Web Services: SOAP y REST

Ingeniería de Sistemas

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

Tecnologías anteriores

70's	Comunicación de procesos en red	• Sockets
80's	Tecnologías de invocación de procedimientos remotos	• Sun RPC (Remote Procedure Call) • DCE (Distributed Computing Environment)
90's	Tecnologías de objetos distribuidos	• **CORBA (Common Object Request Broker Architecture)** • JAVA RMI (Remote Method Invocation) • Microsoft DCOM (Distributed Component Object Model)
00's	Tecnologías de Servicios Web	• REST (REpresentional State Transfer) • SOAP (Simple Object Access Protocol)

CORBA

- Common Object Request Broker Architecture (1991) es un estándar definido por Object Management Group (OMG).
- Permite que diversos componentes de software, escritos en múltiples lenguajes de programación y que corren en diferentes máquinas, puedan trabajar conjuntamente facilitando el desarrollo de aplicaciones distribuidas en entornos heterogéneos.

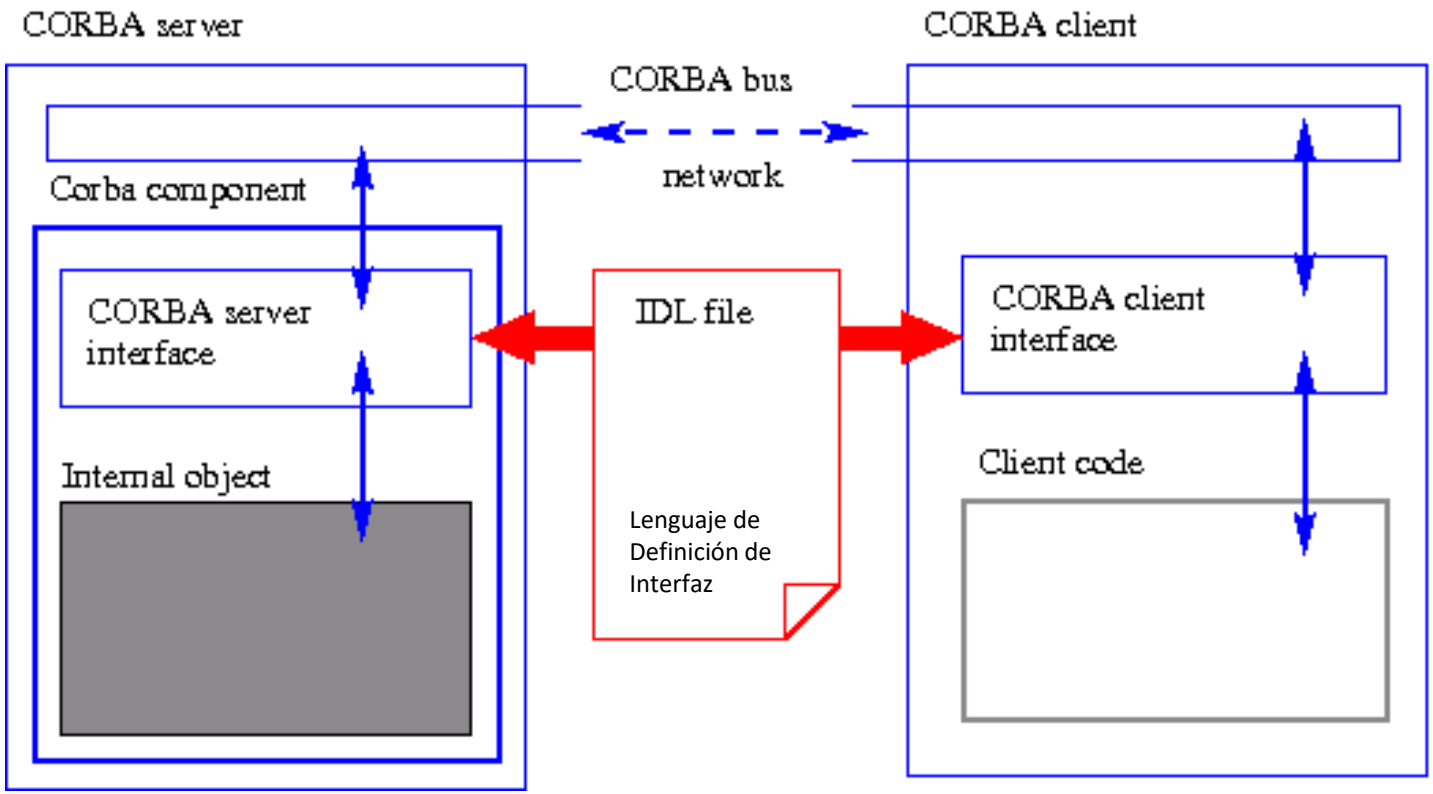
CORBA

- Oculta la programación a bajo nivel de aplicaciones distribuidas.
- Es una arquitectura orientada objetos, las funciones y los datos se agrupan en objetos y estos objetos pueden estar en diferentes máquinas, pero el programador accederá a ellos a través de funciones normales dentro de su programa.

Características

- Independencia del lenguaje de programación y sistema operativo. Libera a los desarrolladores de las limitaciones en cuanto al diseño del software.
- Posibilidad de interacción entre diferentes tecnologías. Normalizar las interfaces entre las diversas tecnologías y poder combinarlas.
- Transparencia de distribución. Ni cliente ni servidor necesitan saber si la aplicación está distribuida o centralizada.
- Transparencia de localización. El cliente no necesita saber dónde ejecuta el servicio y el servicio no necesita saber dónde ejecuta el cliente.
- Integración de software existente. Se amortiza la inversión previa reutilizando el software con el que se trabaja, incluso con sistemas heredados.
- Activación de objetos. Los objetos remotos no están en memoria permanentemente y se hace de manera invisible para el cliente.
- Tipado fuerte de datos, alta capacidad de configuración, libertad de elección de los detalles de transferencia de datos, o la compresión de los datos.

Arquitectura



Web Service

- Es la implementación más común de SOA extendiendo sus características, sobre todo en lo que respecta a interoperabilidad y reusabilidad.
- La Web proporciona un mecanismo de transporte universal, eficiente, robusto, escalable y probado tanto entre aplicaciones internas como con aplicaciones externas a la organización.
- Muchas de las tecnologías anteriores no se diseñaron para aprovechar este mecanismo o fueron diseñadas antes de la Web.

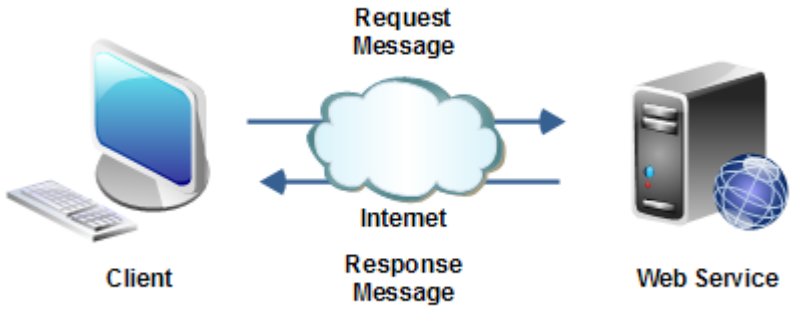
Definición

- Son sistemas software diseñados para soportar interacciones máquina a máquina a través de la red.
- Proporcionan una forma estándar de interoperar entre aplicaciones software que se ejecutan en diferentes plataformas, utilizando mensajes XML intercambiados mediante protocolos de Internet.
- Se identifican mediante URIs (Uniform Resource Identifier) y sus interfaces se pueden definir, describir y descubrir mediante documentos XML.

Definición

Componente de software que utiliza un conjunto de protocolos y estándares para intercambiar datos entre aplicaciones sobre una red.

OASIS (*Organization for the Advancement of Structured Information Standards*).



Características de los WS

- Exponen una interface bien definida soportada en estándares.
- Ocultan los detalles de implementación.
- Se invocan mediante mecanismos basados en estándares.
- Pueden ser de granularidad gruesa o fina.
- Son publicados por el proveedor para su consumo por uno o más clientes.
- Son desacoplados, autónomos e independientes.
- Son reutilizables al poder ser invocados por diferentes clientes.

Aspectos fundamentales

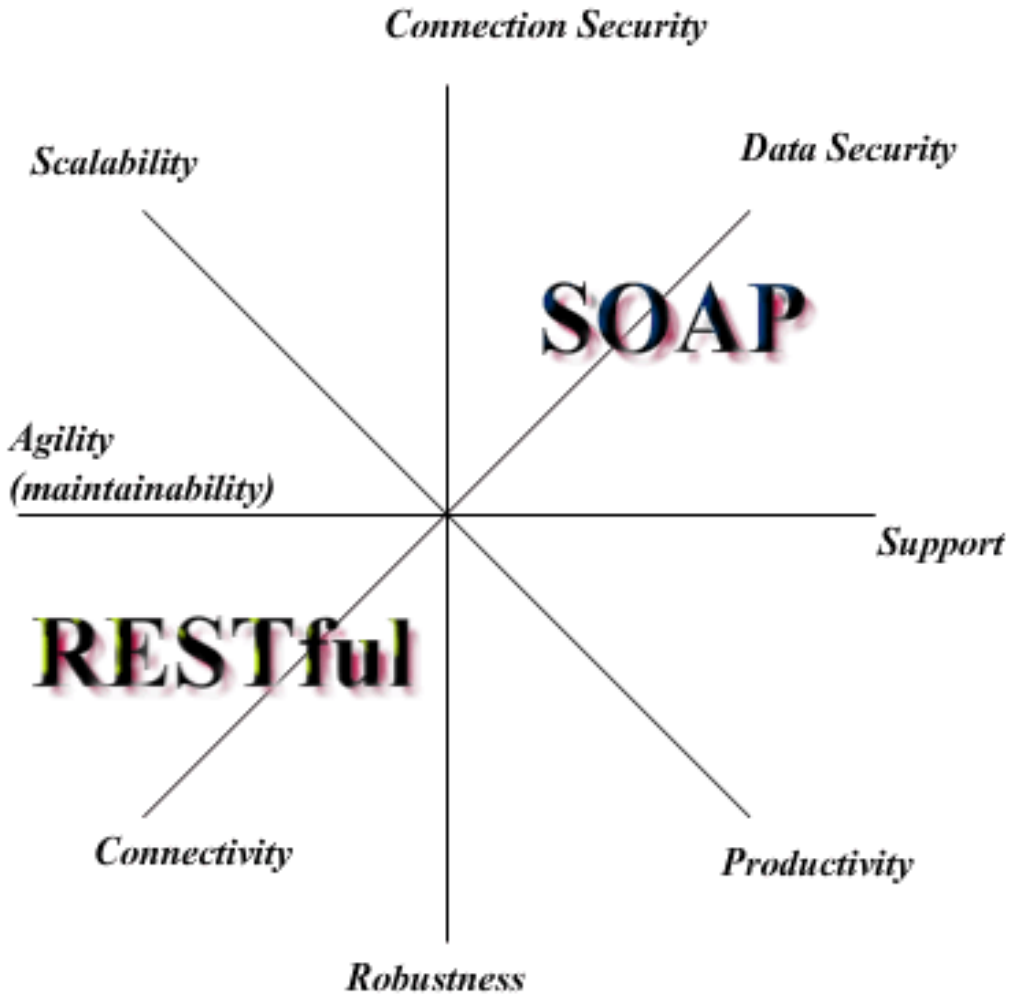
- Operaciones públicas que ofrece el servicio.
- Información sobre los tipos de datos que usa.
- Información sobre los protocolos de comunicación para acceso.
- Localización física del servicio.

