5 REPRESENTATIONAL STATE TRANSFER: REST

En este capítulo queda recogida una visión actual del estilo de arquitectura REST. La base del capítulo es la [disertación] de Roy Thomas Fielding, aunque también añade algunos detalles del funcionamiento que surgieron posteriormente.

5.1 Introducción

REST son las siglas de ***Representational State Transfer***, un estilo de arquitectura especialmente diseñado para los sistemas distribuidos de hipermedios cuyo término acuñó **Roy Thomas Fielding** en su disertación.

La motivación de desarrollar REST era crear un modelo de arquitectura que describiese como debería funcionar la Web, así como que pudiese servir como marco de trabajo para los estándares de protocolos Web. REST ha sido aplicado para describir la arquitectura Web deseada, ayudar a identificar problemas existentes, comparar soluciones alternativas, y asegurar que el protocolo no viole las restricciones que hacen que la Web funcione correctamente. Este trabajo fue realizado por el Internet Engineering Taskforce (IETF) y el World Wide Web Consortium (W3C), gracias a sus esfuerzos por definir un estándar de arquitectura para la Web: HTTP, URI y HTML.

Desde un punto de vista cronológico, REST ha ido evolucionando. Roy Thomas Fielding lo concibió de una manera abierta (también podríamos llamarla abstracta), pero posteriormente, se le ha ido dando forma para que encaje perfectamente en el mundo Web, usando las tecnologías existentes (URI, HTTP y XML). Como se verá posteriormente, hacer uso de estas tecnologías, puede significar renunciar a parte de su esencia.

REST es un estilo de arquitectura, por lo tanto, es conveniente explicar este concepto:

Un **estilo de arquitectura** es un *conjunto coordinado de restricciones* que controlan el funcionamiento y las características de los elementos de la arquitectura y permiten las relaciones de unos elementos con otros.

Los elementos de una arquitectura son tres: componentes, conectores y datos:

- **Componente**: Es una unidad abstracta de instrucciones software y estados internos que proporciona una transformación de los datos a través de su interfaz.
- **Conector**: Es un mecanismo abstracto que hace posible la comunicación, coordinación y cooperación entre componentes.
- **Dato**: Es un elemento de información que se transfiere desde o hacia un componente a través de un conector

Figura 8: Elementos arquitectura

Durante el desarrollo del capítulo se irá definiendo REST como un estilo de arquitectura, es decir, se estudiarán:

- Las restricciones que propone.
- Los elementos (datos, conectores y componentes).
- Las relaciones entre los elementos.

Una vez definido REST, se estudiará la manera de adaptarlo a las tecnologías Web actuales; HTTP, URI y XML. Para finalizar el capítulo, se verán los cabos sueltos que según algunos autores, deja REST en su definición.

5.2 Origen de REST, Roy Thomas Fielding

Roy Thomas Fielding es toda una autoridad en el mundo de la arquitectura de redes de ordenadores. Nació en Laguna Beach, California (Estados Unidos), y recibió un doctorado de la Universidad de California, Irvine en el año 2000.

Fielding ha estado muy involucrado en el desarrollo de las especificaciones HTTP, HTML y URI. Fue cofundador del proyecto Apache HTTP Server, y es miembro de la comunidad OpenSolaris. Actualmente trabaja como jefe científico en Irving, California.

Su disertación, ***Architectural Styles and the Design of Network-based Software Architectures*** (año 2000), describe **REST** como un principio clave en el mundo Web. Desde el primer momento, captó la atención del público. Parte de esta atención es debida a que la disertación contaba con el apoyo de Tim Berners-Lee.

En este capítulo se va a exponer la parte más importante de la disertación de Fielding. Este documento es la primera referencia a REST que existe. Como ya se ha comentado, Fielding fue el que acuñó el término e *inventó* el estilo de arquitectura REST.

5.3 Objetivos

Cuando Roy T. Fielding creó REST su disertación, perseguía unos objetivos muy concretos que Paul Prescod resumió en un artículo [13]:

- Escalabilidad de los componentes de interacción.
- Generalidad de las interfaces.
- Independencia en el desarrollo de componentes.
- Sistemas intermedios para reducir el tiempo de interacción, mejorar la seguridad, y encapsular los sistemas de herencia.

5.4 Restricciones de REST

Para poder conseguir los objetivos que se han comentado en el apartado anterior, REST define un conjunto de restricciones o características que deben cumplir las arquitecturas de Servicios Web:

1. Cliente Servidor
2. Sin estado
3. Caché
4. Sistema de capas
5. Interfaz Uniforme
6. Sistema de capas
7. Código bajo demanda

A continuación se va a explicar en qué consiste cada restricción.

5.4.1 Cliente-Servidor

La primera restricción que tiene REST está en común con el estilo Cliente-Servidor.

Separar lo que es competencia del cliente y el servidor es el principio de todo. Hay que distinguir lo que concierne al interfaz del usuario del almacenamiento de datos.

De esta manera se ayuda a mejorar la portabilidad de la interfaz de usuario a través de múltiples plataformas. Además también se mejora la escalabilidad porque se simplifican las componentes del servidor al no tener que implementar las funcionalidades que van asociadas a la interfaz del usuario. Otro factor importante es que la separación permite a los componentes desarrollarse independientemente.

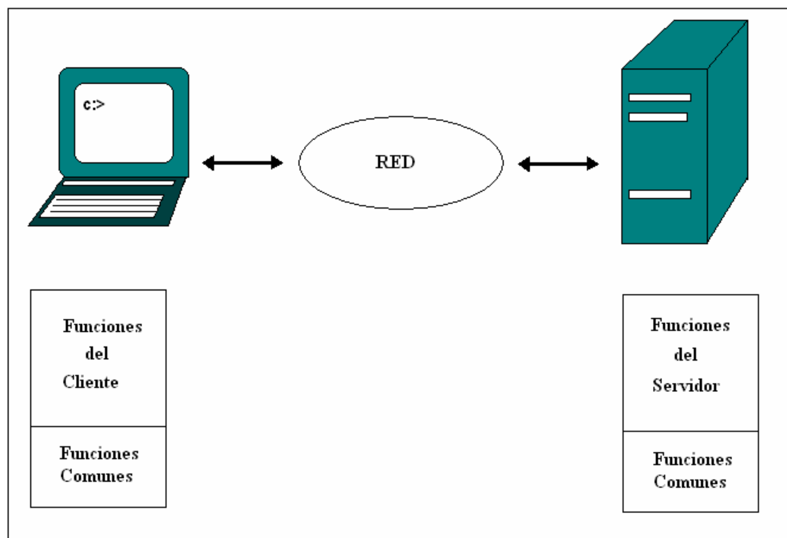

Figura 9: Cliente-Servidor

5.4.2 Sin estado (Stateless)

La segunda restricción está en común con el estilo client-stateless-server.

Cada petición del cliente debe contener toda la información necesaria para que el servidor la comprenda y no necesite mirar ningún dato almacenado previamente sobre el contexto de la comunicación. El estado de la sesión por lo tanto se guarda integramente en el cliente.

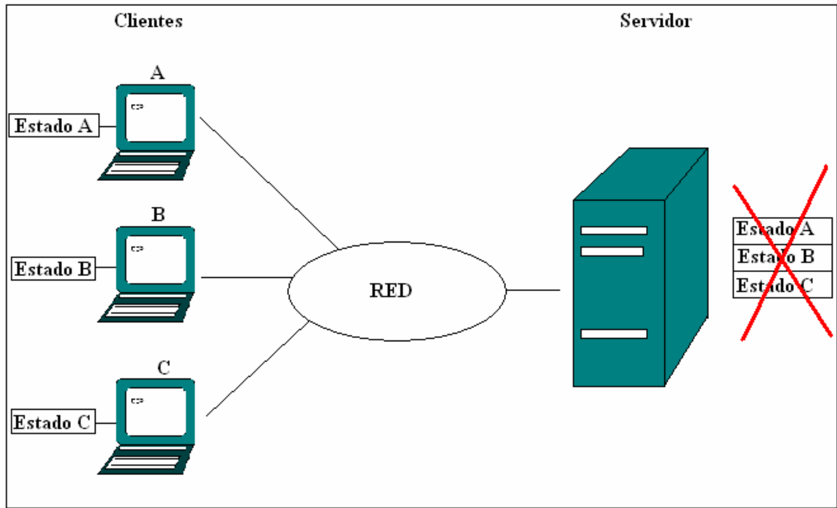

Figura 10: Stateless

Esta restricción mejora la visibilidad, eficiencia y escalabilidad. La visibilidad mejora porque el servidor no tiene que ocuparse de mirar en otros sitios ni realizar más operaciones para comprender la naturaleza de una simple petición. La eficiencia aumenta porque se hace más fácil recuperarse de errores parciales. La escalabilidad se

ve también afectada porque al no hacer falta almacenar los estados entre las peticiones, los componentes pueden liberar recursos más rápidamente. Existe una simplificación de la implementación de aplicaciones al no existir gestión de recursos entre las peticiones.

La desventaja de esta restricción es que puede empeorar el funcionamiento de la red porque incrementa el tráfico de datos repetidos al enviar una serie de peticiones. Esto ocurre porque los datos no pueden quedarse almacenados en el servidor identificando un contexto determinado de comunicación. Además poniendo el estado de la aplicación en el lado del cliente, se reduce el control para obtener un comportamiento consistente en la aplicación. La aplicación se hace muy dependiente de una correcta implementación de la semántica en las distintas versiones del cliente.

5.4.3 Caché

Esta tercera restricción la comparte REST con el estilo Client-Cachestateless-Server Style.

Las respuestas a una petición deben poder ser etiquetadas como *cacheable* o *no-cacheable*. Si una respuesta es cacheable, entonces al cliente cache se le da permiso para reutilizar la respuesta más tarde si se hace una petición equivalente.

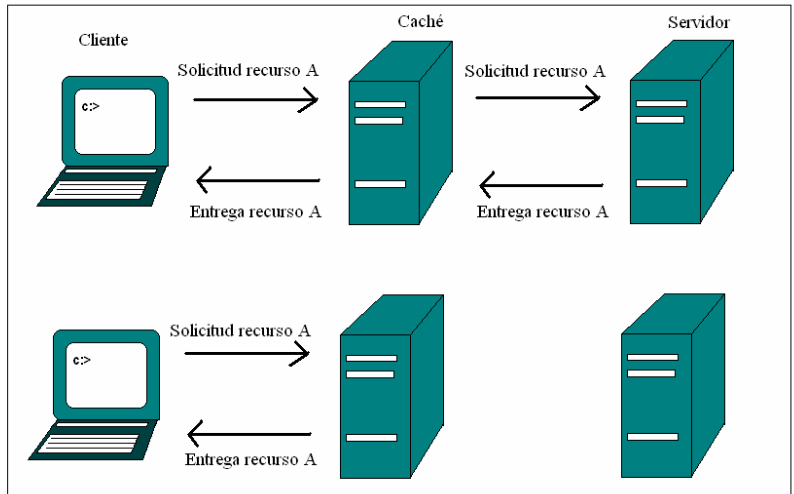

Figura 11: Caché

La ventaja de añadir esta restricción es que se evitarán determinadas peticiones al servidor, mejorando así la eficiencia y escalabilidad. Se va a reducir el tiempo medio de espera de una serie de interacciones.

La desventaja de esta restricción es que puede inducir a un mal funcionamiento de una aplicación si los datos obtenidos del caché difieren de los que se hubiesen obtenido realizando la petición directamente al servidor.

5.4.4 Interfaz uniforme

Las siguientes restricciones que se van a comentar a partir de ahora son realmente las novedades que pretende aportar REST a las nuevas arquitecturas Web.

La principal característica que distingue a REST del resto de estilos de arquitecturas de red es el énfasis de usar una interfaz uniforme entre los componentes. Aplicando los principios de generalidad de la ingeniera del software a los componentes de la interfaz, se simplifica la arquitectura del sistema global y la visibilidad de interacciones se mejora. Las implementaciones se separan de los servicios que proporcionan, lo que anima al desarrollo independiente.

La desventaja de usar una interfaz uniforme es que degrada la eficiencia porque la información transferida está en una forma estandarizada y no según las necesidades que tenga la aplicación. El interfaz de REST está diseñado para ser eficiente con transferencias de datos de hipermedios (suelen ser datos voluminosos). Con esta decisión, está optimizado para la mayor parte de la Web pero no siendo así para otras formas de arquitectura de interacción. Para obtener una interfaz uniforme, REST define cuatro restricciones de interfaz:

- Identificación de recursos
- Manipulación de recursos a través de sus representaciones
- Mensajes auto-descriptivos
- Hipermedios como el motor del estado de la aplicación.

