **Ease of integration:**
  - Data is isolated between applications, creating 'silos'.
  - Web Services act as **glue** between applications and enable easier communications within and across organizations.

- **Code reuse:**
  - Take **code reuse** a step further: a specific functionality exposed by a service is only ever coded once and used repeatedly by consuming applications.

Service-Oriented Architecture (SOA)

- **SOA** is an architectural style: term used to describe a style of software design.

- Defines a way to make software components reusable and interoperable via **public interfaces**.
  - Systems comprise several distributed components.
  - Components interact over a network.
  - Components provide discrete **functionalities** (data sharing & computational capabilities), with well-defined interfaces: **services**
  - Loose coupling of components.

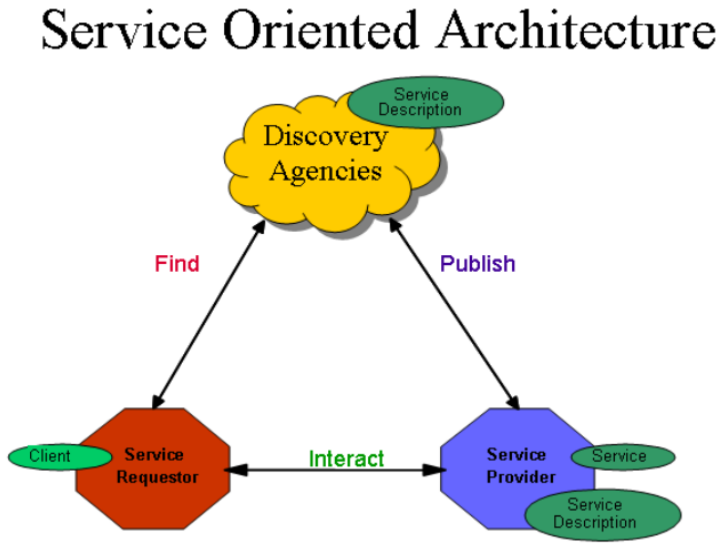
- Popular form (in the early 90s)
  - DCOM, CORBA, RPC, XML-RPC, JSON-RPC, **SOAP**, gRPC (2016), etc.

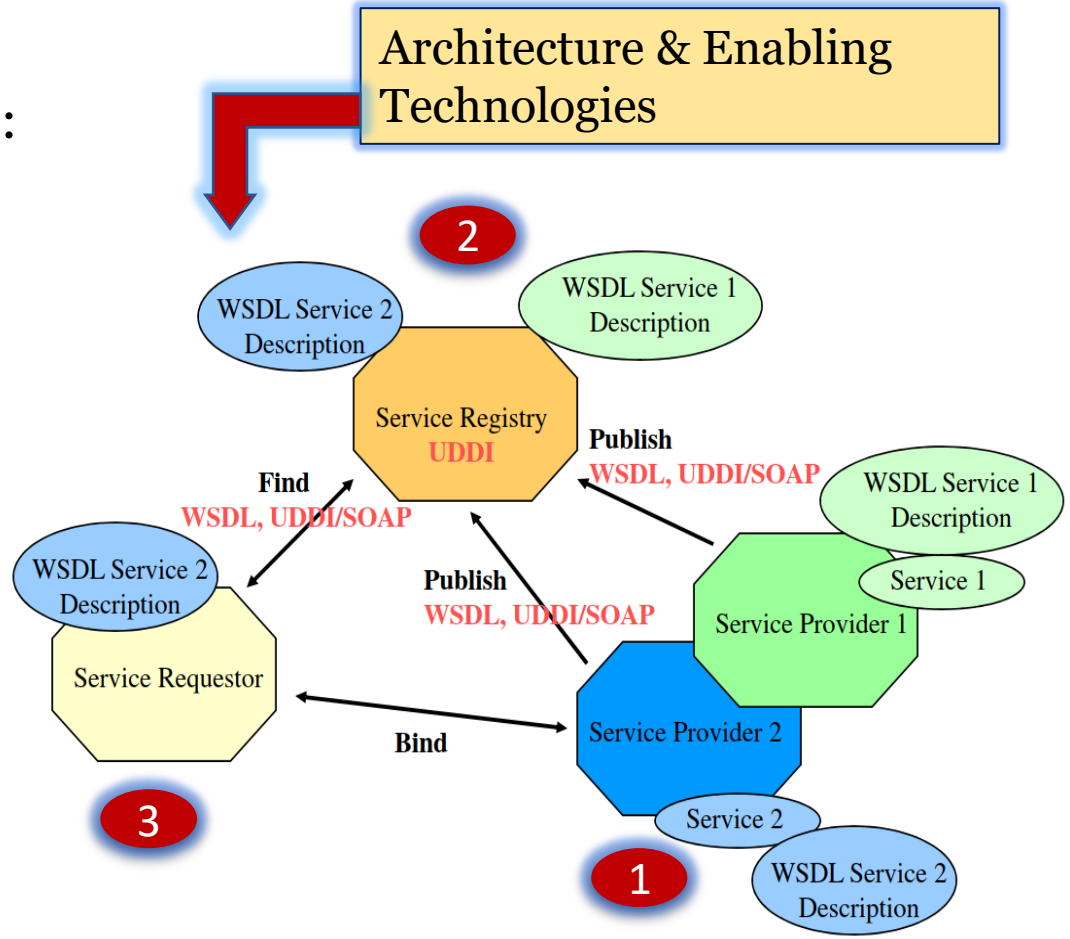
“Big Web Services”