Enfoques más utilizados

Los Web Services pueden implementarse en distintos estilos arquitectónicos. Los enfoques más utilizados son:

SOAP (*Simple Object Access Protocol*)

REST (*Representation State Transfer*)


Connection Security
Scalability
Data Security
SOAP
Agility
(maintainability)
Support
RESTful
Connectivity
Productivity
Robustness

Propósito de la Web

SOAP y REST difieren en el propósito de la Web:

- Para SOAP, la Web es el transporte universal de mensajes.
- Para REST, la Web es el universo de la información accesible globalmente.

Coincidencias

- Ambos estilos señalan cómo se deben cursar las peticiones de servicio a los servidores, la forma en la cual deben enviar los resultados, y cómo se deben publicar o dar a conocer los servicios que están accesibles a través de un servidor web.
- En ambas implementaciones el cliente no tiene conocimiento del lenguaje de implementación del servicio, que puede ser el mismo o diferente del cliente.

Diferencias

- En los servicios tipo REST el cliente utiliza una URL para invocar un proceso remoto. El cliente es responsable de la codificación de la información transmitida y recibida, así como para abrir y cerrar la conexión.

- En los servicios tipo SOAP una capa de software (*middleware)* aísla clientes de la codificación/decodificación de datos y de la gestión de la conexión, haciendo una invocación a un objeto remoto que puede ser definida en el mismo lenguaje de programación del cliente.

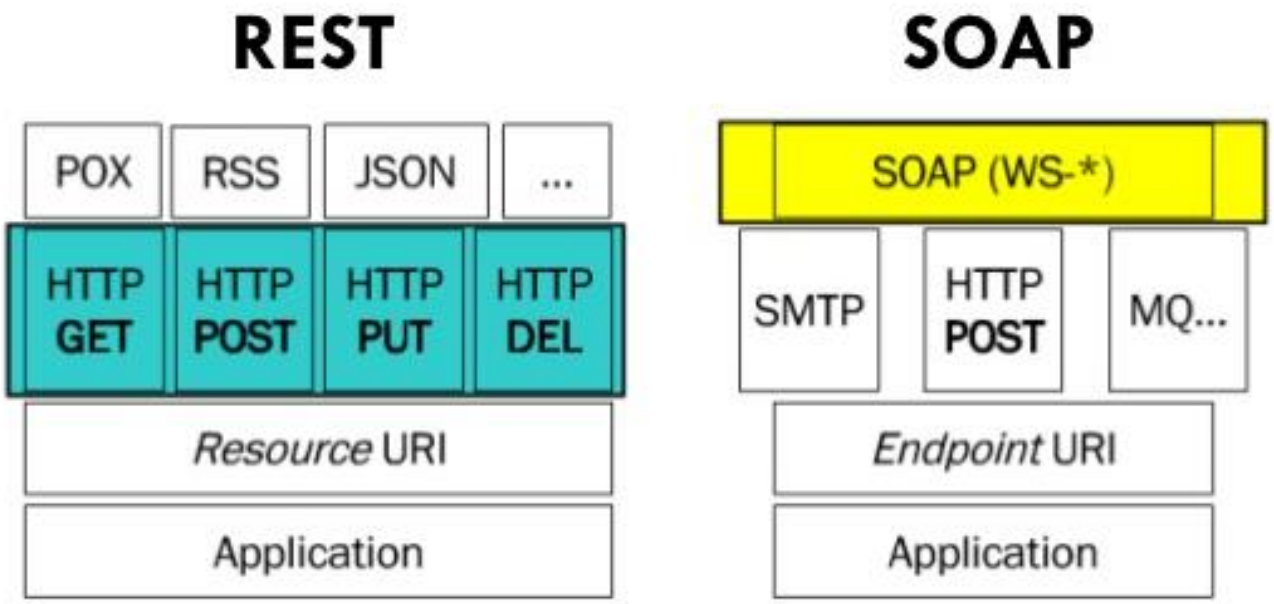
Diferencias

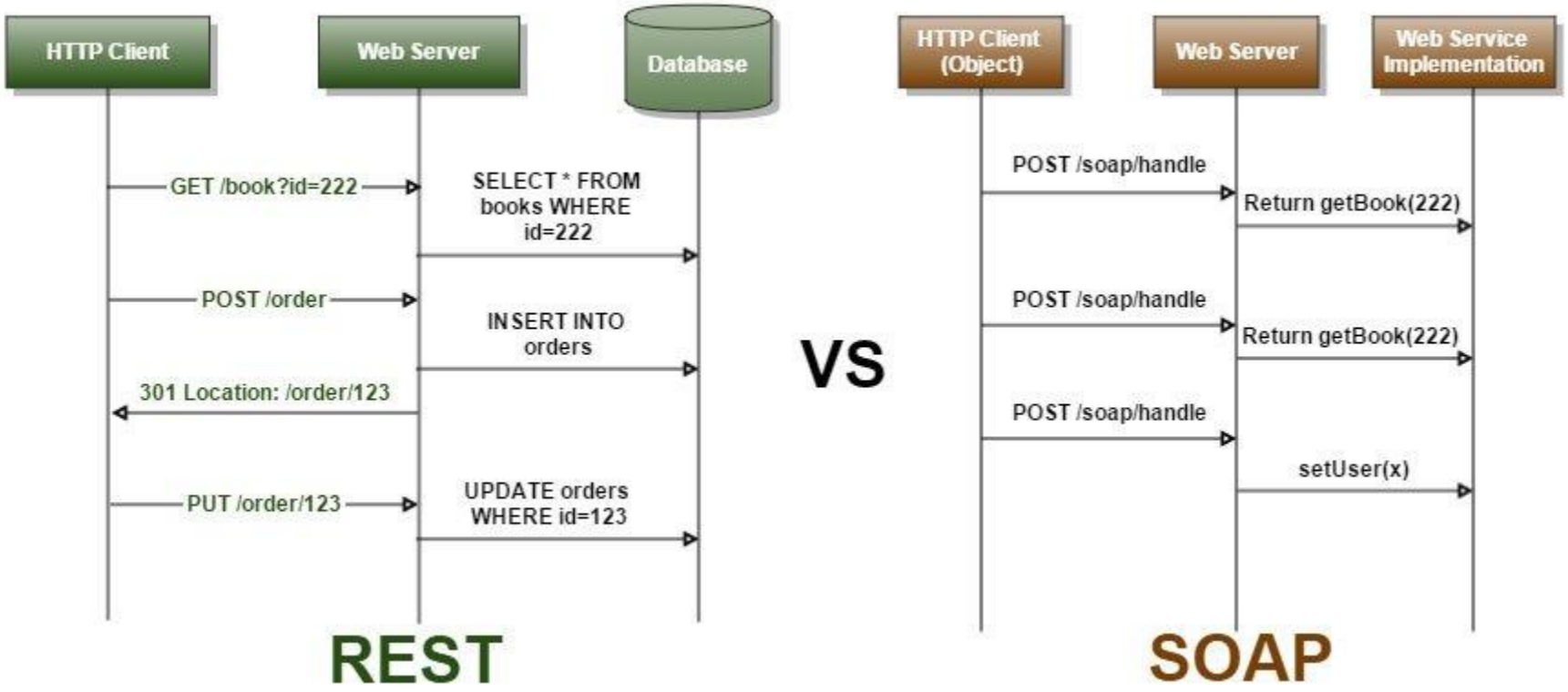
- En los servicios que implementan REST, el cliente realiza un esfuerzo importante para invocar el servicio, mientras que la invocación de servicios en SOAP es mucho más simple.

- La invocación de servicios RESTful consume menos recursos y tiempo para el cliente y proveedor de servicios, que los servicios web SOAP donde la capa de *middleware* tiene que ser implementada en ambos lados (cliente y proveedores de servicios).

Diferencias

- En los servicios tipo SOAP, la unidad básica de comunicación es el mensaje más que la operación.
- Los servicios basados en REST se centran más en interactuar con recursos y estados, que con mensajes y operaciones.