5.4.5 Sistema de capas

Para poder mejorar el comportamiento de la escalabilidad en Internet, se añade la restricción del sistema de capas. Este sistema permite tener una arquitectura compuesta por capas jerárquicas, limitando el comportamiento de los componentes porque no pueden "ver" mas allá de la capa con la que está interactuando.

Usando el sistema de capas se pueden simplificar los componentes moviendo las funcionalidades de uso infrecuente hacia sistemas intermedios compartidos. Estos sistemas pueden usarse para mejorar la escalabilidad permitiendo un balanceo de la carga de los servicios a través de múltiples redes y procesadores.

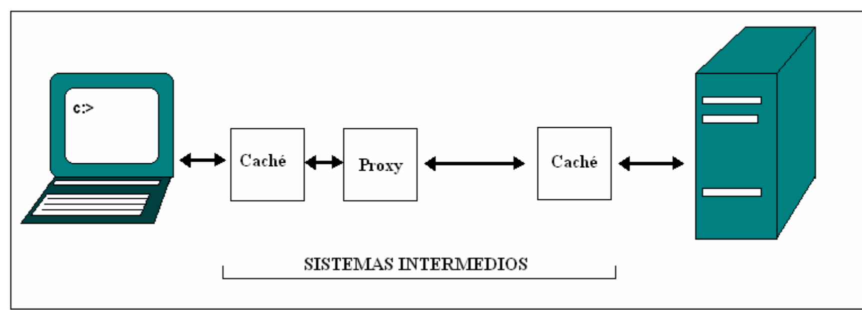

Figura 12: Sistema de capas

La principal desventaja de los sistemas de capas es que añaden cabeceras y retrasos al procesamiento de datos. Para un sistema de red que soporta la restricción de caché, este efecto puede verse reducido por los beneficios de usar un caché compartido en los sistemas intermedios. Colocando caches compartidos en los límites de un dominio organizativo, puede conseguirse una mejora significativa. Las capas además permiten directivas de seguridad para los datos que cruzan los límites organizativos, algo que necesitan los firewalls.

La combinación de interfaz uniforme y sistema de capas tiene unas características arquitectónicas similares a las del estilo pipe-and-filter. Aunque la interacción de REST es en dos sentidos, los flujos de datos de gran volumen de la interacción de los hipermedios, pueden ser procesados como flujos de datos de red. Usando REST, los sistemas intermedios pueden transformar el contexto de los mensajes porque estos son auto-descriptivos, su semántica es visible.

5.4.6 Código bajo demanda

La última restricción es opcional, consiste en permitir a los clientes tener la funcionalidad de descargar y ejecutar código en forma de applets y scripts. Esto simplifica el lado del cliente porque reduce el número de funcionalidades que tiene que tener implementadas al crearse. Las funcionalidades se pueden descargar posteriormente aumentando así la extensibilidad del sistema.

La principal desventaja de esta restricción es que reduce la visibilidad y puede influir en la seguridad del sistema. Sin embargo, tiene un propósito en el diseño arquitectónico de un sistema que abarque límites de organización múltiples. Con esta restricción, la arquitectura gana solamente una ventaja y sufre el resto de desventajas, por ello al ser una restricción opcional, en los contextos en los que el código bajo demanda no sea útil ni necesario, lo mejor será no incluirlo porque puede acarrear más problemas que beneficios.

5.5 Elementos arquitectónicos de REST

Hasta ahora se han visto las restricciones que usa REST para controlar el funcionamiento y las relaciones entre los distintos elementos (componentes, conectores y datos). Ahora se va a ver como concibió R. T. Fielding cada uno de estos elementos para que encajasen dentro de REST.

5.5.1 Datos en REST

A diferencia de los estilos basados en objetos distribuidos, donde todos los datos se encapsulan y son ocultados por los componentes del proceso, la naturaleza y el estado de los datos son un aspecto clave para REST. La forma racional de diseñar los datos

puede verse en la naturaleza de los hipermedios distribuidos. Cuando se selecciona un link, la información debe trasladarse desde la localización donde está almacenado hasta donde será usado (la mayor parte de las veces por un humano). Esto va en contra de otros procesos distribuidos donde es posible, y a veces más eficiente, mover el "proceso agente" (procedimiento almacenado, expresión de búsqueda, etc) hasta los datos en vez de mover los datos hasta el proceso.

Un arquitecto de hipermedios distribuidos solo tiene tres opciones:

1) Renderizar los datos desde donde están localizados y enviar una imagen de formato fijo al receptor.

2) Encapsular los datos con una herramienta de renderización y enviar ambas cosas al receptor.

3) Enviar los datos en bruto al receptor junto con el metadato que describe el tipo de datos, para que el receptor pueda elegir su propia herramienta de renderización.

Cada opción tiene sus propias ventajas y desventajas:

	OPCIÓN 1	OPCIÓN 2	OPCIÓN 3
VENTAJAS	Permite que toda la información sobre la verdadera naturaleza de los datos siga oculta dentro del remitente. Hace que la implementación del cliente sea más sencilla.	Posee una gran flexibilidad	Permite al emisor permanecer simple y escalable, mientras que minimiza el volumen de datos transferidos
INCONVENIENTES	Restringe la funcionalidad del receptor y sitúa la mayor parte de la carga de procesamiento en el servidor, recayendo en problemas de escalabilidad.	Limita la funcionalidad del receptor y puede incrementar en gran manera el volumen de datos transferidos.	Pierde las ventajas de poder ocultar la información y requiere que el emisor y receptor entiendan los mismos tipos de datos.

Tabla 3 Tipos de diseño de datos

REST propone un híbrido de los tres puntos de vista centrándose en comprender los tipos de datos según los metadatos, pero limitándose a lo que es relevante para un interfaz estandarizado. Los componentes de REST se comunican trasfiriendo **representaciones de un recurso** en un formato que se ajusta a un conjunto de tipos de datos estándares. Estos tipos serán seleccionados dinámicamente basándose en las capacidades o deseos del receptor y en la naturaleza del recurso. Si la representación está en el mismo formato que el recurso en bruto, o si difiere, es algo que permanece oculto detrás de la interfaz. Los beneficios de la opción 2 son poder enviar una representación que consiste en instrucciones en el formato de datos estándar de la herramienta de datos encapsulada. REST gana en la separación de las funcionalidades del estilo cliente-servidor, pero permitiendo ocultar la información a través de un interfaz genérico que permite encapsular, evolucionar los servicios y proporciona un juego de funcionalidades amplio que podrá ser descargado.

5.5.1.1 Recursos e identificadores de recursos

La clave de la abstracción de la información en REST es el recurso. Cualquier información que pueda ser nombrada, puede ser un recurso, por ejemplo un documento, una imagen, un servicio temporal, una colección de otros recursos… En otras palabras, cualquier concepto que pueda ser llamado mediante una referencia en hipertexto, encaja en la definición de recurso. La definición exacta es, un recurso R es un miembro de la función $M_R(t)$, donde un tiempo t mapea un conjunto de entidades o valores que son equivalentes. Los valores del conjunto deben ser representaciones del recurso o identificadores del recurso.

Cuando un usuario desea realizar una actividad remota, no accederá a un Servicio directamente, sino que accederá a un **recurso**.

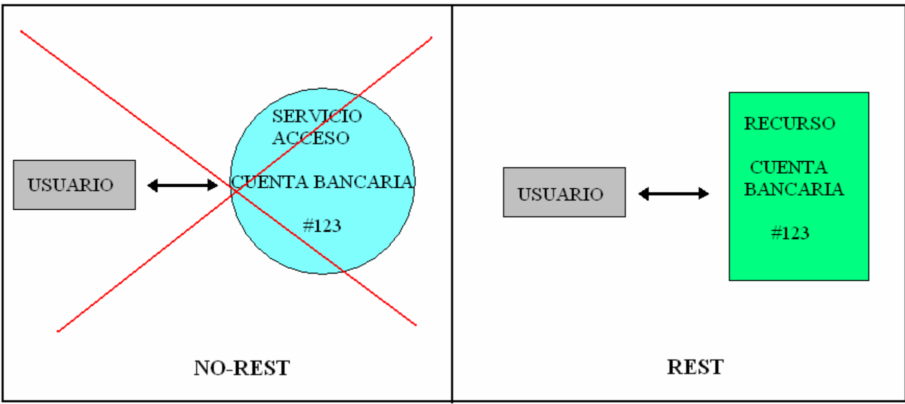

Figura 13: Funcionamiento REST

En el ejemplo de la Figura [14], un usuario quiere conocer remotamente al saldo que tiene en su cuenta bancaria. Según el estilo **REST**, para realizar esta acción, el usuario no accederá directamente a un método que le proporcione la información de la cuenta,

sino que accederá a un **recurso** que contiene el saldo de su propia cuenta bancaria. Como vemos, REST tiene una visión de los Servicios Web en la que **sustituye métodos** remotos por acceso a **recursos** remotos.

Algunos recursos son estáticos en el sentido de que cuando son examinados, se corresponden con el mismo conjunto de valores independientemente del tiempo. Otros varían en el tiempo.

Los recursos proporcionan generalidad abarcando muchas fuentes de información sin tener que distinguir artificialmente el tipo o la implementación. Además permiten unir la referencia a una representación porque puede negociarse el contenido al hacer una petición. Finalmente un autor puede referenciar un concepto en vez de a una representación en particular del concepto. De esta manera se elimina la necesidad de cambiar todos los links existentes siempre que la representación cambie (suponiendo que el autor use el identificador correcto).

REST usa un **identificador de recurso** para identificar un recurso en particular involucrado en una interacción entre componentes. Los conectores de REST proporcionan una interfaz genérica para acceder y manipular los valores de un conjunto de recursos independientemente del software que maneja la petición. La autoridad que asigna los identificadores de recursos, haciendo posible la referencia al recurso, es la responsable de validar la consistencia de la semántica al mapear a lo largo del tiempo.

Los sistemas tradicionales de hipertexto, que normalmente operan en un entorno local o cerrado, usan nodos únicos o identificadores de documento que cambian cada vez que la información cambia, confiando en que los servidores de liks mantengan las referencias separadas del contenido. Como los sistemas de link centralizados no son compatibles con la escalabilidad y los dominios multi-organizativos de la Web, REST apuesta porque sea el autor el que elija el identificador del recurso que mejor se ajuste a la naturaleza del concepto que será identificado. Naturalmente, la calidad del identificador es proporcional a la cantidad de dinero invertida para mantener la validez mientras que la información se mueve o desaparece a lo largo del tiempo.

5.5.1.2 Representación

Los componentes de REST realizan acciones en un recurso usando una **representación** que capture el estado actual o previsto de un recurso y transfiriendo la representación entre los componentes. Una representación es una secuencia de bytes más un metadato de representación que describe esos bytes. Otros nombres más comunes pero menos precisos para una representación incluye los casos: documento, archivo, y entidad de mensaje HTTP.

Lo que un cliente puede solicitar a un Servidor no es un **recurso**, es la ***representación o la modificación de un recurso***. Un recurso es un término abstracto que define a la información a la que desea acceder el cliente, pero la representación es la información tangible que le llega. Por ejemplo, si desde un navegador Web se solicita ver el contenido de una página (recurso,abstracto), se estará solicitando una **representación de un recurso**, que en este caso será un documento HTML (representación, concreta).

Una imagen que esté incluida en ese documento HTML, es la representación de otro recurso diferente. Si se desea modificar algo remotamente, se estará **modificando un recurso**.

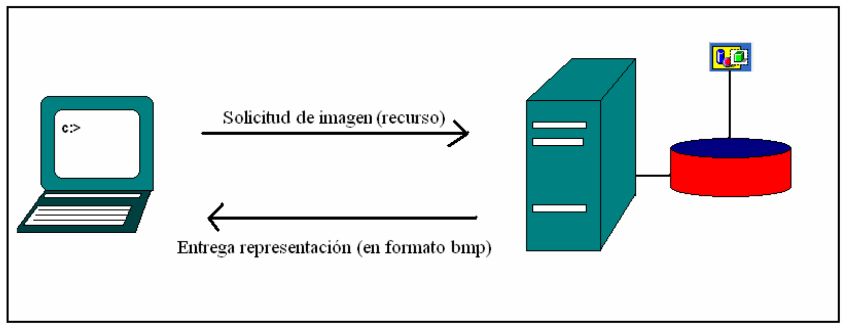

Figura 14: Representación de recurso

Una representación consiste en datos, metadatos que describen los datos y en ocasiones, metadatos que describen a los metadatos (normalmente con propósito de verificar la integridad de un mensaje). Los metadatos suelen venir como parejas nombre-valor, donde el nombre corresponde a un estándar que define la estructura del valor y la semántica. Los mensajes de respuesta pueden incluir metadatos de representación y metadatos de recurso (información sobre el recurso que no es específico a la representación provista).