- Big (aka traditional) Web services are based on W3C Web Services Architecture (+WS-I, etc.)
  - They are modular programs that can be discovered and invoked over a network.
  - They rely on a stack of technologies including **XML**, **SOAP** and **WSDL**.
  - Communication between consuming applications and a SOAP-based WS is based on the SOAP protocol.
  - SOAP messages are usually sent across the network using HTTP, although other bindings are possible.

How Big Web Services Work?

- Traditional Web services:
  - **S**imple **O**bject **A**ccess **P**rotocol (**SOAP**): for communication and invoking services.
  - **W**eb **S**ervices **D**escription **L**anguage (**WSDL**): for describing services.
  - **U**niversal **D**escription, **D**iscovery, and **I**ntegration (**UDDI**): for publishing and locating services.





Source: https://www.w3.org/DesignIssues/WebServices.html

SOAP Web Services: Problems

- SOAP Web services didn't fit well with the Web Architecture.

- Big Web services are not simple: complex to implement and consume.
  - Every new layer creates failure points, leading to interoperability and scalability problems.
  - So many SOAP extensions.
  - Consuming clients need to support SOAP and requires the use of toolkits (for client code generation, etc.).

- No unified interface:
  - Use the Remote Procedure Call (RPC) concept.
  - The method and data are sent int the SOAP message body.
  - SOAP messages are sent using HTTP POST.

- All the requests to a given Web service are sent via HTTP POST to the same URI (end point).

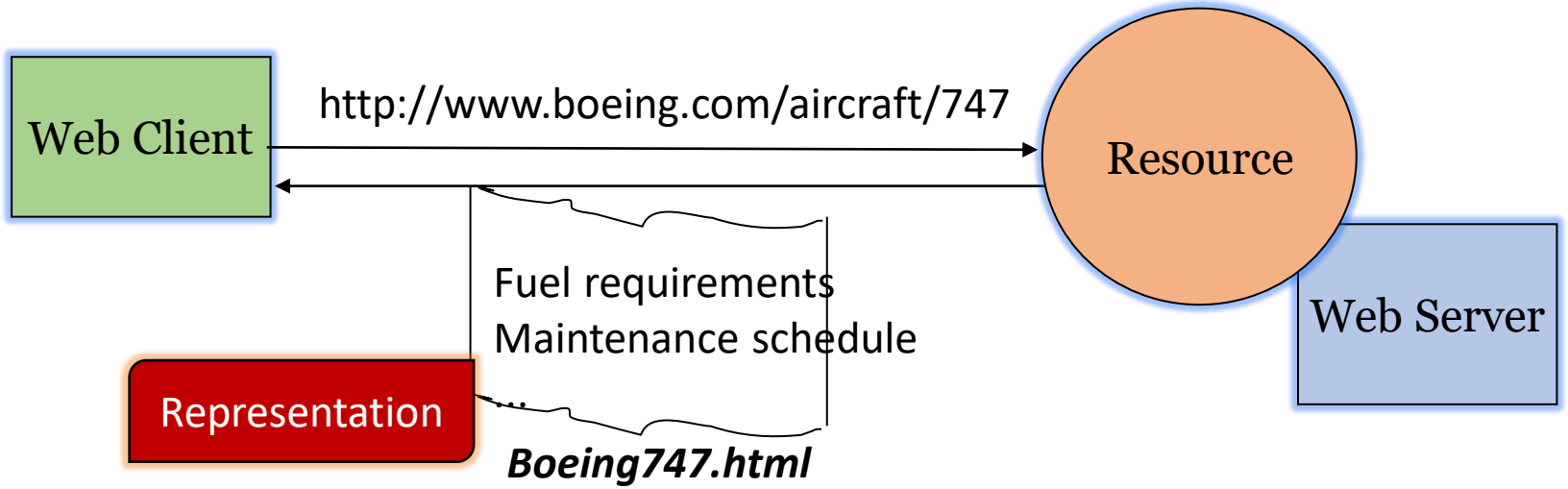
What is REST?

- REST stands for **RE**presentational **S**tate **T**ransfer.

- It was first formalized by Roy Fielding in Chapter 5 of his PhD thesis in 2000.
  - Fielding is one of the principal authors of the HTTP and URI RFCs, and is a co-founder of the Apache Software Foundation.
  - REST was introduced to describe a design pattern for implementing networked systems.
  - REST is an **architecture style**: a set of design principles that can be used to assess an architecture.
  - REST is a collection of network architecture principles which outline how resources are defined and addressed.