REST
POX
RSS
JSON
...
HTTP GET
HTTP POST
HTTP PUT
HTTP DEL
Resource URI
Application
SOAP
SOAP (WS-*)
SMTP
HTTP POST
MQ...
Endpoint URI
Application

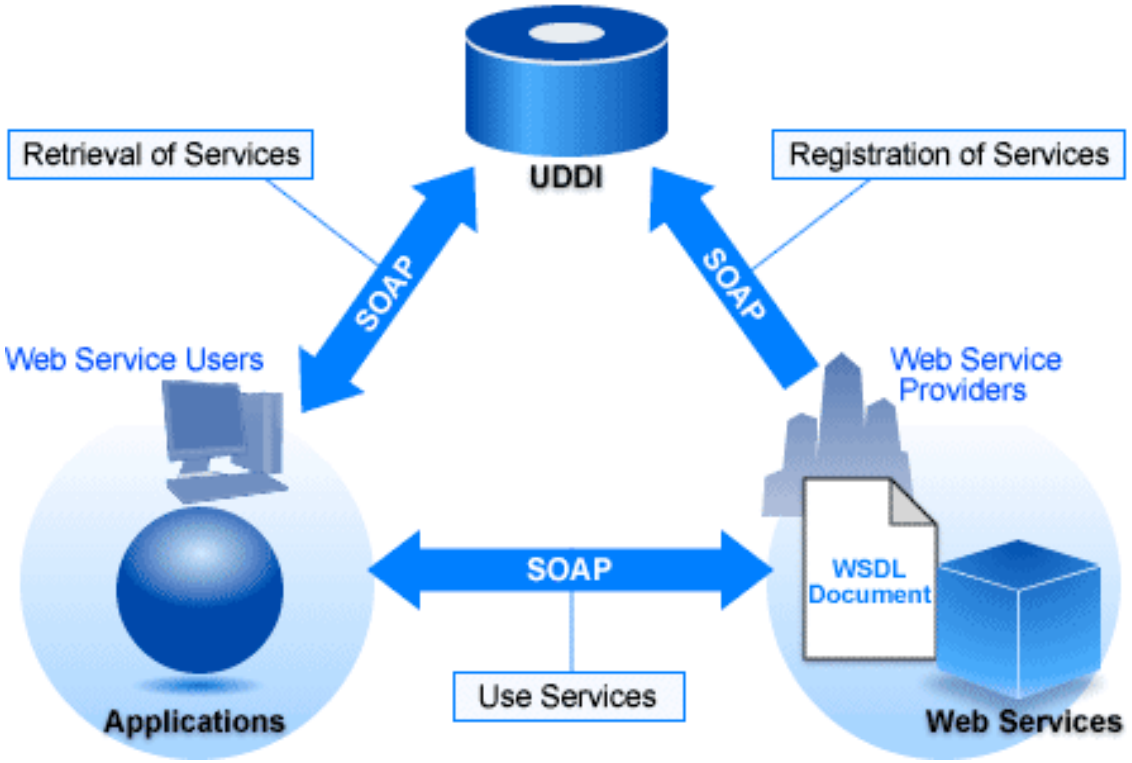

HTTP Client
Web Server
Database
GET /book?id=222
SELECT * FROM books WHERE id=222
POST /order
INSERT INTO orders
301 Location: /order/123
PUT /order/123
UPDATE orders WHERE id=123
REST
VS
HTTP Client (Object)
Web Server
Web Service Implementation
POST /soap/handle
Return getBook(222)
POST /soap/handle
Return getBook(222)
POST /soap/handle
setUser(x)
SOAP

Servicios SOAP

Se rigen por un conjunto de estándares de comunicación basados en XML:

- Protocolo SOAP (Simple Object Access Protocol) para el intercambio de datos.
- Lenguaje WSDL (Web Services Description Language) para describir las funcionalidades de un servicio Web.

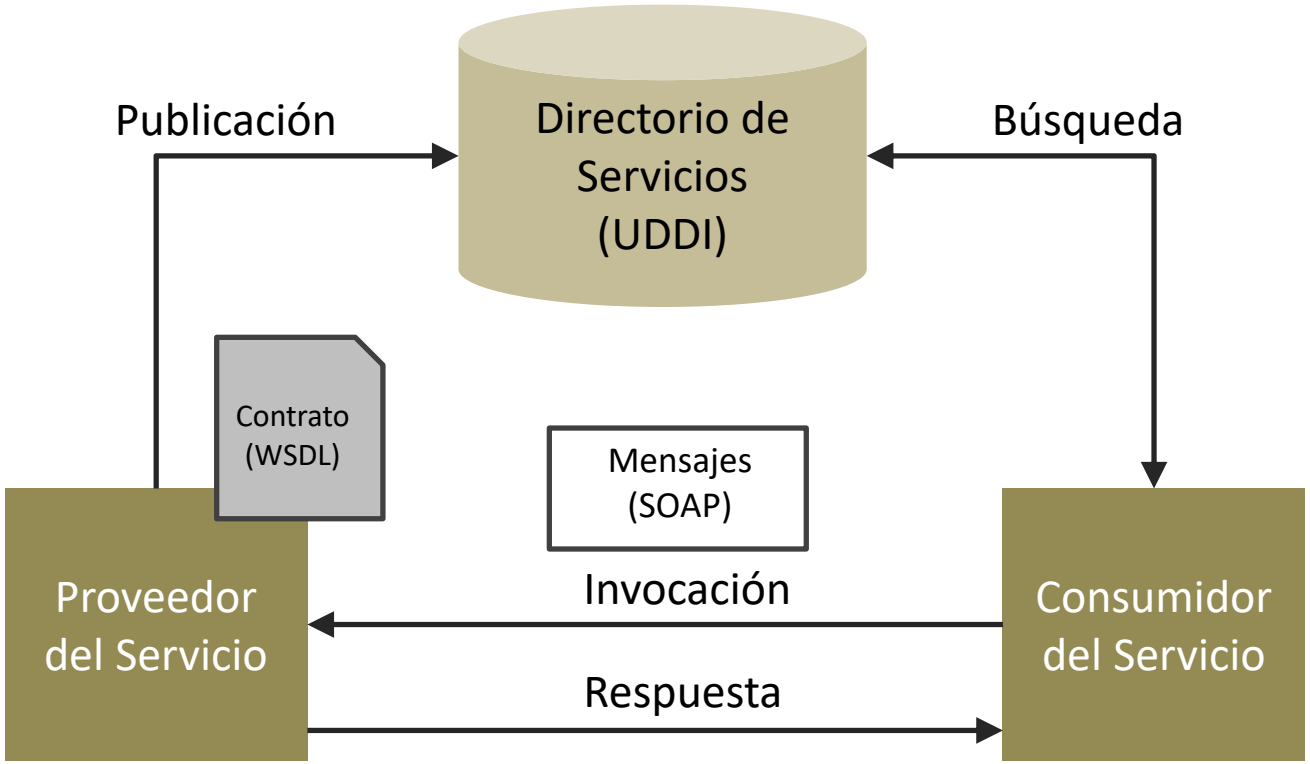
Arquitectura de WS



Elementos de la arquitectura

- **Proveedor del servicio.** Es el propietario del servicio (o plataforma que aloja el servicio).
- **Consumidor del servicio.** Sistema de información que necesita ciertas funciones de negocio (o aplicación que invoca un servicio).
- **Registro del servicio.** Directorio de servicios donde se publican las descripciones y donde se buscan.

Modelo de registro de servicios



Protocolos utilizados

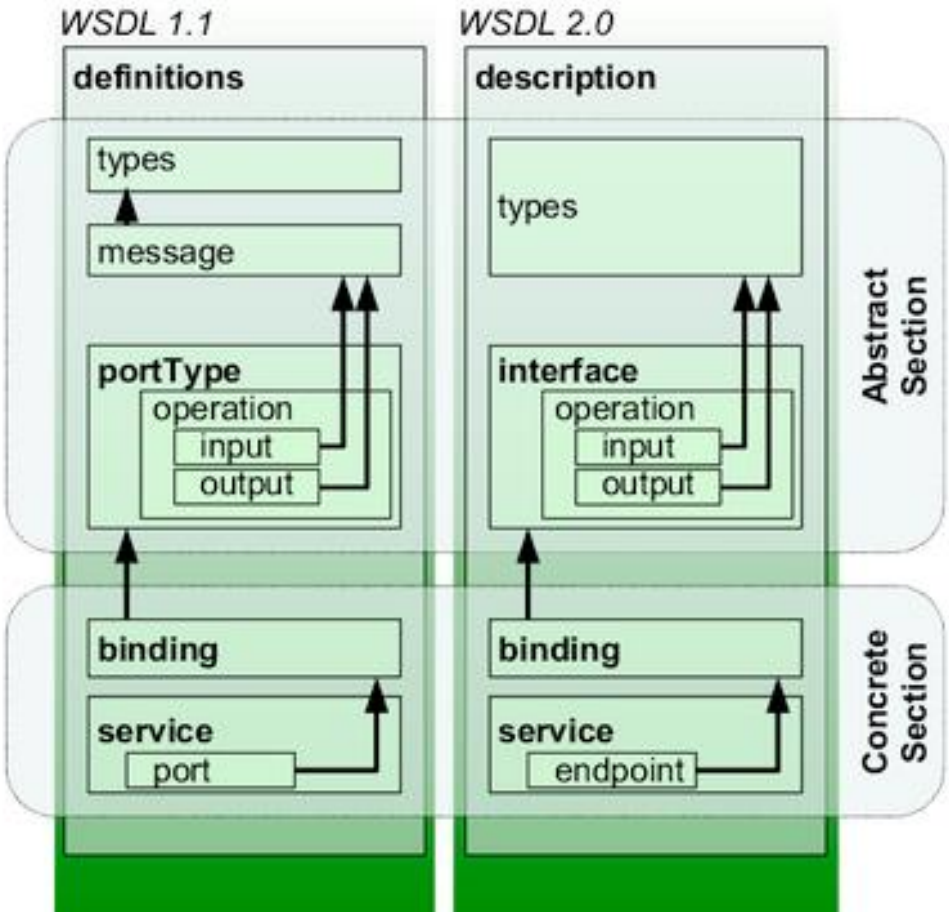
Localización de Servicios (*UDDI*)
Descripción de Servicios (*WSDL*)
Mensajería XML (*SOAP, **XML-RPC***)
Transporte de Servicios (*HTTP, **SMTP**, **FTP**, **BEEP**, **...***)

Capa	Descripción
Localización de servicios	Se encarga del registro centralizado de servicios, permitiendo que éstos sean anunciados y localizados. Se utiliza el protocolo UDDI.
Descripción de servicios	Se encarga de definir la interfaz pública de un determinado servicio. Está definición se realiza mediante WSDL.
Mensajería XML	Es la responsable de codificar los mensajes en XML de forma que puedan ser entendidos por cualquier aplicación. Puede implementar los protocolos XML-RPC o SOAP.
Transporte de servicios	Es la capa que se encarga de transportar los mensajes entre aplicaciones. Normalmente se utiliza el protocolo HTTP aunque también se pueden usar otros como SMTP, FTP o BEEP.

WSDL, UDDI y SOAP

- UDDI (Universal Description, Discovery and Integration): Catálogo donde se registran los servicios.
- WSDL (Web Services Definition Language): Es un lenguaje XML para describir servicios.
- SOAP: Protocolo estándar de mensajería para el intercambio de mensajes entre sistemas.