El control de datos define el propósito de los mensajes entre componentes, como pueden ser la acción solicitada o el significado de una respuesta. También se usa para parametrizar peticiones e invalidar el comportamiento que se tiene por defecto al conectar varios elementos. Por ejemplo, se puede modificar el comportamiento del caché controlando los datos del mensaje de petición o el de respuesta.

Dependiendo del control de datos del mensaje, una representación puede indicar el estado actual del recurso solicitado, el estado deseado o el valor de otro recurso cualquiera. Algunos casos concretos pueden ser la representación de los datos de entrada dentro de la petición de un cliente o la representación de las condiciones de error de una respuesta. Por ejemplo, para añadir un recurso desde una localización remota, es necesario que el autor envíe una representación al servidor, así se podrán establecer los valores del recurso para posteriores peticiones. Si el conjunto de valores de un recurso en un momento dado, consiste en múltiples representaciones, el proceso de negociación del contenido puede ser usado para elegir la mejor representación.

El formato de los datos de una representación es conocido como tipo de medio (media type). Una representación puede ser incluida en un mensaje y procesada por el receptor de acuerdo con el control de datos del mensaje y la naturaleza del tipo de medio. Algunos tipos de medios están previstos para procesos automatizados, otros para ser renderizados para que los usuarios los vean y otros son capaces de englobar a estos dos. Los tipos de medios compuestos se pueden utilizar para incluir representaciones múltiples en un solo mensaje.

El diseño de un tipo de medio puede afectar directamente a la percepción que tiene un usuario de los sistemas de hipermedios distribuidos. Cualquier dato que deba ser recibido antes de que el receptor pueda empezar a renderizar la representación, añade retraso a la interacción. Un formato de datos que contiene la parte más importante de la información de renderización al principio del mensaje, puede ser renderizado mientras el resto de la información está siendo recibida. Esto mejora notablemente la percepción del usuario al contrario que si tuviésemos que recibir todo el mensaje para empezar a renderizar.

Por ejemplo, un navegador Web que pueda renderizar un gran documento HTML mientras está siendo recibido, produce que el usuario tenga una mejor percepción del funcionamiento que si tuviese que esperar a recibir completamente el documento para ver algo en el navegador. Esto ocurre aunque el comportamiento de la red en los dos casos sea el mismo. Hay que darse cuenta de que la habilidad de renderizar y la capacidad de representación también pueden verse afectada por la elección del contenido. Si las dimensiones de tablas con tamaño dinámico y los objetos que transportan deben ser determinadas antes de que puedan ser renderizadas, se sufrirá un retraso a la hora de ver una página de hipermedios.

Para terminar este apartado sobre los datos en REST se incluye la Tabla [3] que es un resumen de los datos usados por REST.

ELEMENTO DATO	EJEMPLO DE LA WEB
Recurso	El objetivo conceptual de una referencia de hipertexto
Identificador de recurso	URI
Representación	Documento HTML, imagen JPEG
Metadato de representación	Tipo de medio
Recurso metadato	Source link
Datos de control	Cache-Control

Tabla 4 Elementos de Datos REST

5.5.2 Conectores en REST

REST usa varios tipos de conectores para encapsular las actividades de acceso a recursos y transferir las representaciones de los recursos, como son los conectores ***Cliente, Servidor, Caché, Resolver y Túnel***.

Los conectores presentan un interfaz abstracto para los componentes de comunicación, realzando la simplicidad que proporciona una separación de las competencias y ocultando las implementaciones subyacentes de los recursos y mecanismos de comunicación. La generalidad de la interfaz permite la sustitución: si los usuarios solo acceden al sistema a través de un interfaz abstracto, la implementación puede ser reemplazada sin que el usuario se de cuenta de ello. Como un conector gestiona la

comunicación de red de los componentes, la información puede ser compartida por múltiples interacciones para potenciar la eficiencia.

Todas las interacciones de REST son sin estado (stateless). Cada petición contiene toda la información necesaria que necesita un conector para poder comprenderla, independientemente de las peticiones que hayan sido procesadas con anterioridad. Esta restricción conlleva cuatro funciones:

1. Elimina cualquier necesidad por parte de los conectores de retener el estado de la aplicación entre dos peticiones, reduciendo el consumo de recursos físicos y mejorando la escalabilidad.

2. Permite que las interacciones sean procesadas en paralelo sin que por ello requieran que el mecanismo de procesamiento comprenda la semántica de la interacción.

3. Permite a un sistema intermedio ver y comprender una petición de forma aislada, que es algo que puede ser necesario cuando los servicios cambian dinámicamente.

4. Fuerza a que esté presente en cada petición toda la información que influya para determinar si se puede usar una respuesta *cacheable*.

La interfaz del conector es similar a una invocación a proceso, pero con diferencias en el paso de parámetros y resultados.

- Los parámetros de entrada consisten en la petición del control de datos, de un identificador de recurso que indica el objetivo de la petición y una representación opcional.

- Los parámetros de salida consisten en responder con el control de los datos, un metadato de recurso opcional y una representación opcional.

Desde un punto de vista abstracto, la invocación es síncrona, pero los parámetros de entrada y de salida pueden ser pasados como flujos (streams) de datos. En otras palabras, el procesamiento puede ser invocado antes de que el valor de los parámetros se conozca totalmente. Así se evita el retardo de las grandes transferencias de datos del procesamiento por lotes.

Los principales tipos de conectores son **Cliente, Servidor y Caché** (los conectores Resolver y Túnel pasan a un segundo plano):

- **Conector Cliente**: Es el que inicia la comunicación realizando una petición.

- **Conector Servidor**: Es el que está escuchando las conexiones y responde a las peticiones para proporcionar acceso a sus servicios. Un componente puede incluir conectores de cliente y de servidor.

- **Conector Caché**: Puede estar alojado en la interfaz de un conector cliente o servidor para salvar las respuestas *cacheables* de las interacciones en curso para usarlas en interacciones posteriores. Un caché puede ser usado por un cliente para evitar la repetición de la comunicación de red, o por un servidor para evitar repetir el proceso de generación de una respuesta. En ambos casos sirve para reducir retraso de la interacción. Un caché se implementa normalmente dentro del espacio de direcciones del conector que lo usa.

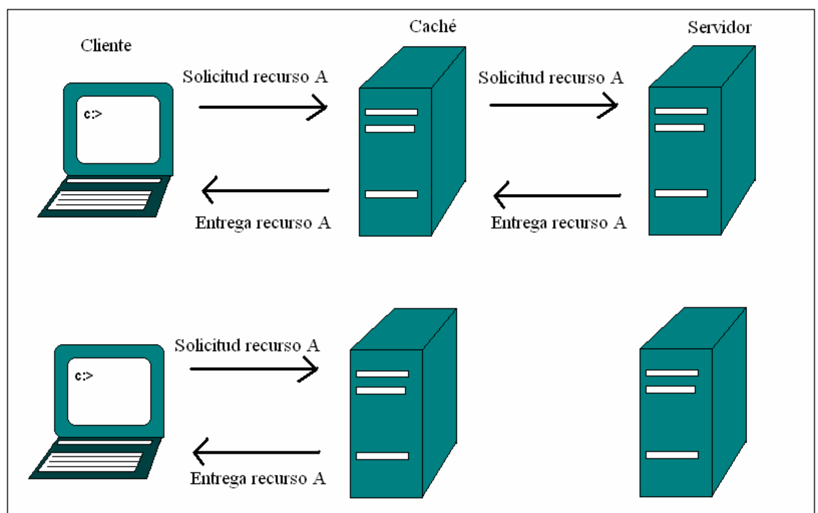

Figura 15: Mecanismo caché

Como puede verse en la Figura [16], el cliente solicita al servidor el recurso A. Si este recurso está definido como *cacheable*, el **servidor caché** almacenará la representación del recurso que se entrega. La siguiente vez que se solicite el recurso A, la petición no llegará más allá del servidor caché. Si la respuesta hubiese sido etiquetada como *no-cacheable* la petición habría sido redirigida de nuevo hasta el servidor.

Algunos conectores de caché son compartidos, esto significa que una respuesta puede ser usada para responder a otro cliente distinto del que originó la respuesta almacenada en el caché. Compartir caché es efectivo reduciendo el impacto en aquellos servidores populares a los que los clientes acceden para obtener la misma información. Es particularmente efectivo cuando el caché se prepara jerárquicamente para cubrir a grandes grupos de usuarios, como es el caso de la Intranet de una compañía, los clientes de un proveedor de servicios de Internet o las Universidades. Sin embargo, el caché compartido también puede conllevar errores si una respuesta no es la misma consultando el caché que realizando una nueva petición.

REST pretende equilibrar la transparencia en el comportamiento del caché con el uso eficiente de la red, antes de asumir que la transparencia absoluta es siempre necesaria.

El caché puede determinar si una respuesta es *cacheable* porque la interfaz es genérica y no específica para cada recurso. Por defecto, una respuesta a una petición repetida, es *cacheable* y la respuesta a otra petición no lo es. Si de alguna manera la autenticación del usuario forma parte de la petición o si la respuesta indica que no puede ser compartida, entonces la respuesta sólo será *cacheable* para un caché no-compartido. Un

componente puede modificar este comportamiento incluyendo el control de datos que puede marcar una interacción como *cacheable*, *no-cacheable*, o *cacheable* por un periodo de tiempo.

Un **Conector Resolver** traduce parcial o completamente el identificador de un recurso en una dirección de red, esto es necesario para establecer conexiones entre componentes. Por ejemplo, la mayoría de las URI incluyen un servidor de nombres DNS como mecanismo para identificar el recurso. Para iniciar una petición, un navegador Web conseguirá el hostname de la URI y usará un *resolver* DNS para obtener la dirección de protocolo de Internet. Otro ejemplo es que algunos esquemas de identificación necesitan un sistema intermedio para traducir un identificador permanente en una dirección más transitoria para acceder al identificador de recurso.

Un último tipo de conector es el **Conector Túnel**, que simplemente permite la comunicación a través de los límites de la conexión, como pueden ser firewalls o puertas de acceso de bajo nivel. La única razón por la cual se modela como una parte de REST y no como algo abstracto que forma parte de la red, es que algunos componentes de REST pueden cambiar dinámicamente de tener un comportamiento dinámico a comportarse como un túnel. El principal ejemplo es un proxy HTTP que cambia a ser un túnel en respuesta a una petición del método CONNECT, permitiendo así al cliente comunicarse directamente con un servidor remoto usando un protocolo diferente como TLS (los proxies no permiten su uso). El túnel desaparece cuando los extremos terminan la comunicación.

Para terminar este apartado sobre los conectores en REST se incluye la Tabla [4] que es un resumen de los conectores usados por REST.

CONECTOR	EJEMPLO DE LA WEB
Cliente	libwww, libwww-perl
Servidor	Libwww, Apache API, NSAPI
Cache	Cache del navegador
Resolver	Bind (DNS lookup library)
Túnel	SOCKS, SSL tras HTTP CONNECT

Tabla 5 Conectores REST

5.5.3 Componentes en REST

Los componentes que tiene REST son: **Agente Usuario, Servidor, Gateway (puerta de acceso) y Proxy**.

Un **Agente Usuario** usa un conector cliente para iniciar una petición y ser el verdadero receptor de la respuesta. El ejemplo más común es el navegador Web, proporciona

acceso a los servicios de información y renderiza las respuestas del servicio de acuerdo con las necesidades de la aplicación.

El **Servidor** usa un conector de servidor para gobernar el espacio de nombres cuando se le solicita un recurso. Es el que realiza la representación de sus recursos y debe ser el receptor último de todas las peticiones que soliciten modificar los valores de sus recursos. Cada servidor proporciona una interfaz genérica para sus servicios como una jerarquía de recursos. Los detalles de implementación de los recursos se ocultan detrás de la interfaz.

Los componentes de los sistemas intermedios actúan a la vez como cliente y como servidor para llevar a sus destinos las posibles transacciones, peticiones y respuestas. Un componente **Proxy** es un sistema intermedio seleccionado por el cliente para proporcionar la encapsulación de la interfaz de otros servicios, transporte de datos, perfeccionamiento del funcionamiento o seguridad. Los componentes de un **Gateway** son un sistema intermedio impuesto por la red o el servidor para proporcionar una encapsulación del interfaz de otros servicios, transmisión de datos, perfeccionamiento del funcionamiento o seguridad. La diferencia entre proxy y puerta de acceso es que el cliente es el que determina cuando va a usar un proxy.