Why Is It Called Representational State Transfer?

➢ A typical Web Client-Web Server interaction:



- The Client invokes a Web resource using a URI.
- A **representation** of the resource is returned (in this case as an HTML document).
- The representation (*e.g.*, Boeing747.html) places the client in a new **state**.
  - When the client selects a hyperlink in Boeing747.html, it accesses another resource.
  - The **new representation** places the client application into yet another **state**.
  - Thus, the client application **transfers** state with each resource representation.

Why Is It Called Representational State Transfer?

- "**R**epresentational **S**tate **T**ransfer is intended to evoke an image of **how a** well-designed Web application **behaves**: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (**state transitions**), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use."

  –Roy Fielding

What is REST and Why We Need It?

- REST is not really a technology or a protocol, it is a model for designing and building lightweight Web services.

- REST is **not a standard**, but does prescribe the **use** of standards
  - It relies on several well-established standards: HTTP, URI, Media Types, JSON, XML, etc.

- REST is just a design pattern/an architecture style.
  - One can understand and follow its principles and constraints.

- REST builds on the fundamental idea behind the Web: everything on the Web is a ***resource***.

What is REST and Why We Need It?

- **REST** is a way to reunite the **human Web** (the one designed for our own use) **and** the programmable Web (designed for consumption by programs/applications).

- REST is simple to implement :
  - Uses existing Web standards
  - The necessary infrastructure has already become pervasive
  - RESTful Web services are lightweight and scalable
  - HTTP traverses firewall
  - Easy to consume by client applications

- Relies on HTTP and inherits its advantages, mainly: the ***ility** properties.

What Makes a Web Service RESTful?

- REST is not tied to any technology or platform.
  - It does not dictate how to implement a Web service.

- However, it introduces best practices known as ***constraints***
  - ✓ REST Constraint describe how the server should process requests and responds to them.

- Designed, implemented and operating within these constraints, the system (client applications and services) gains ***desirable properties***.

REST Constraints

To be considered **RESTful**, a Web service must satisfy all the **five** mandatory REST constraints.

- Fielding identifies **five** key **constrains** (and one optional) for a RESTful architecture:

1. Client-server
2. Stateless
3. Uniform interface
4. Cacheable
5. Layered system
6. Code on demand (optional)

gained system properties

*bility

- Scalability
- Flexibility
- Simplicity
- Modifiability
- Visibility
- Portability
- Reliability
- Addressability

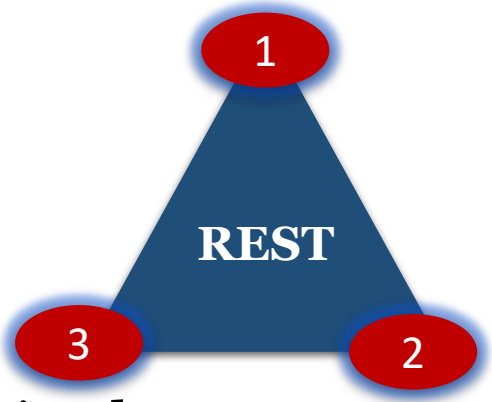
RESTful Web Service: Main Concepts

1. **Nouns:**
   - resource naming: URIs

2. **Representation:**
   - XML, JSON, etc.

3. **Verbs:** operations on resources (interaction) HTTP's uniform interface

Nouns (Resources)
unconstrained
i.e., http://api.example.com/employees/12345



Verbs
constrained
i.e., GET, POST, PUT, DELETE

Representations
constrained
i.e., XML, JSON

REST Architectural Elements

1 Data elements:

- **Resources:** the conceptual target of a hyperlink
- **Resource identifiers**: URI
- **Representations:** HTML/JSON/XML documents
- **Representation metadata:** media type, last-modified time, charset, etc.
- **Resource metadata:** source link, alternates
- **Control data:** if-modified-since, cache-control

2 Connectors:

- Client
- Server
- Cache
- Resolver (bind/DNS)
- Tunnel (SOCKS, SSL)

3 Components:

- Origin server
- Gateway
- Proxy
- User agent

SOAP vs RESTful Web Services

RESTful Web Services	vs	SOAP Web Services
▪ REST is not a protocol ▪ Support many formats ▪ Messages are smaller in size and consume lesser bandwidth ▪ Better in terms of performance with better caching support ▪ No third-party tools are required to consume REST Web services ▪ More flexible ▪ Less coupling between REST clients and server. ▪ Simple and lightweight ▪ Closer in design and philosophy to the Web.		▪ SOAP is a protocol. ▪ Use different protocols for communication: HTTP, SMTP, or FTP. ▪ SOAP uses only XML for messages ▪ SOAP clients are tightly coupled with the server. ▪ Hard to develop and require tools. ▪ SOAP WSs are complex. ▪ Use SOAP/HTTP - SOAP envelope, marshaling/unmarshalling overhead.