WSDL



Define el QUÉ hace el servicio:

- Operaciones disponibles
- Entradas, salidas y mensajes de error
- Definiciones de tipos para los mensajes.

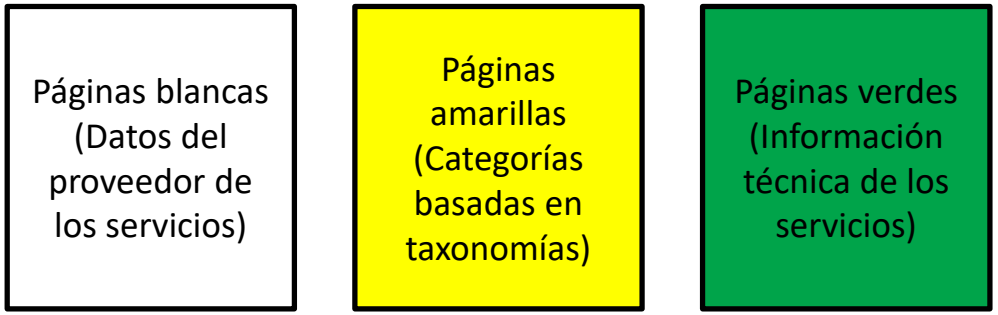
Define el CÓMO Y DÓNDE del servicio:

- Cómo se tiene que llamar (formato de los datos: SOAP)
- Protocolo de acceso (red)
- Dónde está el servicio (URL).

Elemento WSDL	Descripción
<?xml version=”1.0”>	Un documento WSDL es como cualquier documento XML y se basa en los esquemas, por lo que debe comenzar con dicha etiqueta.
<definitions>	Comienzo del documento, este tag agrupa a todos los demás elementos
<types>	Se definen los tipos de datos utilizados en los mensajes. Se utilizan los tipos definidos en la especificación de esquemas XML.
<message>	Se definen los métodos y parámetros para realizar la operación. Cada message puede consistir en una o más partes (parámetros). Las partes pueden ser de cualquiera de los tipos definidos en la sección anterior.
<portType>	Esta sección es la más importante, ya que definen las operaciones que pueden ser realizadas, y los mensajes que involucran (por ejemplo el mensaje de petición y el de respuesta).
<binding>	Se definen el formato del mensaje y detalles del protocolo para cada portType.
<service>	Localización física del servico

UDDI

- Proporciona los mecanismos para publicar servicios en un registro, y para que sus consumidores puedan buscarlos y encontrar su localización física.
- Define una especificación técnica para la construcción de un directorio distribuido de servicios, un lenguaje XML para el almacenamiento de los datos y un servicio web (basado en SOAP) para guardar y recuperar información de los registros UDDI.