A continuación se muestra la Tabla [5] que es una gráfica resumen de los componentes REST:

COMPONENTE	EJEMPLO DE LA WEB
Servidor Origen	Apache httpd, Microsoft IIS
Gateway	Squid, CGI, Reserve Proxy
Proxy	CERN Proxy, Netscape Proxy, Gauntlet
Usuario Agente	Netscape Navigator, Linxs, MOMspider

Tabla 6 Componentes REST

5.6 Vistas de la arquitectura REST

Ahora que se han analizado los elementos de REST por separado, puede hacerse uso de las **vistas** de la arquitectura para describir las **relaciones que existen entre los elementos**, es decir, observar como trabajan los elementos de manera conjunta. Se usarán tres tipos de vista: ***Procesos, Conectores** y **Datos***.

5.6.1 Vista de proceso

La vista de proceso de una arquitectura es principalmente efectiva para analizar las relaciones entre los componentes desde el punto de vista del camino que siguen los datos a través del sistema. Desafortunadamente la interacción de un sistema real

normalmente involucra un gran número de componentes por lo que puede resultar un tanto caótico.

La siguiente Figura [17], va a resultar útil para observar el camino que siguen las peticiones desde que el cliente las realiza. Se muestra un ejemplo de vista de proceso de una arquitectura basada en REST para un instante determinado durante el procesamiento de tres peticiones en independientes en paralelo.

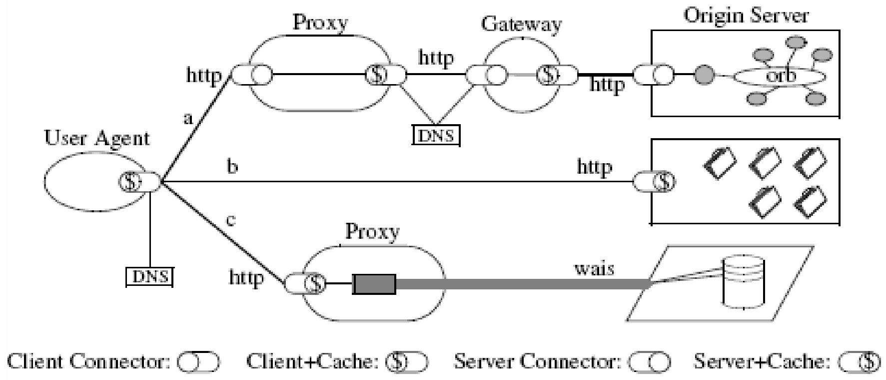

Figura 16: Vista proceso

Se pueden observar tres interacciones diferentas: a, b y c. Cada una representa una petición que realiza el cliente para conseguir un recurso en un servidor remoto diferente. Las interacciones no fueron completadas por el conector de caché que se encuentra en el lado del cliente, por tanto, las peticiones han tenido que ser redirigidas al servidor remoto de acuerdo con el identificador de recurso y la configuración del conector cliente.

- La petición **a)** ha sido redirigida a un proxy local que accede a una puerta de acceso que ha encontrado con una búsqueda al DNS, posteriormente manda la petición a un servidor cuyos recursos están definidos usando una arquitectura de petición de objetos encapsulados.

- La petición **b)** se envía directamente a un servidor que puede llevarla a cabo usando su propio caché. Como ya se comentó anteriormente el caché de cliente falló al intentar este proceso, pero el servidor posee otro caché propio que si ha podido responder.

- La petición **c)** se envía a un proxy que puede usar el acceso WAIS, un servicio de información separado de la arquitectura Web, que traduce la respuesta WAIS a un formato reconocido por el interfaz de conector genérico. Cada componente sólo se entera de la interacción con el conector de su propio cliente o del servidor.

La separación de las tareas de las que se preocupan el cliente y el servidor simplifica la implementación de los componentes, reduce la complejidad de la semántica del conector, aumenta la efectividad del funcionamiento y mejora la escalabilidad de los

componentes del servidor. La restricción del sistema de capas permite a los sistemas intermedios (proxies, puertas de acceso y firewalls) introducirse en varios puntos de la comunicación sin tener que cambiar los interfaces entre los componentes, permitiendo así ayudar a la comunicación y mejorar el funcionamiento de cara a la escalabilidad.

REST permite los sistemas intermedios obligando a que los **mensajes** sean **auto-descriptivos** (la interacción es sin estado entre las peticiones). Los métodos estándar y los tipos de medio se usan para indicar la semántica y el intercambio de información y las respuestas indican si son *cacheables* o no de una manera explícita.

Cuando los componentes se conectan dinámicamente, su disposición y función ante acciones de una aplicación determinada tiene características similares al estilo pipe-and-filter. Aunque los componentes de REST se comunican de manera bidireccional, el procesamiento de cada dirección es independiente. El interfaz de conector genérico permite a los componentes tomar parte en las propiedades de cada petición o respuesta.

Los servicios pueden ser implementados usando una compleja jerarquía de sistemas intermedios y servidores distribuidos. La naturaleza stateless de REST permite a cada interacción ser independiente de las demás, eliminando la necesidad de que toda la topología de componentes se entere de todo, que es una tarea imposible para una arquitectura basada en Internet. Además permite a los componentes actuar como destinos o sistemas intermedios, determinándolo dinámicamente según el objetivo de cada petición. Los conectores solo necesitan conocer la existencia de otros conectores durante la comunicación, aunque pueden conocer la existencia y capacidades de otros componentes por razones de funcionamiento.

5.6.2 Vista de conector

La vista de conector de una arquitectura se centra en los mecanismos de comunicación entre componentes. Para una arquitectura basada en REST, son particularmente interesantes las restricciones que definen la **interfaz genérica**.

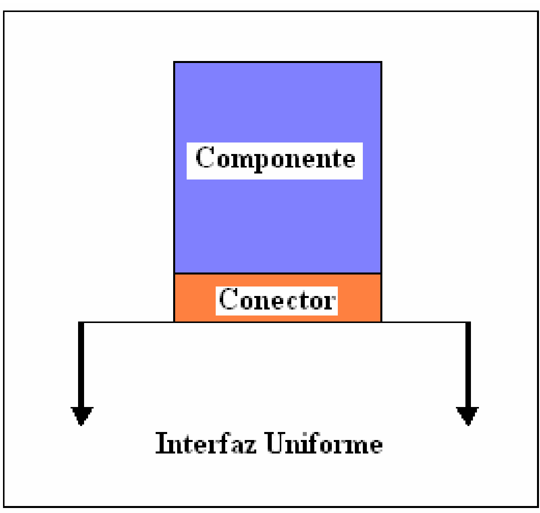

Figura 17: Vista conector

El conector del cliente examina el **identificador de recurso** para seleccionar el mecanismo de comunicación apropiado para cada petición. Por ejemplo, un cliente puede configurarse para conectarse a un componente proxy específico, o para que rechace las peticiones a un determinado subconjunto de identificadores.

REST no restringe la comunicación para que use un protocolo en particular, pero **restringe el interfaz entre los componentes**, y por lo tanto el alcance de la interacción y de la implementación. Por ejemplo, el principal protocolo de transporte es HTTP, pero la arquitectura también incluye acceso a recursos que hay en los servidores de red como puede ser FTP, Gopher y WAIS. La interacción con esos servicios está restringida a la semántica de los conectores de REST. Esta restricción sacrifica algunas de las ventajas de otras arquitecturas como por ejemplo la interacción con estado que usa WAIS, para obtener las ventajas de una interfaz simple y genérica de semántica de conectores. Las interfaces genéricas hacen posible acceder a muchos servicios a través de un simple proxy. Si una aplicación necesita las capacidades de otra arquitectura, puede implementarlas e invocarlas como si fuera un sistema separado corriendo en paralelo, es un caso parecido a como la arquitectura Web tiene un interfaz con los recursos “telnet” y “mailto”.

5.6.3 Vista de datos

La vista de datos de una arquitectura nos revela el estado de la aplicación como un flujo de información a través de los componentes.

REST está concebido para los sistemas de información distribuidos, ve una aplicación como una estructura cohesiva de información y alternativas de control a través de la que un usuario puede realizar una tarea. Por ejemplo, buscar una palabra en un diccionario on-line es una aplicación, igual que realizar una visita a un museo virtual o estudiar unos apuntes para un examen. Cada aplicación define sus objetivos para los sistemas subyacentes.

La interacción de los componentes ocurre en forma de mensajes de tamaño dinámico. Los mensajes pequeños o medianos se usan para el control de la semántica, pero la mayor parte de la aplicación trabaja con grandes mensajes que contienen una representación completa del recurso. La forma más frecuente de semántica para peticiones es solicitar la representación de un recurso, (como por ejemplo el método “GET” de HTTP), que puede ser almacenado en caché para un uso posterior.

REST concentra todo el control de estado en la representación recibida como respuesta a las interacciones. El objetivo es mejorar la escalabilidad del servidor eliminando cualquier necesidad de tener al servidor enterado del estado del cliente, sólo debe saber de la petición actual. El estado de una aplicación está por lo tanto definido por sus peticiones pendientes, la topología de los componentes conectados, las peticiones activas en esos conectores, el flujo de datos de representación en respuesta a esas peticiones y el procesamiento de las representaciones cuando son recibidas por el agente usuario.

Una aplicación alcanza un **estado estacionario** siempre que *no tenga peticiones pendientes*, por ejemplo si no tiene peticiones pendientes y todas las respuestas al conjunto de peticiones realizadas han sido completamente recibidas o por lo menos hasta el punto de poder tratarse como flujo de datos de representación. Para una aplicación en el navegador, este estado se corresponde con una "página Web", incluyendo la representación principal y las representaciones auxiliares, tales como imágenes, applets, y las hojas del estilo. La importancia de los estados estacionarios es ver el impacto que tiene en la **forma de percibir un usuario** el funcionamiento y ver como afecta el tráfico a ráfagas a la red.

La percepción que tiene el usuario del funcionamiento de una aplicación en el navegador está determinada por el **tiempo de retardo** entre estados estacionarios; el periodo de tiempo que transcurre entre el instante en el que se selecciona un link de hipermedio de la página Web, y el instante en el que la información ha sido renderizada. La optimización del funcionamiento de un navegador está centrada en reducir este retardo de comunicación.

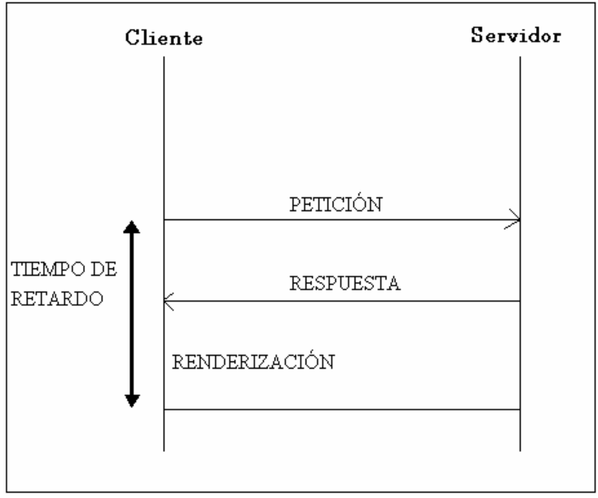

Figura 18: Tiempo de retardo

Como las arquitecturas basadas en REST se comunican transfiriendo representaciones de recursos, el tiempo de retardo puede mejorar mediante el diseño de los protocolos de comunicación y el diseño de los formatos de representación de los datos. La capacidad de renderizar una respuesta a la vez que está siendo recibida está determinada por el diseño de los tipos de medios.

Una observación interesante es que la **petición más eficiente** de la red es la que no utiliza la red. Es decir, la capacidad de reutilizar una respuesta ***cacheable*** da lugar a una mejora considerable del funcionamiento de la aplicación. Aunque el uso del caché añade retardo a cada petición individual debido a las operaciones de búsqueda, el retardo medio de la petición se reduce significativamente cuando incluso es solo un pequeño porcentaje de peticiones el que consigue obtener una respuesta del caché.

El siguiente punto de control de una aplicación reside en la **representación del primer recurso solicitado**, obtener esa representación es una prioridad. Por tanto, la interacción de REST se mejora con los protocolos que "primero responden y después piensan". En otras palabras, un protocolo que requiere múltiples interacciones sobre cada acción del usuario y que da prioridad a cosas como negociar las capacidades, será desde el punto de vista perceptivo más lento que otro protocolo que envíe la respuesta que crea más conveniente y si se ha equivocado, proporcione al cliente una lista con las posibles alternativas.

El estado de la aplicación está controlado y almacenado por el usuario y puede estar compuesto por representaciones de múltiples servidores. Esto libera al servidor de los problemas de escalabilidad que supone almacenar el estado. Además permite al usuario manipular directamente el estado, anticiparse a los cambios y saltar de una aplicación a otra (por ejemplo mediante los libros de direcciones y los diálogos de entradas URI).

La aplicación modelo es por tanto un motor que se mueve de un estado al siguiente examinando y eligiendo entre las transiciones de estado alternativas. Se hace teniendo en cuenta el interfaz usuario del navegador de hipermedios. Sin embargo, el **estilo no obliga** a que **todas las aplicaciones sean navegadores**. De hecho, los detalles de la aplicación se ocultan desde el servidor mediante el conector de interfaz genérico y así un usuario podría ser un robot que realiza una recuperación de datos para un servicio de indexación, un agente que busca información según unos criterios o un programa de mantenimiento que comprueba que no haya enlaces de referencia rotos.

5.7 HTTP, URI y XML en REST

Desde 1994, REST ha sido usado para guiar y diseñar la arquitectura de la Web moderna. Ahora trataremos de describir la experiencia aprendida de aplicar REST a los estándares de Internet para HTTP, URI y XML. HTTP y URI son las especificaciones que definen un interfaz genérico usado por todos los componentes de interacción en la Web.

5.7.1 Estandarizando la Web

La motivación de desarrollar REST era crear un modelo de arquitectura que describiese como debería funcionar la Web, así como que pudiese servir como marco de trabajo para los estándares de protocolos Web. REST ha sido aplicado para describir la arquitectura Web deseada, ayudar identificando problemas existentes, comparar soluciones alternativas, y asegurarse de que el protocolo no viole las restricciones que hacen que la Web funcione correctamente. Este trabajo fue realizado por el Internet Engineering Taskforce (IETF) y el World Wide Web Consortium (W3C), gracias a sus esfuerzos por definir un estándar de arquitectura para la Web: HTTP, URI y HTML.

La primera edición de un estilo de arquitectura parecido a REST fue desarrollada entre Octubre de 1994 y Agosto de 1995, principalmente como un medio para comunicar los conceptos Web como se escribieron las especificaciones de HTTP/1.0 y los inicios de

HTTP/1.1. Fue continuamente mejorado en los siguientes cinco años y se aplicó a varias revisiones y extensiones de los estándares de protocolos Web. A REST se le conoció originalmente por "HTTP Object Model", un nombre que puede inducir a confusión. El nombre de "Representational State Transfer" intenta evocar la imagen de cómo se comporta una aplicación Web bien diseñada: una red de paginas Web (una máquina de estados virtual), dando como resultado la siguiente página (representando el siguiente estado de la aplicación) que se le transfiere al usuario para que lo renderice y use.

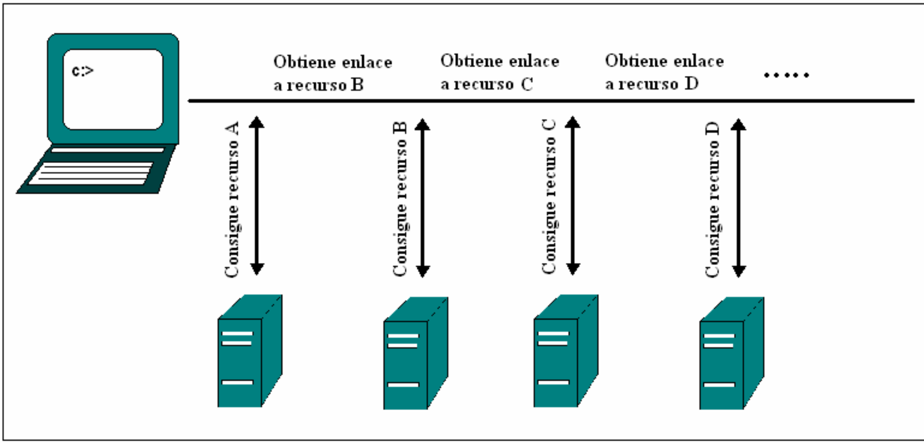

Figura 19: REST transición de estados

REST no esta concebido para capturar todas las posibilidades de uso de los estándares de protocolos Web. Hay aplicaciones de HTTP y URI que no encajan en el modelo de aplicación de un sistema de hipermedios distribuidos. Lo importante, sin embargo, es que REST captura todos los aspectos de los sistemas de hipermedios distribuidos que son considerados imprescindibles para el comportamiento y funcionamiento de la Web. Optimizar el comportamiento del modelo dará como resultado un comportamiento óptimo de la arquitectura Web desarrollada. En otras palabras, REST está optimizado para el caso común, de manera que las restricciones aplicadas a la arquitectura Web están optimizadas para ese caso.

5.7.2 REST aplicado a URI

Uniform Resource Identifiers (URI) son los elementos más simples de la arquitectura Web y a la vez los más importantes. URI ha sido conocido por muchos nombres: direcciones WWW, Universal Document Identifiers, Universal Resource Identifier, URL y URN. Dejando a un lado el nombre, la sintaxis URI ha permanecido relativamente sin cambios desde 1992. Sin embargo, la especificación de las direcciones Web también define el alcance y la semántica de lo que entendemos por recurso, el cual ha cambiado desde las primeras arquitecturas Web. REST define **URI** como **identificador de recursos**. También definió toda la semántica de la interfaz genérica para manipular los recursos a través de sus representaciones.

5.7.2.1 Redefinición de recurso

Las primeras arquitecturas Web definían **URI** como un **identificador de documentos**. Los identificadores se definían en los términos de localizaciones de documentos en la red. Los protocolos Web podían ser usados para recuperar esos documentos. Sin embargo, esta definición resultó ser insatisfactoria por varias razones:

- El autor estaría identificando el contenido transferido, lo que implicaría que el identificador debería cambiar siempre que el contenido cambie.

- Existen muchas direcciones que se corresponden con un servicio y no con un documento.

- Existen direcciones que no se corresponden con un documento en algunas ocasiones, como sucede cuando un documento ya no existe o cuando la dirección se está usando solamente para nombrar y no para alojar información.

La definición de recurso en REST está basada en una simple premisa: los identificadores deberían cambiar lo menos posible porque la Web no usa servidores de links, los autores necesitan un identificador que encaje con la semántica que ellos conciben como una referencia a hipermedio, permitiendo a la referencia permanecer estática aunque el resultado de acceder a esa referencia cambie con el tiempo. REST logra esto definiendo un **recurso** como la semántica de lo que el **autor piensa identificar** mejor que definirlo como el valor correspondiente a esa semántica en el momento en el que se crea la semántica. Por tanto es tarea del autor asegurarse de que el identificador elegido para una referencia identifica la semántica.

5.7.2.2 Manipulación

Definir los recursos como una URI que identifica un concepto en vez de un documento, deja una pregunta: ¿Cómo se accede, manipula o transfiere un concepto para conseguir algo útil cuando seleccionamos un link de hipertexto? REST responde a esta pregunta definiendo las cosas que son manipuladas como **representaciones del identificador de recurso**, en vez del recurso en sí. Un servidor mantiene la relación entre el identificador de recurso y el conjunto de representaciones que le corresponden. Por tanto, un recurso se manipula transfiriendo representaciones a través de una interfaz genérica definida por el identificador de recurso.

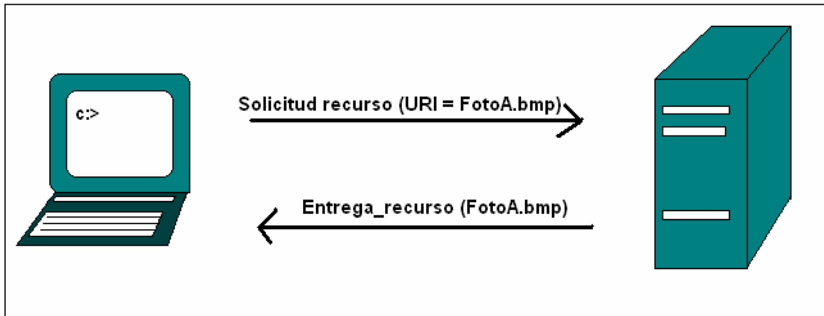

Figura 20: Solicitud recurso

La información que se oculta es una de las claves de la ingeniería del software que motiva el interfaz uniforme de REST, porque un cliente esta restringido a manipular la representación de un recurso en vez de acceder directamente a su implementación. La implementación puede ser construida en la forma deseada por su autor sin que esto tenga ningún impacto en como los clientes usan la representación. Además si existen múltiples representaciones del recurso en el instante en el que se quiere acceder a él, un algoritmo de selección de contenido puede ser usado dinámicamente para seleccionar el que mejor se adapte a las necesidades del cliente. La desventaja, por supuesto, es que un recurso no es tan directo como un archivo.

Las **URI** a las que se accede pueden ser **físicas o lógicas**. Visto de otra manera, los recursos a los que acceden los clientes pueden ser generados dinámicamente (URI lógica). Por ejemplo, si se accede a una aplicación que está conectada a una base de datos en el servidor, al realizar una consulta, se está solicitando una búsqueda en concreto, y no hace falta que el servidor tenga almacenadas todas las posibles combinaciones de las búsquedas como páginas estáticas, puede conectarse a la base de datos, y cuando obtenga el resultado generar dinámicamente la representación del recurso.

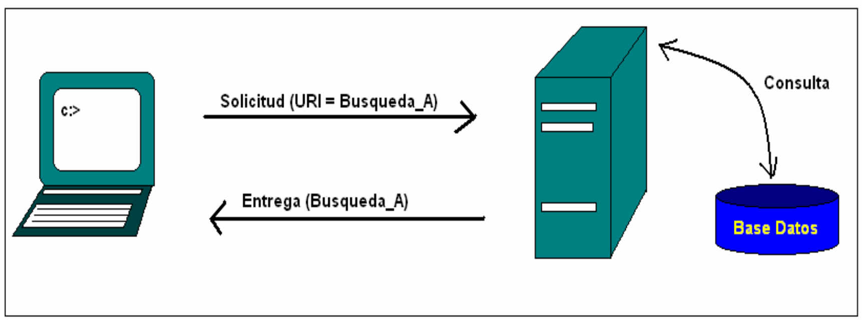

Figura 21: Representación recurso dinámico

Este mecanismo también puede ser útil para solicitar una representación de un recurso en un determinado formato. Por ejemplo, se puede solicitar que la información sea devuelta en formato XML, HTML… En este caso, el servidor debería acceder al recurso, generar la representación en el formato que se le indique y entregarla. La información tendrá diferentes URIs según su representación, por ejemplo: `http://informacion_deseada.xml` o `http://informacion_deseada.html`.

5.7.2.3 Acceso remoto

El reto del acceso remoto a través de un interfaz uniforme de la Web es debido a la separación entre la representación que puede ser recuperada por un cliente y el mecanismo que se usa en el servidor para almacenar, generar o recuperar el contenido de esa representación.

Para poder trabajar con un recurso existente, el autor primero tiene que obtener la URI del recurso fuente específico: el conjunto de URIs que emparejan el manejador subyacente con el recurso objetivo. No siempre un recurso se mapea como un fichero en concreto, pero todos los recursos que no son estáticos derivan de otros recursos. Un autor podría encontrar todos los recursos fuente que deben ser editados para modificar la representación de un recurso. Los mismos principios se aplican a cualquier forma derivada de representación, negociación de contenidos, scripts, servlets, configuraciones de mantenimiento…

El recurso no es el objeto almacenado. El recurso no es un mecanismo que usen los servidores para manejar el objeto almacenado. El recurso es un mapeo conceptual. El servidor recibe el identificador (que identifica el mapeo) y lo aplica a una implementación particular del mapeo para encontrar la implementación del manejador que le corresponde en ese momento. Posteriormente, la implementación del manejador selecciona la acción y respuesta apropiadas basándose en el contenido solicitado. Su naturaleza no se le puede revelar un cliente que tenga solamente acceso a través del interfaz Web.