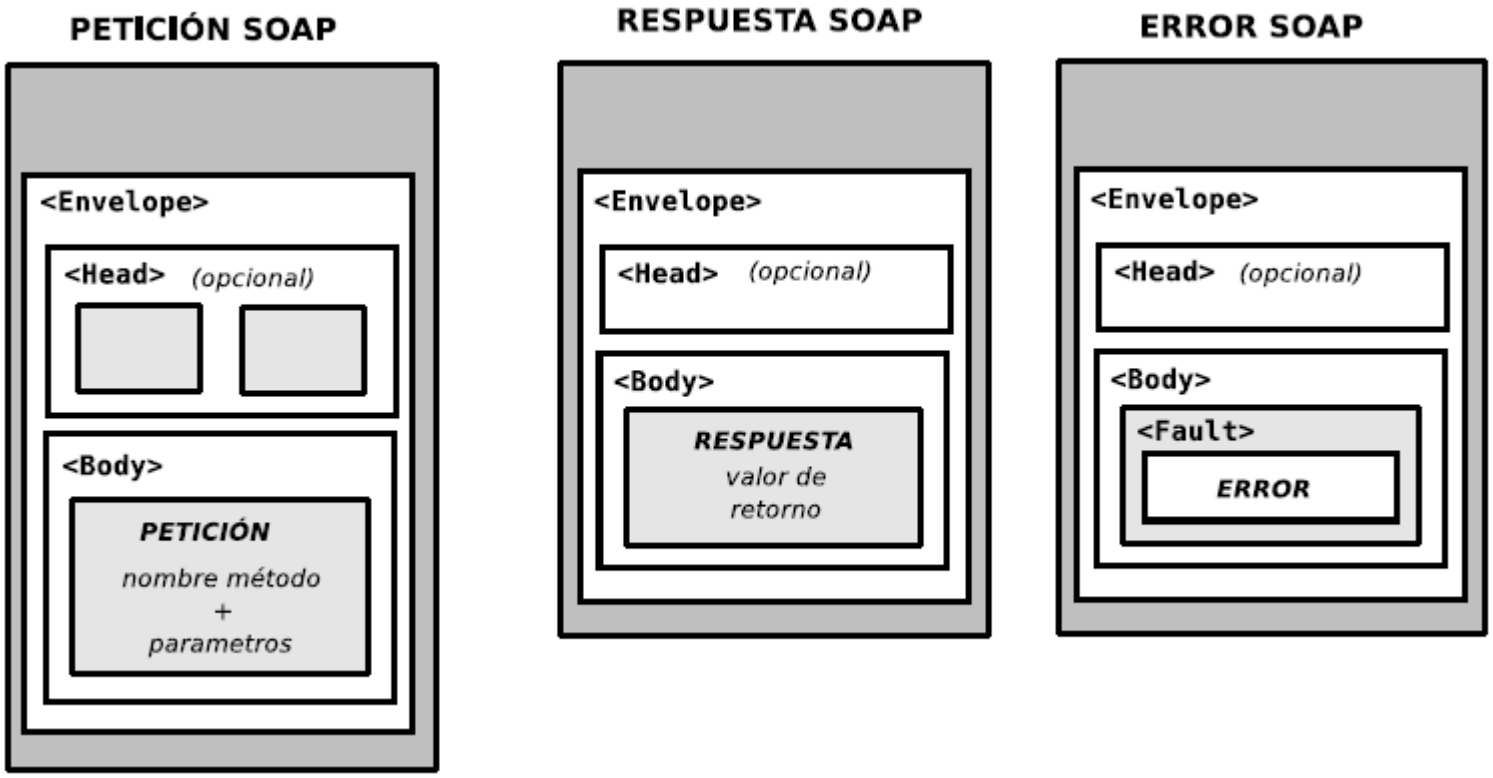


Petición y respuesta

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetails xmlns="http://warehouse.example.com/ws">
      <productID>827635</productID>
    </getProductDetails>
  </soap:Body>
</soap:Envelope>
```

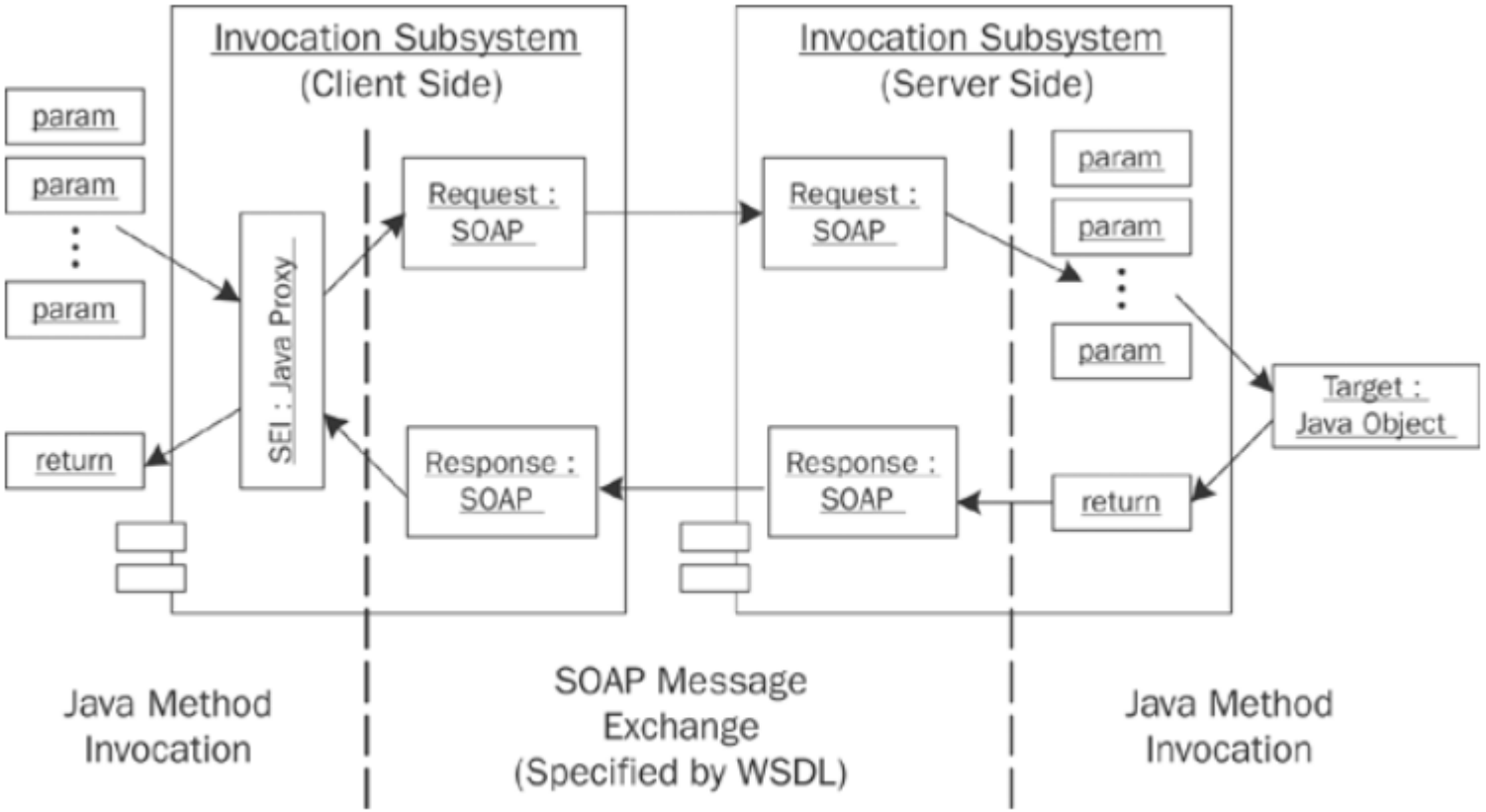
```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetailsResponse xmlns="http://warehouse.example.com/ws">
      <getProductDetailsResult>
        <productName>Toptimate 3-Piece Set</productName>
        <productID>827635</productID>
        <description>3-Piece luggage set. Black Polyester.</description>
        <price currency="NIS">96.50</price>
        <inStock>true</inStock>
      </getProductDetailsResult>
    </getProductDetailsResponse>
  </soap:Body>
</soap:Envelope>
```

Estructura de mensaje



Intercambio de mensajes

- El mensaje que se envía desde la aplicación cliente a la aplicación servidor, solicitando la ejecución de un método al que se pasan una serie de parámetros.
- El subsistema de invocación cliente se traduce en una llamada al método del proxy SEI (Service Endpoint Interface) dentro de la solicitud/respuesta SOAP y viceversa.
- La invocación del subsistema por lado del servidor de la solicitud/respuesta SOAP se traduce a una llamada al método Java (Java Object).
- El mensaje que se envía desde la aplicación servidor al cliente, y que contiene datos XML con los resultados de la ejecución del método solicitado.


Invocation Subsystem
(Client Side)
Invocation Subsystem
(Server Side)
param
param
param
return
SEI : Java Proxy
Request :
SOAP
Response :
SOAP
Request :
SOAP
Response :
SOAP
param
param
param
return
Target :
Java Object
Java Method
Invocation
SOAP Message
Exchange
(Specified by WSDL)
Java Method
Invocation

Características	• **Las operaciones son definidas como puertos WSDL.** • **Dirección única para todas las operaciones.** • **Múltiple instancias del proceso comparten la misma operación.** • **Componentes fuertemente acoplados.**
Ventajas	• Fácil de utilizar por lo general. • La depuración es posible. • Las operaciones complejas se pueden esconder detrás de una fachada. • Envolver APIs existentes es sencillo. • Incrementa la privacidad. • Herramientas de desarrollo.
Desventajas	• Los clientes necesitan saber las operaciones y su semántica antes de usarlas. • Los clientes necesitan puertos dedicados para diferentes tipos de notificaciones. • Las instancias del proceso son creadas implícitamente.

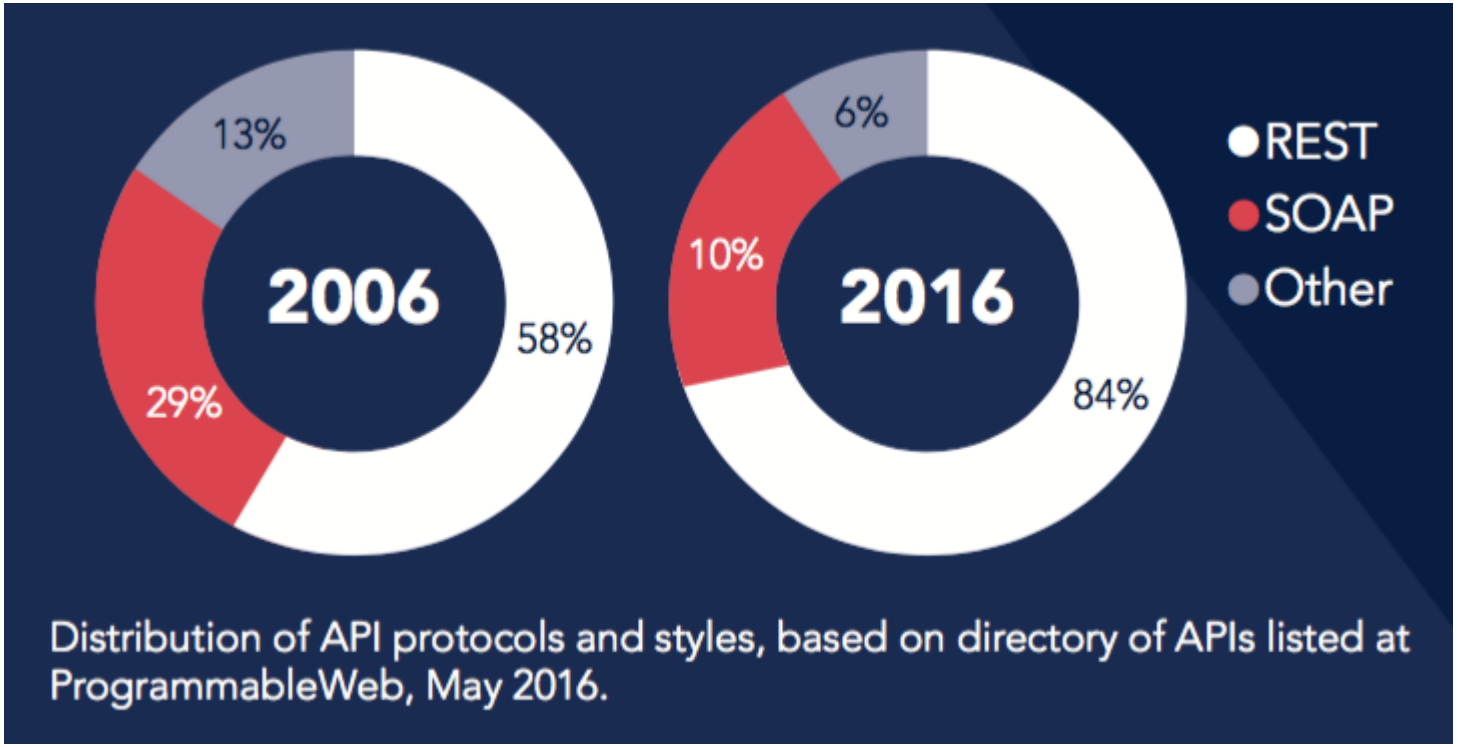
REST

- Representation State Transfer (REST) es un estilo de arquitectura para sistemas hipermedia distribuidos (Fielding, 2000).
- Proporciona un conjunto de restricciones arquitectónicas que insisten en la escalabilidad de las interacciones de los componentes, la generalidad de interfaces, implementación independiente de los componentes y componentes de mediación, con el fin de reducir la latencia de la interacción, reforzar la seguridad, y encapsular los sistemas heredados.

La motivación de REST es capturar las características de la Web como plataforma de procesamiento distribuido.

Definición

- No es un estándar aunque hace uso de diferentes estándares: HTTP, XML, URL, HTML, etc.
- Define un grupo de principios arquitectónicos por los cuales se diseñan servicios web haciendo énfasis en los recursos del sistema, incluyendo cómo se accede al estado de dichos recursos y cómo se transfieren por HTTP hacia clientes escritos en diversos lenguajes.
- Se ha convertido en el modelo predominante para el diseño de servicios, desplazando a SOAP y WSDL por tener un estilo más sencillo de utilizar.



Distribution of API protocols and styles, based on directory of APIs listed at ProgrammableWeb, May 2016.

Use in Technology driven sectors

REST

- Social Media
- Web Chat
- Mobile

SOAP

- Financial
- Telecommunication
- Payment Gateways

Objetivos de REST

- **Escalabilidad de la interacción con los componentes.** La Web ha crecido exponencialmente sin degradar su rendimiento. Una prueba de ellos es la variedad de clientes que pueden acceder a través de la Web: estaciones de trabajo, sistemas industriales, dispositivos móviles, etc.
- **Generalidad de interfaces.** Gracias al protocolo HTTP, cualquier cliente puede interactuar con cualquier servidor HTTP sin ninguna configuración especial.

Objetivos de REST

- **Puesta en funcionamiento independiente.** Los clientes y servidores pueden ponerse en funcionamiento durante años. Por tanto, los servidores antiguos deben ser capaces de entenderse con clientes actuales y viceversa. Diseñar un protocolo que permita este tipo de características resulta muy complicado. HTTP permite la extensibilidad mediante el uso de las cabeceras, a través de las URIs, a través de la habilidad para crear nuevos métodos y tipos de contenido.