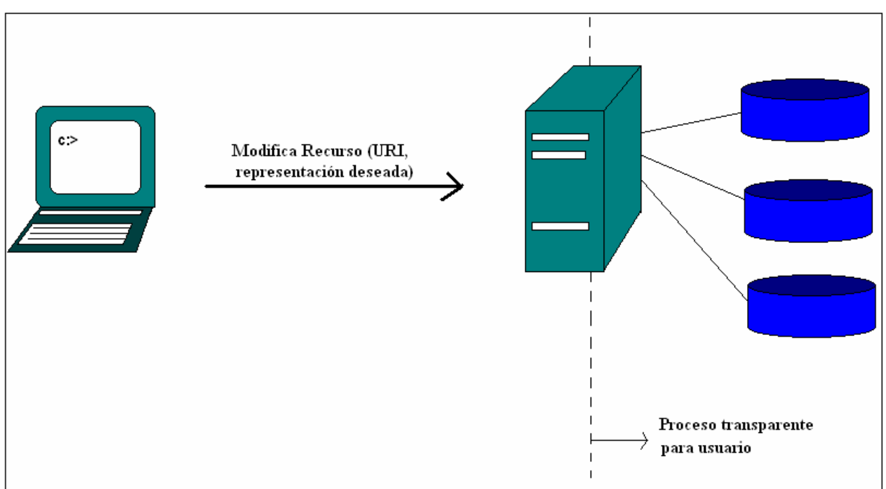

Figura 22: Modificación Recurso

Por ejemplo, se va a considerar el caso de lo que ocurre cuando un sitio Web crece en cuanto al número de usuarios y decide reemplazar sus viejos servidores X basados en una plataforma XOS por servidores Apache corriendo en FreeBSD. El hardware de almacenamiento de disco se cambia. El sistema operativo también se cambia. El servidor HTTP también se cambia. Quizás hasta los métodos de generación de respuestas se cambian. Sin embargo, lo que no necesitamos cambiar es el interfaz Web: si se diseña correctamente, el espacio de nombres del nuevo servidor puede migrar

desde el viejo, esto significa que desde la perspectiva del cliente que solo sabe de recursos y no de cómo están implementados, nada ha cambiado excepto la robustez del sitio.

5.7.2.4 Ligando la semántica al URI

Como se ha mencionado, un recurso puede tener muchos identificadores. En otras palabras, pueden existir dos o más URIs diferentes que tienen semántica equivalentes cuando se usan para acceder a un servidor. También es posible tener dos URIs que den como resultado el mismo mecanismo cuando se usan para acceder al servidor, y que sin embargo cada URI identifique un recurso diferente, porque no signifiquen lo mismo.

La semántica es un producto de asignar los identificadores de recursos y de dotar a los recursos con representaciones. En cualquier momento, el software del servidor o del cliente necesita conocer o comprender el significado de una URI. Ellos actúan conducidos por la asociación que hizo el creador entre un recurso y la representación de la semántica identificada por el URI. En otras palabras, no hay recursos en el servidor, solamente hay mecanismos que proporcionan respuestas a través de un interfaz definido por los recursos. Puede parecer extraño pero es la esencia de lo que hace que la Web funcione con tantas y tan diferentes implementaciones.

Es competencia de todo ingeniero definir las cosas en términos de las características de los componentes que serán usados para obtener el producto terminado. La Web no funciona de esa manera. La arquitectura Web consiste en restricciones de los componentes del modelo de comunicación, basadas en el rol de cada componente durante una acción de la aplicación. Esto evita que los componentes tengan que asumir nada que vaya más allá de la abstracción de recurso, escondiendo así el mecanismo que se usa en cada lado del interfaz abstracto.

5.7.2.5 Problemas entre REST y URI

Como la mayoría de los sistemas del mundo real, no todos los componentes que se desarrollan de la arquitectura Web obedecen todas las restricciones presentes en este diseño de arquitectura. REST ha sido usado para definir mejoras en la arquitectura y para identificar errores. Los problemas ocurren cuando, debido a la ignorancia o descuido, una implementación de software se desarrolla violando las restricciones arquitectónicas. Mientras que los problemas no se puedan evitar en general, es posible identificarlos antes de que se estandaricen.

Aunque el diseño de URI encaja perfectamente con las características de los identificadores de REST, la sintaxis sola es insuficiente para forzar a los autores a definir sus propias URI de acuerdo al modelo de recurso. Una práctica errónea es incluir información que **identifique al usuario** dentro de la **URI** que referencia la representación de una respuesta de hipermedio porque eso significaría que el servidor controlaría el estado de la sesión almacenando las preferencias de los usuarios y cosas similares. Sin embargo, violando las restricciones de REST, esos sistemas producen que

el **caché compartido no sea efectivo**, reduciendo la escalabilidad del servidor y dando como resultado efectos indeseables cuando los usuarios comparten referencias con otros.

Otro conflicto con el interfaz de recursos de REST ocurre cuando el software intenta tratar la Web como un **sistema distribuido de ficheros**. Puesto que los sistemas de ficheros exponen la implementación de su información, las herramientas existen para poder migrar esa información a través de múltiples sitios para hacer un balance de cargas y redistribuir el contenido a los usuarios. Sin embargo, pueden hacerlo sólo porque los ficheros tienen un conjunto de semántica fijo (una secuencia de bits conocidos) que pueden ser fácilmente duplicados. Por el contrario, tratar de migrar el contenido de un servidor Web como ficheros fallará porque la interfaz de recurso no siempre encaja con la semántica de sistema de ficheros, y porque los datos y metadatos están incluidos y dan sentido a la semántica de una representación. El contenido de un servidor Web puede ser copiado a sitios remotos, pero solo si se copia también el mecanismo y la configuración del servidor, o copiando selectivamente solo los recursos cuyas representaciones se sabe que son estáticas.

5.7.3 REST aplicado a HTTP

HTTP tiene un papel especial en la arquitectura Web por ser el principal protocolo del nivel de aplicación para la comunicación entre componentes Web, y por ser el único protocolo diseñado específicamente para transferir representaciones de recursos. Al contrario que URI, se necesitaron muchos cambios para que HTTP soportase la nueva arquitectura Web.

Este apartado va a centrarse en tres características de HTTP:

- Extensibilidad
- Mensajes Auto-descriptivos
- Métodos HTTP y su uso en REST

5.7.3.1 Extensibilidad

Una de las mayores metas de REST es soportar el despliegue gradual y fragmentado de cambios dentro de una arquitectura ya desarrollada. HTTP fue modificado para soportarlo mediante la introducción de requerimientos y reglas que extienden cada elemento de la sintaxis de protocolo.

5.7.3.1.1 Versiones del protocolo

HTTP es una familia de protocolos, distinguidos por un número mayor o menor que comparten el nombre principal porque corresponden al protocolo esperado cuando se comunican directamente con un servicio basado en el espacio de nombres URL. Un

conector debe obedecer a las restricciones que indica la versión del elemento del protocolo HTTP incluida en cada mensaje.

La versión HTTP de un mensaje representa las capacidades del protocolo del remitente y la compatibilidad (el mayor número de versión) del mensaje que se envía. Esto permite a un cliente usar un subconjunto reducido (HTTP/1.0) de características a la hora de realizar una petición normal de HTTP/1.1, mientras que al mismo tiempo se le indica al receptor que soporta una comunicación completa en HTTP/1.0. En otras palabras, se permite una especie de negociación de la forma del protocolo. Cada conexión en la cadena de peticiones y respuestas puede operar con el mejor nivel de protocolo teniendo en cuenta la limitación de algunos clientes o servidores que tomen parte en la cadena.

La intención del protocolo es que el servidor debería siempre responder con la mayor versión del protocolo y que se entienda con la mayor versión del mensaje de petición del cliente. La restricción es que el servidor no puede usar las características opcionales de su mayor versión del protocolo que no estén presentes en el cliente. No existen características exigidas de un protocolo que no puedan ser usadas entre versiones anteriores y posteriores porque sería un cambio incompatible. Las únicas características de HTTP que pueden depender de la versión del protocolo son las que se interpretan por sistemas adyacentes de la comunicación, porque HTTP no requiere que toda la cadena de componentes intermediarios en las peticiones y respuestas tenga la misma versión.

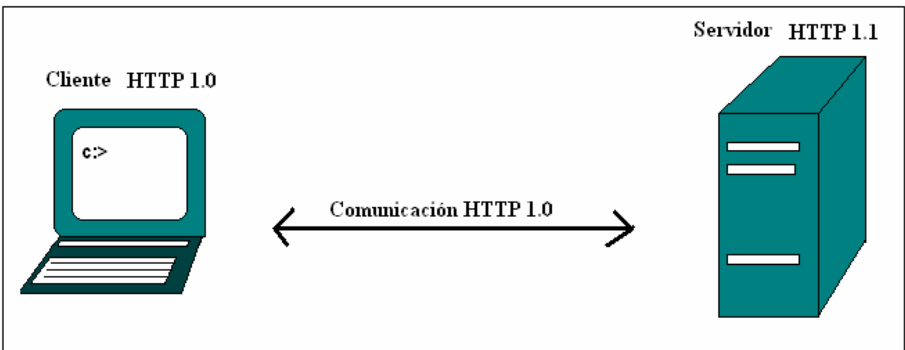

Figura 23: Versiones HTTP Cliente-Servidor

Esas reglas existen para ayudar al despliegue de múltiples revisiones de los protocolos y para prevenir que los arquitectos de HTTP olviden que el despliegue del protocolo es un aspecto importante del diseño. Lo realizan haciendo fácil la diferenciación entre cambios compatibles con el protocolo y cambios incompatibles. Los compatibles son fáciles de desplegar y la comunicación de las diferencias se puede alcanzar dentro del flujo del protocolo. Los cambios incompatibles son difíciles de desplegar porque necesitan asegurar que son aceptados antes de que el flujo del protocolo comience.

5.7.3.1.2 Elementos de protocolo extensibles

Desde los orígenes, HTTP no define un conjunto de reglas consistente para definir como deben desplegarse los cambios dentro del espacio de nombres. Éste era uno de los primeros problemas abordados por la especificación.

La semántica de la petición del HTTP está significada por el nombre del método de la petición. La **extensión del método está permitida** siempre que un conjunto de semánticas que se puedan estandarizar, pueda ser compartido entre cliente, servidor y cualquier sistema intermedio que haya entre ellos. Desafortunadamente, las primeras extensiones de HTTP, específicamente el método **HEAD**, hicieron que el análisis de una respuesta HTTP fuera dependiente del conocimiento de la semántica del método requerido. Esto condujo a una contradicción en el despliegue: si un receptor necesita conocer la semántica de un método antes de que pueda ser reenviado con seguridad a un sistema intermedio, entonces todos los sistemas intermedios deben estar actualizados antes de que se pueda desplegar.

Estos problemas en el despliegue **fueron solucionados** separando las reglas de análisis y reenvío de mensajes HTTP de la semántica asociada con los nuevos elementos del protocolo HTTP. Por ejemplo, HEAD es el único método para el que la longitud del contenido en el campo de cabecera, tiene otro significado que decir la longitud del cuerpo del mensaje, y ningún nuevo método puede cambiar el cálculo de la longitud del mensaje.

Asimismo, HTTP necesita una regla general para interpretar los nuevos códigos de estado de la respuesta, de manera que las nuevas respuestas podrían ser desplegadas sin dañar a los viejos clientes. Por tanto se impone la regla de que cada código de estado pertenezca a una clase diferenciada por los tres primeros dígitos decimales:

RANGO DE CÓDIGOS	SIGNIFICADO DEL MENSAJE
100-199	El mensaje contiene una respuesta de información provisional
200-299	La petición tuvo éxito
300-399	La petición necesita ser redirigida a otro recurso
400-499	El cliente cometió un error que no debe repetirse
500-599	El servidor encontró un error, pero que el cliente podrá obtener la respuesta más tarde (o a través de otro servidor)

Tabla 7 Significado mensajes HTTP

Como la regla para los nombres de los métodos, ésta regla de extensibilidad introduce un requisito en la arquitectura actual, que es anticipar futuros cambios. Los cambios pueden ser desplegados en una arquitectura existente sin que haya reacciones adversas en los componentes.

5.7.3.1.3 Actualizaciones

Añadiendo en la cabecera, el campo de actualización para HTTP/1.1 se reduce la dificultad de desplegar cambios incompatibles, permitiendo al cliente avisar de que quiere usar un protocolo mejor mientras se comunica con un protocolo más antiguo. La actualización fue añadida específicamente para soportar la sustitución de HTTP/1.x con los futuros protocolos que serán más eficientes con algunas tareas. Por tanto, HTTP no sólo soporta extensibilidad interna, sino que también soporta una sustitución completa durante una conexión activa. Si el servidor soporta un protocolo mejor y decide cambiar, solamente responde con un estado 101 y continúa como si la petición fuese recibida en ese protocolo mejorado.

5.7.3.2 Mensajes auto-descriptivos

REST tiene la restricción de que los mensajes entre componentes sean auto-descriptivos para soportar un procesamiento inmediato de las interacciones. En este apartado se estudiarán algunas características de los mensajes HTTP.

- La identificación de un host dentro de la petición
- Codificación de capas
- Independencia semántica
- Independencia del transporte
- Límites de tamaño
- Control de Caché
- Negociación del contenido

5.7.3.2.1 Identificación de Host dentro de la petición

HTTP/1.0 y HTTP/1.1 incluyen la información de la URL del host de destino dentro del campo cabecera de un mensaje de petición. El despliegue de esta característica se hizo tan importante que la especificación HTTP/1.1 indica que se rechace cualquier petición que no contenga el campo host. Como resultado, existen muchos servidores ISP que

contienen decenas de miles de sitios Web virtuales basados en el nombre, que comparten la misma dirección IP.

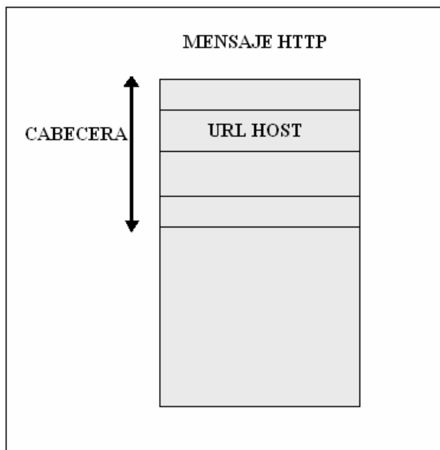

Figura 24: Mensaje HTTP, URL HOST

5.7.3.2.2 Codificación de capas

HTTP hereda su sintaxis de MIME. MIME no define tipos de medios basados en capas, prefiere colocar una etiqueta del tipo de medios exterior dentro del valor del campo del tipo de contenido. Sin embargo, esto impide a un receptor determinar la naturaleza de un mensaje codificado sin decodificar las capas. Una de las primeras extensiones de HTTP trabajaba sobre este fallo listando la codificación de las capas externas de forma separada dentro del campo de codificación de contenido, y colocando una etiqueta para los tipos de medios en el tipo de contenido. Esta fue una solución bastante mala porque cambiaba la semántica del tipo de contenido sin cambiar el nombre del campo, como resultado creaba confusión.

Una solución mejor habría sido continuar tratando el tipo de contenido como el tipo de medio más externo, y usar un nuevo campo para describir el tipo anidado dentro de ese tipo. Desafortunadamente, la primera extensión fue desplegada antes de identificar el fallo.

REST identificó la necesidad de otra capa de codificación: aquella situada en un mensaje por un conector para mejorar la transferencia por la red. Esta nueva capa, llamada **Transfer-Encoding** en alusión a un mismo concepto de MIME, permite a los mensajes ser codificados para transferirse sin que esto implique que la representación está codificada por naturaleza. Transferir codificaciones puede ser añadido o quitado por los agentes que transfieren, por cualquier razón, sin cambiar la semántica de la representación.

5.7.3.2.3 Independencia semántica

Como se ha descrito antes, el análisis de un mensaje HTTP se ha separado de su semántica. El análisis del mensaje, incluyendo la búsqueda y procesado de los campos de cabecera, ocurre de forma separada del proceso de análisis del campo de contenido. De esta manera los sistemas intermedios pueden procesar y reenviar rápidamente los mensajes HTTP, y las extensiones pueden ser desplegadas rápidamente sin influir esto en los analizadores existentes.

5.7.3.2.4 Independencia del transporte

Las primeras versiones de HTTP, incluyendo la mayoría de las implementaciones de HTTP/1.0, usaban el protocolo de transporte subyacente, con él indicaban el final de la respuesta de un mensaje. Un servidor indicaba el final de un mensaje de respuesta cerrando la conexión TCP. Desafortunadamente, esto constituye una condición de fallo considerable en el protocolo: un cliente no tiene modo alguno de saber si una respuesta ha finalizado o simplemente se vio truncada por un error en la red. Para solucionar esto, en HTTP/1.0 se incluyó un campo de cabecera que indicaba la longitud del contenido en bytes siempre que la longitud sea conocida.

Para HTTP/1.1 se definió la codificación para "partes", con ella, para una representación cuyo tamaño es desconocido al comienzo de su generación (cuando se envía el campo cabecera), se permite dividirla en pedazos de los cuales se conocen su tamaño. Con este método también se permite enviar los metadatos al final del mensaje, permitiendo así crear metadatos opcionales al principio mientras se están generando, sin añadir tiempo de retardo.

5.7.3.2.5 Limites de tamaño

Una barrera frecuente para la flexibilidad de la capa de aplicación es la tendencia a sobre-especificar los límites de tamaño en los parámetros del protocolo. Aunque existen algunos límites parciales en los parámetros del protocolo (memoria disponible), especificando esos límites en el protocolo se restringe a todas las aplicaciones a los mismos límites, independientemente de su implementación. El resultado es que se obtiene un protocolo que no puede extenderse mucho de su creación original.

En el protocolo HTTP, no hay límites para la longitud del URI, del campo cabecera, de la longitud de la representación, o de la longitud de cualquier valor de un campo que consista en una lista de datos. Aunque los antiguos clientes Web tienen problemas bien conocidos con las URI de más de 255 caracteres, esto es suficiente para darse cuenta de que el problema está en la especificación de HTTP y no en limitar a todos los servidores.

Aunque no necesitaba inventar limitaciones artificiales, HTTP/1.1 necesitó definir un conjunto apropiado de códigos de estado de respuesta para indicar cuando un elemento de un protocolo dado es demasiado grande para que los servidores lo procesen. Dichos códigos de respuesta fueron añadidos en las condiciones URI-Solicitada demasiado

grande, campo cabecera demasiado grande, y cuerpo del mensaje demasiado grande. Desafortunadamente, no hay manera de que un cliente indique a un servidor que puede tener límite de recursos, lo que conlleva problemas cuando un dispositivo con recursos limitados, como puede ser una PDA, intenta usar HTTP sin un interfaz intermediario que posibilite la comunicación.

5.7.3.2.6 Control de Caché

Como REST trata de equilibrar la necesidad de eficiencia y tiempo de espera con la transparencia semántica del comportamiento del caché, es muy necesario que HTTP permita a la aplicación determinar los requerimientos de caché antes que imponerlos por si mismo. Lo más importante que tiene que hacer el protocolo es describir completa y exactamente los datos que se van a transferir, de manera que ninguna aplicación se vea engañada al pensar que tiene algo cuando realmente tiene otra cosa diferente. HTTP/1.1 hace esto añadiendo el control de caché y varios campos de cabecera.

5.7.3.2.7 Negociación de Contenido

Todos los recursos transforman una petición (que consiste en un método, campo cabecera de petición, y a veces una representación) en una respuesta (que consiste en un código de estado, campo de cabecera de la respuesta, y a veces una representación). Cuando una petición HTTP puede dar lugar a múltiples representaciones en el servidor, el servidor puede mantener una negociación del contenido con el cliente para determinar cuál es la representación que mejor se adapta a las necesidades del cliente. Esto es realmente una selección de contenido más que una negociación.

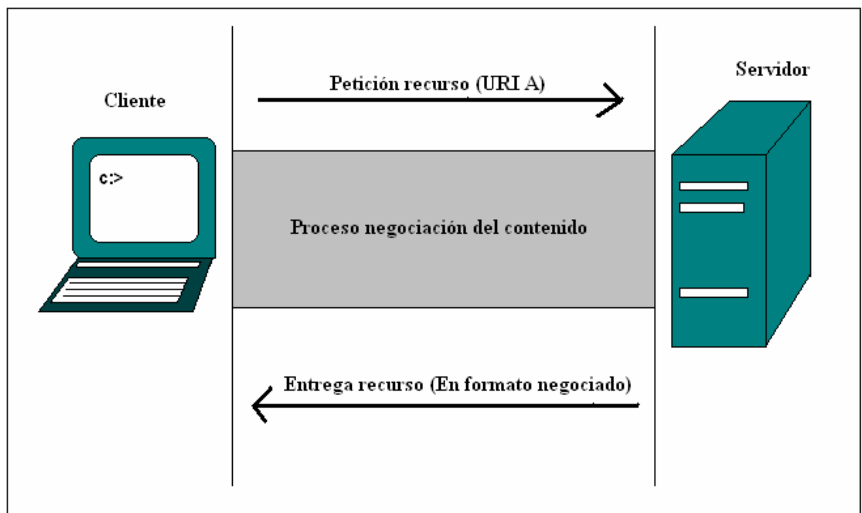

Figura 25: Negociación del contenido

Aunque hubo muchas implementaciones de la negociación del contenido desplegadas, no se incluyó en la especificación de HTTP/1.0 porque no existió un subconjunto

interoperable en el momento en que se publicó. Esto fue en parte, debido una pobre implementación de NCSA Mosaic, la cual debía enviar 1Kb de información de preferencia en los campos de cabecera para hacer la negociación del recurso. Como menos del 0,001% de todas las URI pueden negociar el contenido, el resultado aumentaba considerablemente el tiempo de retardo y aportaba unos beneficios muy pequeños, lo que condujo a que los navegadores posteriores evitaran la negociación del contenido.

La **negociación guiada por el servidor** ocurre cuando el servidor varía la representación de la respuesta por causa de una determinada combinación del código de estado en el campo cabecera del método de petición, o por cualquier variación extraordinaria en los parámetros. El cliente necesita ser notificado cuando esto ocurre, de manera que un caché pueda conocer cuando es semánticamente transparente usar una respuesta del caché para una petición futura, y también que un usuario pueda sustituir las preferencias detalladas. HTTP/1.1 introduce el campo *Vary* en la cabecera para este propósito. *Vary* simplemente lista las dimensiones del campo cabecera sobre las que una respuesta puede variar.

En la negociación guiada por el servidor, el usuario le dice al servidor lo que es susceptible de aceptar. El servidor, entonces, elige la representación que mejor se ajusta a las capacidades del cliente. Sin embargo, esto no es un problema manejable, porque requiere no solo información de lo que el usuario acepta sino también de cómo de bien acepta las características y cual era el propósito del usuario con la representación. Por ejemplo, si un usuario quiere ver una imagen en la pantalla, puede que prefiera un simple mapa de imágenes, pero el mismo usuario con el mismo navegador puede que prefiera una representación PostScript si lo que realmente quiere hacer es imprimirla. También depende de si el usuario ha configurado correctamente el navegador de acuerdo con sus preferencias en cuanto al contenido. En resumen, un servidor raramente podrá hacer afectiva la negociación guiada por el servidor, pero es la única forma automática definida por las primeras versiones de HTTP.

HTTP/1.1 añadió la noción de **negociación guiada por el usuario**. En este caso, cuando un usuario solicita un recurso negociable, el servidor responde con una lista de posibles representaciones. El usuario puede elegir cual es la mejor de acuerdo con sus propias capacidades. La información acerca de las representaciones disponibles puede ser suministrada por medio de una representación diferente, dentro de los datos de la respuesta, o como suplementaria a la respuesta que mejor se adapta. La última funciona mejor en la Web porque las interacciones adicionales solo llegan a ser necesarias si el usuario decide que una de las variantes es mejor. La negociación guiada por el usuario es simplemente es un reflejo automático de el modelo normal de servidor, lo que significa que puede tomar todas las ventajas del funcionamiento de REST.

Las negociaciones guiadas por el usuario y el servidor tienen la dificultad de comunicar las características actuales de las dimensiones de la representación (por ejemplo, decir si un navegador soporta tablas HTML pero no insertar elementos). Sin embargo, la negociación guiada por el usuario tiene las ventajas de no tener que enviar las preferencias en cada petición, teniendo así mayor información del contexto para hacer una decisión ante distintas alternativas, y no interferir con la caché.

Una tercera forma de negociación, es la **negociación transparente**, da el permiso a un caché intermediario de actuar como agente, para seleccionar la mejor representación. La petición puede ser resuelta internamente por cualquier otro caché, y por tanto es posible que ninguna petición de la red sea hecha. Haciendo esto, sin embargo, están funcionando como negociaciones guiadas por el servidor, deben por tanto añadir la información adecuada para que otras caché no se equivoquen.

5.7.3.3 Métodos HTTP y su uso en REST

Usar los métodos HTTP GET, POST, PUT y DELETE, es suficiente para poder implementar cualquier Servicio Web. A continuación se adjunta la Tabla [7] en la que se indica la función de cada uno dentro de una comunicación REST.