- **Compatibilidad con componentes intermedios.** Los más populares intermediaros son varios tipos de proxys para Web. Algunos como las caches, se utilizan para mejorar el rendimiento. Otros como los firewalls permiten reforzar las políticas de seguridad. Y por último, los gateways permiten encapsular sistemas no propiamente Web. Por tanto, la compatibilidad con intermediarios nos permite reducir la latencia de interacción, reforzar la seguridad y encapsular otros sistemas.

Características de REST

- Es un sistema cliente-servidor. Los clientes generan solicitudes a los servidores y estos la procesan, generando una respuesta apropiada.
- Las solicitudes y respuestas se construyen alrededor de la transferencia de representaciones de los recursos. Un recurso puede ser esencialmente cualquier concepto que pueda ser tratado. Una representación de un recurso es típicamente un documento que captura el estado actual o previsto de un recurso.
- Utiliza sustantivos y verbos como lenguaje haciendo énfasis en la legibilidad. No requiere el análisis de XML y no requiere de una cabecera de mensaje hacia y desde un proveedor de servicios, haciendo por tanto un menor consumo de ancho de banda durante la comunicación.

Principios arquitectónicos

- Interfaz uniforme para la identificación de los recursos.
- Utilizar los métodos estándar de HTTP.
- Comunicación sin mantener estados.
- Recursos con múltiples representaciones.

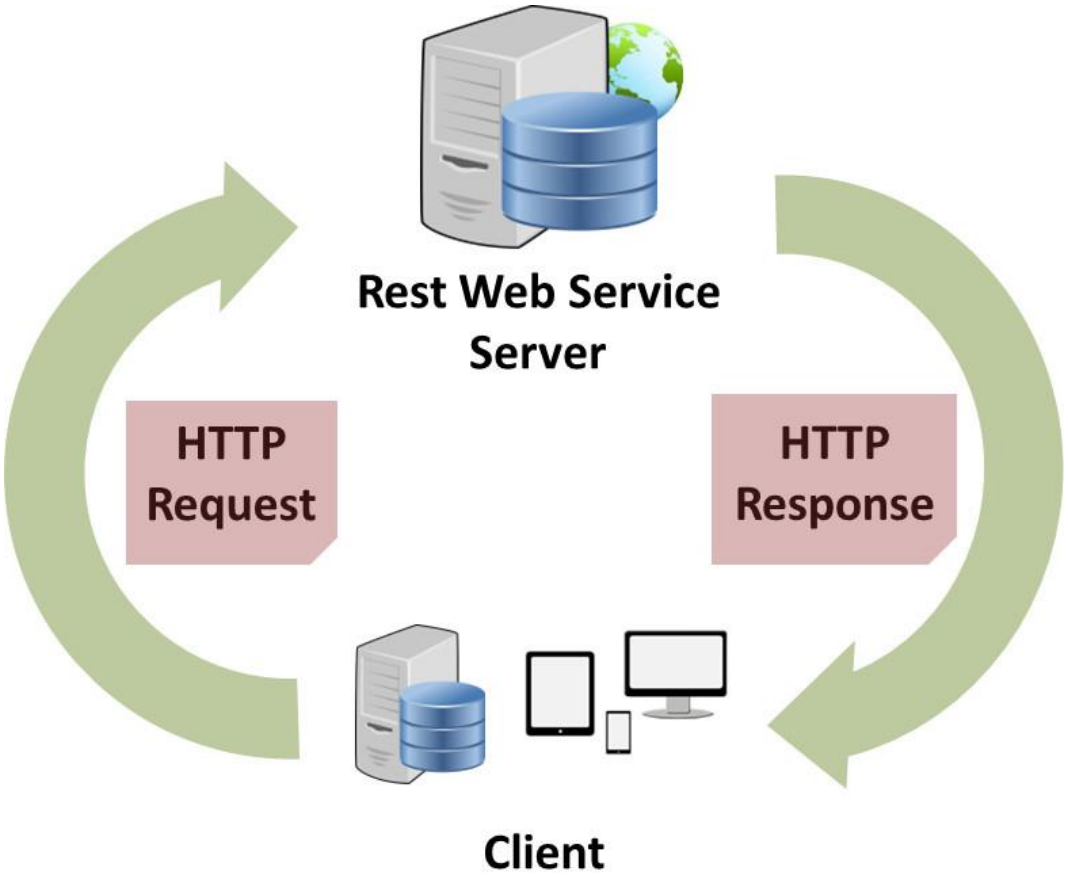
Interfaz uniforme

- La identificación de los recursos debe ser únicamente a través de su URI (*Uniform Resource Identifier*).
- Su representación se basa en una estructura al estilo de directorios, manteniendo una jerarquía, con una única ruta raíz y que ésta a su vez tenga sub-rutas, como un sistema basado en reglas.

REGLAS	RECURSOS
http://ejemplo.com/materiales/{temas}/{id}	http://ejemplo.com/materiales/arte/24
http://ejemplo.com/materiales/{año}/{mes}/{dia}/{temas}/{id}	http://ejemplo.com/materiales/2012/12/10/politica/66
http://ejemplo.com/materiales/{temas}/{zona}/{id}	http://ejemplo.com/materiales/politica/medioriente/33
http://ejemplo.com/materiales/{temas}/{zona}/{pais}/{id}	http://ejemplo.com/materiales/politica/medioriente/irak/45

Métodos estándar de HTTP

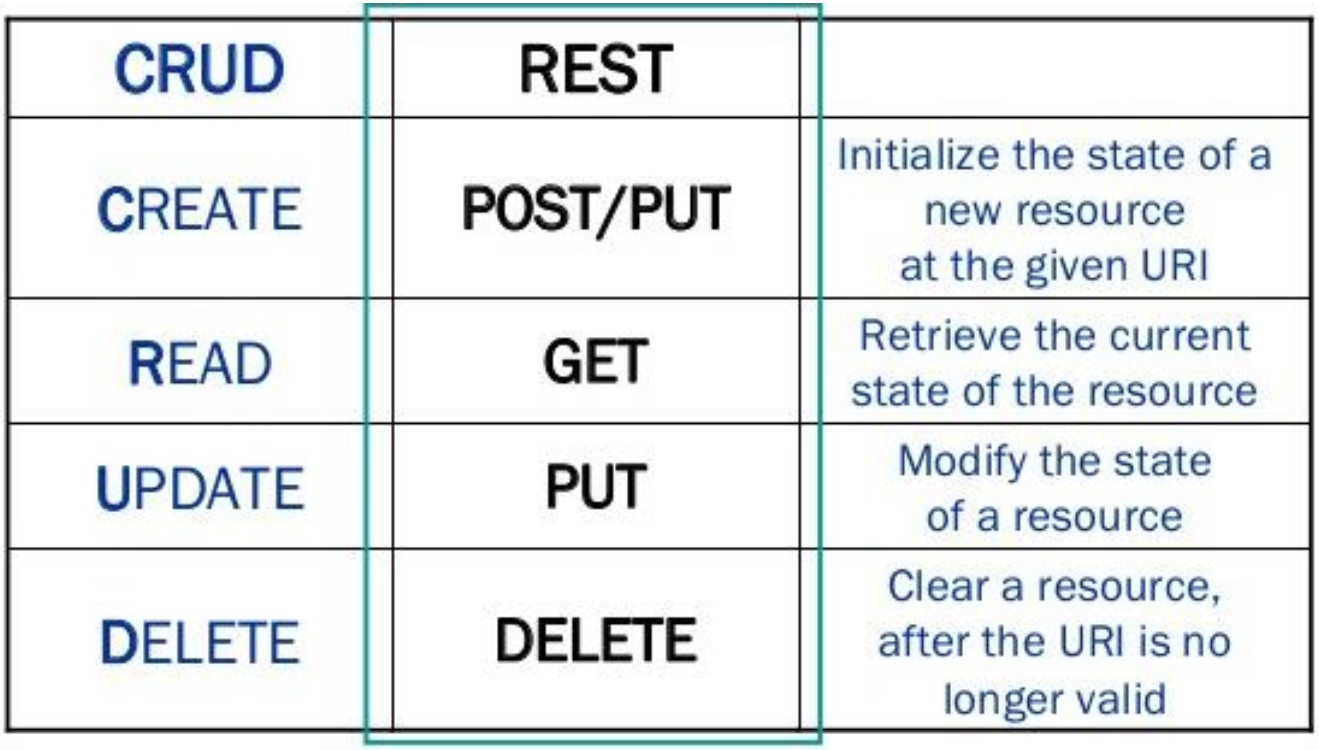
- Una de las características clave de los servicios tipo REST es la utilización de los métodos estándar HTTP (**GET**, **PUT**, **POST**, **DELETE**), siguiendo las definiciones existentes en el protocolo (**RFC2616**).
- Estos métodos hacen posible las principales acciones que se pueden realizar con los recursos.
- No se conoce cómo el servicio implementa el almacenamiento de los datos ni qué tecnologías se utilizan para implementar el servicio.
- Todo lo que se conoce es que cliente y servidor se comunican a través de HTTP, que se usa dicho protocolo de comunicaciones para enviar peticiones, y que las representaciones de los recursos se intercambian entre el cliente y el servidor a través del intercambio de URIs.



REST permite que los desarrolladores utilicen estos métodos, manteniendo una consistencia con el protocolo.

Métodos estándar de HTTP

- **GET** para obtener el estado de un recurso del servidor.
- **PUT** para cambiar el estado de un recurso o actualizarlo.
- **POST/PUT** inicializa el estado de un nuevo recurso en el servidor.
- **DELETE** para eliminar un recurso del servidor.



CRUD	REST	
CREATE	POST/PUT	Initialize the state of a new resource at the given URI
READ	GET	Retrieve the current state of the resource
UPDATE	PUT	Modify the state of a resource
DELETE	DELETE	Clear a resource, after the URI is no longer valid

Relación entre las operaciones CRUD y los métodos estándar HTTP

Recurso	GET	PUT	POST	DELETE
http://ejemplo.com/ordenes/	Retorna la colección de órdenes	Reemplaza la colección entera por otra	Añade una nueva orden a la colección	Elimina la colección
http://ejemplo.com/ordenes/10	Retorna los detalles de la orden con id 10	Actualiza la orden por otra, sino existe la crea		Elimina la orden con id 10 de la colección
http://ejemplo.com/clientes/21/ordenes/	Retorna la colección de órdenes que pertenecen al cliente con id 21	Reemplaza la colección entera del cliente con id 21 por otra	Añade una nueva orden a la colección del cliente con id 21	Elimina la colección de órdenes del cliente con id 21

GET/RETRIEVE

- El método GET se utiliza para **RECUPERAR** el recurso.
- No se recomienda usar el método para actualizar el recurso. Una operación GET debe ser segura e idempotente.
- Primero se debe determinar cuál es el recurso que se va a manejar y el tipo de representación que se va a utilizar.
- El ejemplo a seguir es un servicio web que gestiona alumnos en un curso, con la URI: *http://restfuljava.com*.

GET/RETRIEVE

Se define la siguiente representación para el recurso alumno:

```
<alumno>
    <nombre>Esther</nombre>
    <edad>15</edad>
    <link>/alumnos/Esther</link>
</alumno>
```

Se definimos la siguiente representación para el recurso lista de alumnos:

```
<alumnos>
    <alumno>
        <nombre>Esther</nombre>
        <edad>15</edad>
        <link>/alumnos/Esther</link>
    <alumno>
    <alumno>
        <nombre>Juan</nombre>
        <edad>15</edad>
        <link>/alumnos/Juan</link>
    <alumno>
</alumnos>
```

GET/RETRIEVE

- Una vez definida la representación, se asume que las URIs tienen la forma: *http://restfuljava.com/alumnos* para acceder a la lista de alumnos, y *http://restfuljava.com/alumnos/{nombre}* para acceder a un alumno específico con el identificador con el valor *nombre*.
- Se hace las peticiones sobre el servicio:
  - Si se quiere obtener la información de un alumno, realizamos una petición a la URI con su nombre: *http://restfuljava.com/alumnos/Esther*
  - Si se quiere obtener la información de la lista de alumnos, realizamos una petición a la URI: *http://restfuljava.com/alumnos*

HTTP GET

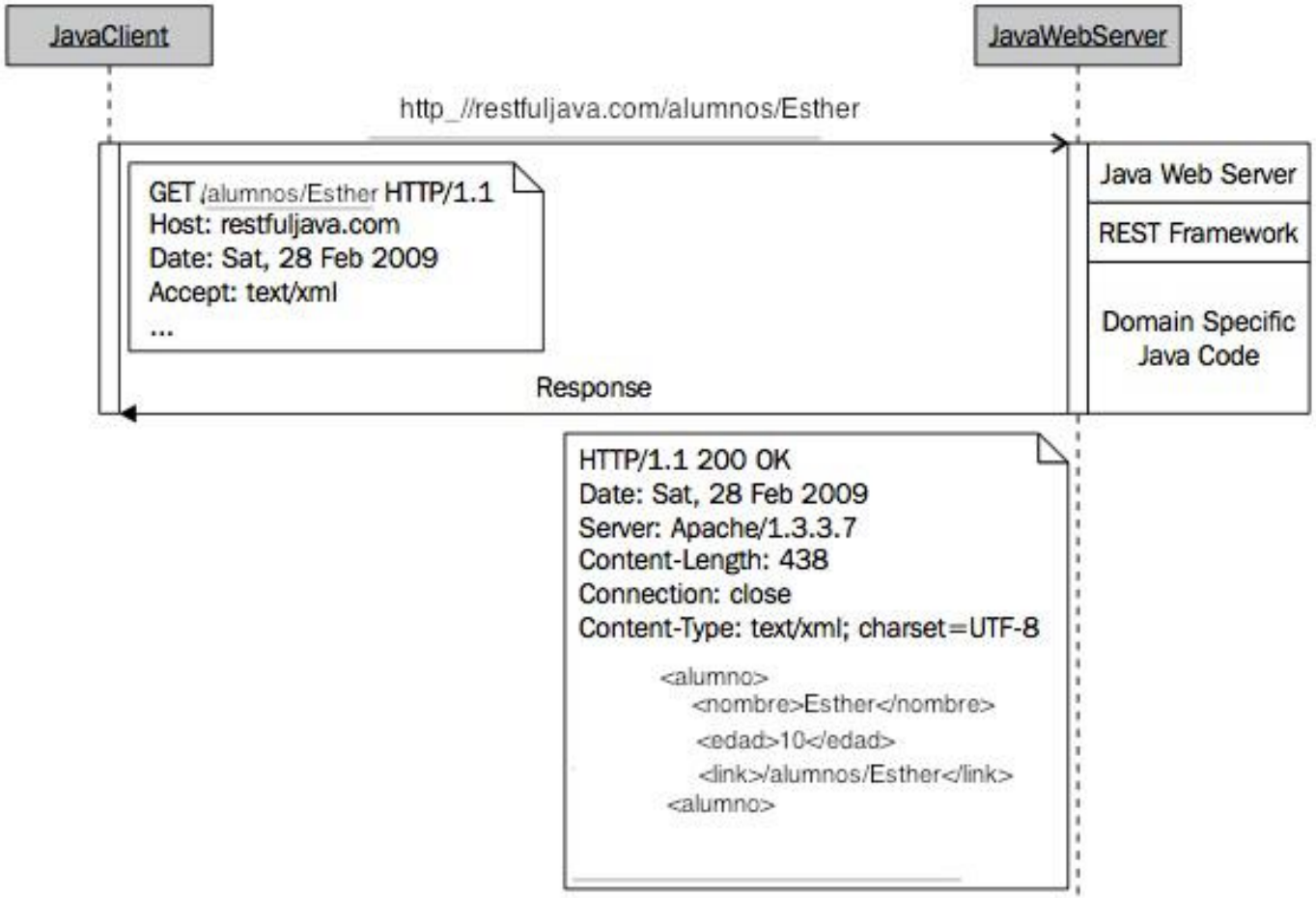
Petición 1. La respuesta es una representación del alumno con nombre *Esther*:

```
<alumno>
    <nombre>Esther</nombre>
    <edad>15</edad>
    <link>/alumnos/Esther</link>
</alumno>
```

Petición 2. La respuesta es una representación de la lista de alumnos (suponiendo que solo hay dos):

```
<alumnos>
    <alumno>
        <nombre>Esther</nombre>
        <edad>15</edad>
        <link>/alumnos/Esther</link>
    <alumno>
    <alumno>
        <nombre>Juan</nombre>
        <edad>15</edad>
        <link>/alumnos/Juan</link>
    <alumno>
</alumnos>
```

Secuencia de la petición



Secuencia de la petición

- Un cliente Java realiza una petición HTTP con el método GET y Esther es el identificador del alumno. El cliente establece la representación solicitada a través del campo de cabecera Accept.
- El servidor web recibe e interpreta la petición GET como una acción RETRIEVE. En este momento, el servidor web cede el control al framework RESTful para gestionar la petición. Remarquemos que los frameworks RESTful no recuperan de forma automática los recursos, ése no es su trabajo. La función del framework es facilitar la implementación de las restricciones REST. La lógica de negocio y la implementación del almacenamiento es el papel del código Java específico del dominio.
- El programa del lado del servidor busca el recurso Esther. Encontrar el recurso podría significar buscarlo en una base de datos, un sistema de ficheros, o una llamada a otro servicio web.
- Una vez que el programa encuentra a Esther, convierte el dato binario del recurso a la representación solicitada por el cliente.
- Con la representación convertida a XML, el servidor envía de vuelta una respuesta HTTP con un código numérico de 200 (Ok) junto con la representación solicitada. Si hay algún error, el servidor HTTP devuelve el código numérico correspondiente, pero es el cliente el que debe tratar de forma adecuada el fallo. El fallo más común es que el recurso no exista, en cuyo caso se devolvería el código 404 (Not Found).

Códigos HTTP

- Todos los mensajes entre el cliente y el servidor son llamadas del protocolo estándar HTTP. Para cada acción de recuperación, enviamos una petición GET y obtenemos una respuesta HTTP con la representación del recurso solicitada, o bien, si hay un fallo, el correspondiente código de error (por ejemplo, 404 Not Found si un recurso no se encuentra; 500 Internal Server Error si hay un problema con el código Java en forma de una excepción).
- En las peticiones de recuperación de datos resulta recomendable también implementar un sistema de caché. Para hacer esto utilizaremos el código de respuesta 304 Not Modified en caso de que los datos no hubiesen cambiado desde la última petición que realizamos (se podría pasar un parámetro con la fecha en la que obtuvimos la representación por última vez). De esta forma, si un cliente recibe ese código como respuesta, sabe que puede seguir trabajando con la representación de la que ya dispone, sin tener que descargar una nueva.

POST/CREATE

- El método POST se utiliza para CREAR recursos. Las peticiones POST no son idempotentes.
- De nuevo, la URI para añadir un nuevo alumno a la lista será: http://restfuljava.com/alumnos. El tipo de método para la petición lo determina el cliente.
- Se asume que el alumno con nombre Manuel no existe en la lista. La nueva representación XML es:

```
<alumno>
    <nombre>Manuel</nombre>
    <edad>14</edad>
    <link></link>
</alumno>
```

Secuencia de la petición

1. Un cliente Java realiza una petición HTTP a la URI *http://restfuljava.com/alumnos*, con el método HTTP POST. La petición POST incluye una representación en forma de XML de *Manuel.*
2. El servidor web recibe la petición y delega en el *framework* REST para que la gestione; el código dentro del *framework* ejecuta los comandos adecuados para almacenar dicha representación.
3. Una vez que se ha completado el almacenamiento del nuevo recurso, se envía una respuesta de vuelta: si no ha habido ningún error, se envía el código 201 (Created); si se produce un fallo, se envía el código de error adecuado. Además, se puede devolver en la cabecera Location la URL que dará acceso al recurso recién creado.

   Location: http://restfuljava.com/alumnos/Manuel

PUT/UPDATE

- El método PUT se utiliza para ACTUALIZAR (modificar) recursos, o para crearlos si el recurso en la URI especificada no existe. Es decir, PUT se utiliza para establecer un determinado recurso, dada su URI, a la representación proporcionada, exista o no.
- Para actualizar un recurso, primero se necesita su representación en el cliente; en segundo lugar, en el lado del cliente se actualiza el recurso con los nuevos valores deseados; y finalmente, se actualiza el recurso mediante una petición PUT, adjuntando la representación correspondiente.
- Para modificar la edad de Esther, de 15 a 16, la nueva representación será:

```
<alumno>
        <nombre>Esther</nombre>
        <edad>16</edad>
        <link>/alumnos/Esther</link>
</alumno>
```

Secuencia de la petición

- Un cliente Java realiza una petición HTTP PUT a la URI http://restfuljava.com/alumnos/Esther, incluyendo la nueva definición XML.
- El servidor web recibe la petición y delega en el framework REST para que la gestione; el código dentro del framework ejecuta los comandos adecuados para actualizar la representación de Esther.
- Una vez que se ha completado la actualización, se envía una respuesta al cliente. Si el recurso que se ha enviado no existía previamente, se devolverá el código 201 (Created). En caso de que ya existiese, se podría devolver 200 (Ok) con el recurso actualizado como contenido, o simplemente 204 (No Content) para indicar que la operación se ha realizado correctamente sin devolver ningún contenido.

Diferencia entre PUT y POST

POST: Publica datos en un determinado recurso. El recurso debe existir previamente, y los datos enviados son añadidos a él. Por ejemplo, para añadir nuevos alumnos con POST hemos visto que debíamos hacerlo con el recurso lista de alumnos (/alumnos), ya que la URI del nuevo alumno todavía no existe. La operación **no es idempotente**, es decir, si añadimos varias veces el mismo alumno aparecerá repetido en nuestra lista de alumnos con URIs distintas.

PUT: Hace que el recurso indicado tome como contenido los datos enviados. El recurso podría no existir previamente, y en caso de que existiese sería sobrescrito con la nueva información. A diferencia de POST, PUT **es idempotente**. Múltiples llamadas idénticas a la misma acción PUT siempre dejarán el recurso en el mismo estado. La acción se realiza sobre la URI concreta que queremos establecer (por ejemplo, /alumnos/Esther), de forma que varias llamadas consecutivas con los mismos datos tendrán el mismo efecto que realizar sólo una de ellas.

DELETE/DELETE

El método DELETE se utiliza para BORRAR representaciones.

La secuencia de pasos necesarios para procesar la petición es la siguiente:

1. Un cliente Java realiza una petición DELETE a la URI http://restfuljava.com/alumnos/Esther
2. El servidor web recibe la petición y delega en el framework REST para que la gestione; nuestro código dentro del framework ejecuta los comandos adecuados para borrar la representación de Esther.
3. Una vez que se ha completado la actualización, se envía una respuesta al cliente. Se podría devolver 200 (Ok) con el recurso borrado como contenido, o simplemente 204 (No Content) para indicar que la operación se ha realizado correctamente sin devolver ningún contenido.

Comunicación sin estado

- Cada petición HTTP debe contener toda la información necesaria para interpretar dicha petición (peticiones completas e independientes). Como resultado, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre peticiones.

- Es necesario incluir dentro del encabezado y cuerpo HTTP la petición, los parámetros, contexto y datos precisos para que el servidor pueda generar la respuesta. Así se garantiza mejorar el rendimiento de los servicios web.

Comunicación sin estado

- Algunas aplicaciones basadas en HTTP utilizan cookies y otros mecanismos para mantener el estado de la sesión, cosa que no es soportada por REST.
- REST exige que el estado sea transformado en estado del recurso y sea mantenido en el cliente. Un servidor no debería guardar el estado de la comunicación de cualquiera de los clientes que se comunican con él más allá de una petición única. La razón es la escalabilidad, el número de clientes que pueden interactuar con el servidor se ven significativamente afectados si fuese necesario mantener el estado del cliente.

Representaciones múltiples

- Permite que cada cliente pueda manejar en qué formato desea que le devuelvan los datos por cada petición al servidor, accede al cliente la capacidad de manejar un determinado formato de datos al solicitarlo, especificándolo en la petición.
- Para esto es necesario utilizar el atributo **HTTP Accept** en el encabezado del mensaje y definir el tipo en el ***content-type***.
- Se utilizan tipos MIME (*Multipurpose Internet Mail Extensions*) como posibles tipos en los cuales se retornará la respuesta en dicho formato.
- Esto permite que los servicios sean consumidos independientemente de lenguajes y plataformas.

Tipos MIME	Formato
application/xml	XML
application/json	JSON
application/xhtml+xml	XHTML

Diseño REST

1. Identificar todas las entidades conceptuales que se desean exponer como servicio y crear una URL para cada recurso. Los recursos deberían ser nombres): http://www.service.com/entities/001, en lugar de, http://www.service.com/entities/getEntity?id=001.
2. Categorizar los recursos conforme a si los clientes pueden obtener un representación del recurso o si pueden modificarlo. Para obtenerlos, hay que hacer los recursos accesibles utilizando un HTTP GET. Para modificarlos, hay que hacer los recursos accesibles mediante HTTP POST, PUT y/o DELETE.
3. Todos los recursos accesibles mediante GET no deberían tener efectos secundarios. Los recursos deberían devolver la representación del recurso y, por tanto, invocar al recurso no debería ser el resultado de modificarlo.

Diseño REST

5. Ninguna representación debería estar aislada. Es recomendable poner hipervínculos dentro de la representación de un recurso para que los clientes puedan obtener más información.
6. Especificar el formato de los datos de respuesta mediante un esquema (DTD, W3C Schema, etc.) Para los servicios que requieran un POST o un PUT es aconsejable también proporcionar un esquema para especificar el formato de la respuesta.
7. Describir como nuestro servicio ha de ser invocado, mediante un documento WSDL/WADL o simplemente HTML.

Ejemplo de diseño

- Sistema de gestión de una lista de empleados.
- Hay dos tipos de recursos, *Employee* y *Allemployee* y dos tipos de URIs.
- Cada recurso de tipo *Employee* debe tener su propia URI con una representación adecuada.
- La colección de recursos es otro recurso con su propia URI.
- Al asignar el formato, hay que tener en cuenta que se trata de la representación.
- No se puede acceder directamente al recurso, hay que obtener una representación de este, pudiendo ser un documento HTML, XML, una imagen u otro fichero.
- Para cada uno de los recursos, se tiene que decidir cuál va a ser su representación.

Ejemplo de diseño

Si el formato de representación es XML , el formato del empleado podría ser el siguiente :

```
<employee xmlns='HTTP://example.org/my-example-ns/'>
    <name>Full name goes here</name>
    <title>Persons title goes here</title>
    <phone-number>Phone number goes here</phone-number>
</employee>
```

Ejemplo de diseño

Para el listado de empleados podría ser:

```
<employee-list xmlns='HTTP://example.org/my-example-ns/'>
  <employee-ref href="URI of the first employee"/>Full name of
  the first employee goes here</employee>
  <employee-ref href="URI of employee #2"/>Full name of the
  employee #2 goes here</employee>
  ..
  <employee-ref href="URI of employee #N"/>Full name of the
  employee #N goes here</employee>
</employee-list>
```

Ejemplo de diseño

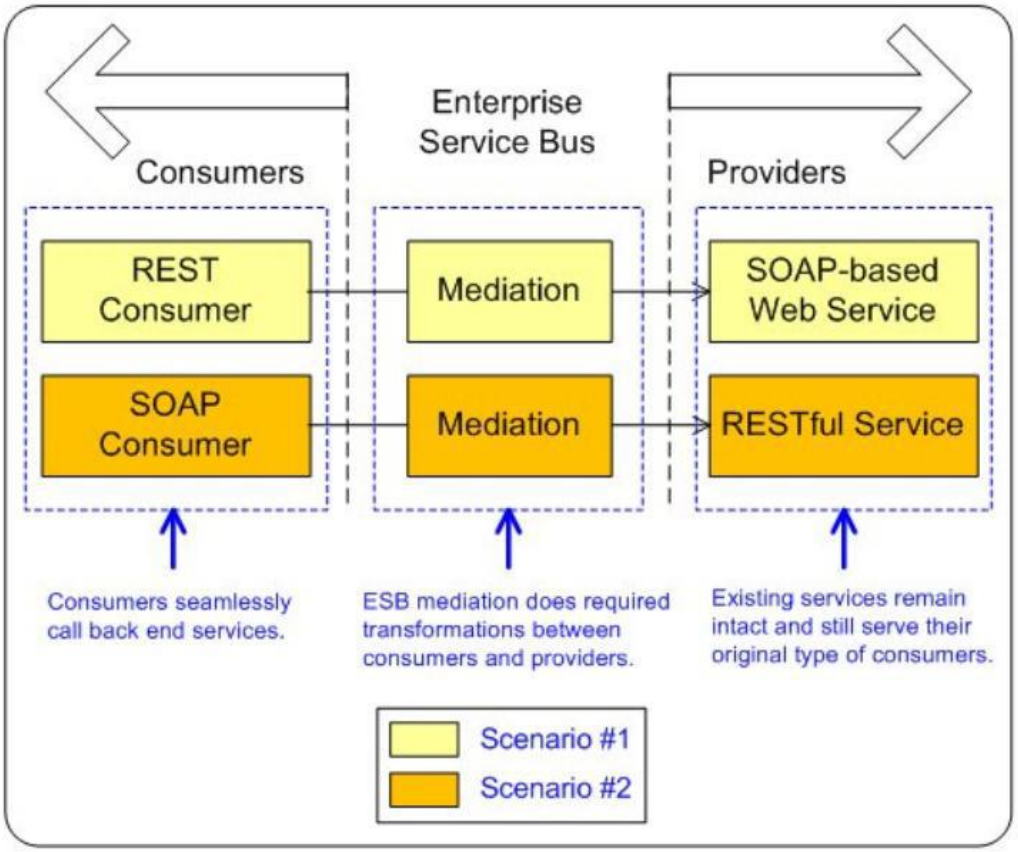
Recurso	Método	Representación	Códigos de estado
Employee	GET	Formato del empleado	200, 301, 410
Employee	PUT	Formato del empleado	200, 301, 400, 410
Employee	DELETE	-	200, 204
All Employees	GET	Formato de la lista de empleados	200, 301
All Employees	POST	Formato de la lista de empleados	201, 400

Características	• **Las operaciones se definen en los mensajes.** • **Una dirección única para cada instancia del proceso.** • **Cada objeto soporta las operaciones estándares definidas.** • **Componentes débilmente acoplados.**
Ventajas	• Bajo consumo de recursos. Las instancias del proceso son creadas explícitamente. • El cliente no necesita información de enrutamiento a partir de la URI inicial. • Los clientes pueden tener una interfaz *listener* genérica para las notificaciones. • Suele ser fácil de construir y adoptar.
Desventajas	• Gran número de objetos. • Manejar el espacio de nombres (URIs) puede ser engorroso. • La descripción sintáctica/semántica muy informal (orientada al usuario). • Pocas herramientas de desarrollo.

Transformación entre servicios

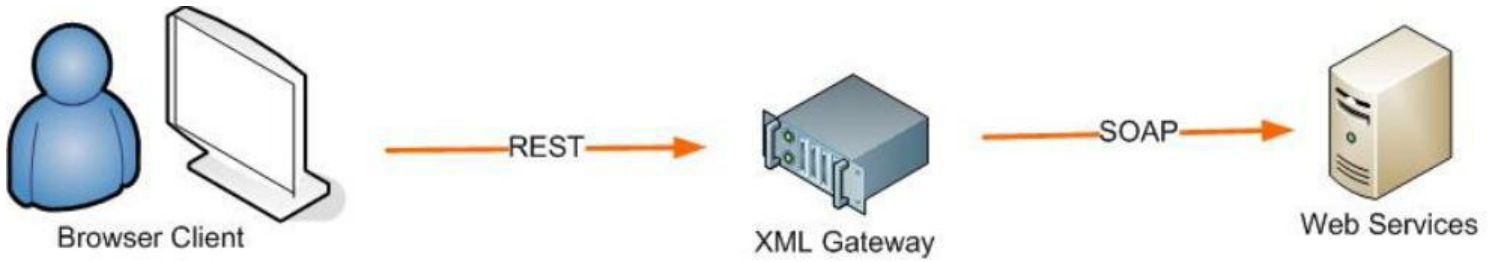
- Debido a que SOA está destinado a la integración de sistemas heterogéneos, es posible que ambos tipos de servicios web tengan que coexistir en la misma organización.
- Existen enfoques que permiten unificar ambas implementaciones de servicios y marcos que hacen posibles la conversión entre ambos tipos de servicios.
- Esta transformación se puede realizar mediante artefactos de mediación ubicados entre clientes y proveedores de información. Los enfoques principales son: buses de servicios o ESB (*Enterprise Service Bus*) y puertas de enlace o *gateways* SOA.

Buses de servicios



Tipología de los ESB (Abbas, 2009)

Puertas de enlace



Arquitectura de la conversión Vordel (O'Neill, 2008)