MÉTODO	FUNCIÓN
GET	Solicitar recurso
POST	Crear recurso nuevo
PUT	Actualizar o modificar recurso
DELETE	Borrar recurso

Tabla 8 Métodos REST

Todos los **Servicios** accesibles vía **HTTP** deben ser **idempodentes** [8], "Idempodente" significa que una vez que se le pasa un mensaje a un Servicio, no existen efectos adicionales si se pasa el mismo mensaje de nuevo. Por ejemplo, un mensaje cuya función es borrar datos, "Borra el recurso A". Se puede enviar una vez, o diez veces consecutivas, pero todo debe quedar igual al final de cada procesado del mensaje. De la misma manera, GET es siempre idempodente, porque no afecta al estado. Los métodos GET, DELETE, PUT y POST que usa REST, están concebidos para ser idempodentes.

5.7.4 REST y XML

REST está especialmente orientado a los sistemas distribuidos de hipermedios. Tiene la necesidad de crear representaciones a partir de recursos, éstas pueden ser: gráficos, sonidos, páginas HTML, video o cualquier tipo de datos en general. Desde los comienzos de REST se apostó muy fuerte por XML como solución a la codificación de las representaciones, sobre todo debido su potencial a la hora de codificar cualquier tipo de datos. Esta relación entre REST y XML no es una imposición, es más bien una recomendación a tener en cuenta a la hora de codificar representaciones complicadas o que no están estandarizadas. Si un usuario realiza una petición para obtener información y por diversas causas no es conveniente o práctico usar XML, es algo totalmente razonable.

La recomendación de XML ha ayudado a poner un poco de orden en la situación caótica de los comienzos de REST

5.7.5 Repercusiones de usar HTTP y URI

Si unimos HTTP y URI a la naturaleza de REST, que apuesta por la transparencia en vez del encapsulamiento, obtenemos unas excelentes características de **seguridad y tiempo respuesta**.

La **seguridad** se ve mejorada debido a la facilidad que encuentran los Firewalls a la hora de aplicar una política de acceso. HTTP unido a REST es totalmente transparente, sólo con mirar la cabecera de un mensaje y observar el método, el origen y el destino, se puede decidir si se permite a un usuario el acceso a los recursos. Es bastante sencillo evitar código malicioso oculto dentro de la carga de un mensaje.

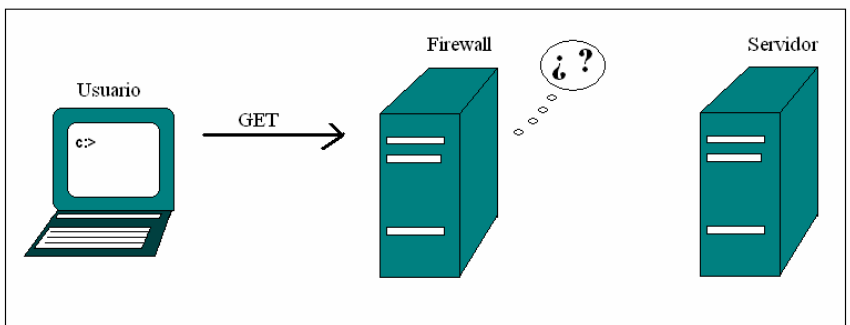

Figura 26: Comportamiento Firewall

Como se puede ver en la Figura [27], si llega una petición GET, y existen dudas sobre su intención, el administrador del servidor podrá estar tranquilo porque GET, por su propia definición, sólo puede solicitar recursos y por lo tanto no le está permitido modificar los datos del servidor.

La transparencia también tiene su repercusión en el tiempo de respuesta. Los sistemas intermedios pueden decidir rápidamente si una respuesta entregada por el servidor es "cacheable" (por tanto, debe ser almacenada), o si una petición del usuario puede ser respondida desde caché. Lo único que necesitan mirar es la cabecera del mensaje.

5.8 Cuestiones por resolver

En este apartado se va a exponer la opinión de algunos autores sobre algunos cabos sueltos o aspectos que deja REST en el aire a la hora de diseñar Servicios Web.

5.8.1 David Megginson

David Megginson es el director de Megginson Technologies, ha sido un miembro activo de las comunidades SGML y XML. Actualmente también pertenece a un grupo de trabajo sobre información XML en el W3C.

Este autor trata de plantear y dar solución a cuatro aspectos del diseño de servicios Web REST: la identificación de recursos, el descubrimiento de servicios, el significado de los enlaces y la normalización. En un último apartado habla de la carencia que tiene REST en cuanto a la estandarización de los tipos de contenido.

5.8.1.1 Cuestiones de un diseño REST [11]

REST ahora tiene un significado más amplio y es fácil de explicar: piezas de datos (probablemente codificados con XML) situadas en la Web, que manipulamos usando los métodos HTTP GET, PUT y DELETE (prácticamente CRUD, excepto que *Create* y *Update* se encuentran combinadas en PUT). Si Tratamos de explicar SOAP en una sencilla frase como ésta, veremos la diferencia que existe.

Esta simplicidad debe hacer sonar algunas alarmas. ¿Esconde REST algunas trampas? Los Restafarians (adeptos a REST) señalan que REST constituye las bases del éxito de la Web, pero eso es solamente la parte GET (y POST). Tenemos muy poca experiencia usando PUT y DELETE en las páginas Web normales, y menos todavía para mantener datos almacenados. Incluso los grandes servicios de ventas de Amazon y eBay usan sólo GET (y POST en el caso de eBay); de hecho, la mayoría de los firewalls vienen configurados por defecto para bloquear PUT y DELETE desde que los administradores Web los ven como posibles agujeros de seguridad.
REST es más manejable que XML-RPC y WS-* a la hora de usar XML en la Web, pero hay algunos temas que debemos tratar antes. El mantenimiento de los datos no es fácil, y mientras que WS-* lo hace más difícil de lo que debe ser, ni el más simple de los modelos REST lo hace trivial. Vamos a estudiar algunos aspectos del diseño REST que son importantes:

- Identificación
- Listar y descubrir recursos
- El significado de un enlace
- ¿Cuánta normalización?
- La palabra “C” (contenido)

5.8.1.2 Identificación

La primera cuestión que trataremos es que los Restafarians consideran que la identificación y localización son la misma cosa. Siguiendo con esto, veremos cómo hacer una identificación persistente en recursos XML. Por ejemplo, asumamos que `http://www.example.org/airports/ca/cyow.xml` es al mismo tiempo un

identificador único de un objeto de datos XML y la localización de un objeto en la Web. Esto es un punto muy importante en REST. Según los Restafarians no debe haber interfaces donde los identificadores están ocultos dentro de un objeto XML devuelto de una petición POST a URLs no relacionadas.

5.8.1.2.1 GET y PUT

Veamos un caso simple. Diremos que descargamos el fichero de datos XML de `http://www.example.org/airports/ca/cyow.xml`. Este sería el ejemplo:

```
<airport>
 <icao>CYOW</icao>
 <name>Macdonald-Cartier International Airport</name>
 <political>
  <municipality>Ottawa</municipality>
  <region>ON</region>
  <country>CA</country>
 </political>
 <geodetic>
  <latitude-deg>45.322</latitude-deg>
  <longitude-deg>-75.669167</longitude-deg>
  <elevation-msl-m>114</elevation-msl-m>
 </geodetic>
</airport>
```

Lo copiamos en un lápiz USB, lo traemos a casa desde el trabajo, lo copiamos en el ordenador portátil, y trabajamos con el fichero mientras estamos sin conexión durante un vuelo. El fichero ya no tiene una conexión directa con su URL: ha pasado por otras transferencias desde que usamos HTTP GET para descargarlo. ¿Cómo sabemos con qué estamos trabajando o dónde deberíamos ponerlo cuando terminemos?

Si esta información no debe permanecer online, algunas de las ventajas de REST se evaporan, porque ahora no podemos usar las tecnologías Web existentes. Como un identificador, la URL claramente forma parte del estado del recurso, y corresponde al fichero de datos XML; como localización, sin embargo, es una información superflua y corresponde sólo al nivel de protocolo (HTTP).

5.8.1.2.2 ¿Dónde va el identificador de documento?

Asumamos que hemos decidido que la URL es un identificador adecuado y pertenece a la representación XML. Ahora, ¿Cómo hacemos esto de una manera elegante? xml:id está fuera de la cuestión porque está diseñado solo para llevar un nombre XML que identifique una parte de un documento, no una URI que identifica el documento entero. Podríamos usar (o abusar) de xml:base, sería algo como esto:

```
<airport xml:base="http://www.example.org/airports/ca/cyow.xml">
 ...
</airport>
```

No sabemos exactamente como trataría con esto un procesador XLink. ¿Se resolvería la URL relativa "cyyz.xml" como `http://www.example.org/airports/ca/cyyz.xml` o como `http://www.example.org/airports/ca/cyow.xmlcyyz.xml`? También existe la posibilidad de que algunas APIs bastante experimentadas ya, "digieran" primero el atributo xml:base de manera que el código de la aplicación nunca lo viese. ¿Verá la gente que apoya el estándar XML que el uso de xml:base es legítimo?

Si xml:id no se puede usar, y xml:base da problemas, parece que no hay una manera estándar de identificar documentos XML REST, y cada tipo de documento XML necesitará su propia solución. ¿Necesita el mundo otro atributo del tipo xml:*?

Necesitaremos escuchar como los desarrolladores de REST han tratado la persistencia del identificador cuando éste es la URL.

5.8.1.3 Listar y descubrir recursos

El segundo tema que vamos a tratar es el descubrimiento de los recursos. Comenzaremos con un ejemplo que tiene fallos e intentaremos corregirlo.

Digamos que tenemos una gran colección de de datos XML con URLs como:

`http://www.example.org/airports/cyow.xml`

`http://www.example.org/airports/cyyz.xml`

Como todos comparten el mismo prefijo, sería razonable asumir que usar una operación HTTP GET en ese prefijo (`http://www.example.org/airports/`) devolverá una lista de enlaces a todos los datos (debemos reconocer que las URL son opacas y por tanto nadie debería caer en la cuenta de esto, pero sólo es un ejemplo):

```
<airport-listing                                    xmlns:xlink="http://www.w3.org/1999/xlink"
xml:base="http://www.example.org/airports/">
  <airport-ref xlink:href="cyow.xml"/>
  <airport-ref xlink:href="cyyz.xml"/>
  <airport-ref xlink:href="cyxu.xml"/>
  ...
</airport-listing>
```

Esto es un ejemplo maravilloso de REST porque muestra como un motor de búsqueda podría encontrar e indexar cada recurso XML. Sin embargo, alguien que haya trabajado en un sistema grande puede ver que hay un problema de escalabilidad (dejando de lado otros problemas de privacidad y seguridad). Para una lista de varias docenas de recursos, es una gran aproximación. Para una lista de varios cientos, es manejable. Para una lista de varios miles de recursos, empieza a tener un gran consumo de ancho de banda cada vez que alguien hace un GET, y para una lista de varios millones de recursos, es simplemente ridículo.

Las aplicaciones basadas en HTML diseñadas para humanos normalmente emplean una combinación de búsqueda y paginación para tratar de descubrir recursos de una gran colección. Por ejemplo, podríamos empezar especificando que estamos interesados sólo en los aeropuertos que se encuentran a 200km de Toronto, la aplicación devolvería una única página de resultados (los 20 primeros resultados), con un enlace para dejarnos ver la siguiente página si nos interesa.

¿Cómo funcionaría esto en una aplicación de datos basada en REST? Claramente, queremos usar GET en vez de peticiones POST, porque las búsquedas puras están libres de efectos colaterales. Presumiblemente, terminaremos añadiendo algunos parámetros de búsqueda para limitar los resultados:

```
http://www.example.org/airports/?ref-point=cyyz&radius=200km&has-
iap=yes
```

Esto no es exactamente el tipo de URL REST que hay en los ejemplos, pero es parecida a la que se usa en los servicios Web REST de Amazon, por lo que quizás no estemos tan desencaminados. Por supuesto, deberá haber alguna manera para que los sistemas conozcan cuales son los parámetros disponibles. Ahora, quizá, el resultado sea algo como esto (asumiendo 20 resultados en la página):

```
<airport-listing xmlns:xlink="http://www.w3.org/1999/xlink"
   xml:base="http://www.example.org/airports/?ref-point=cyyz&radius=500nm&has-
iap=yes">
  <airport-ref xlink:href="cyow.xml"/>
  <airport-ref xlink:href="cyyz.xml"/>
  <airport-ref xlink:href="cyxu.xml"/>
  ...
  <next-page-link                    xlink:href="http://www.example.org/airports/?ref-
point=cyyz&radius=500nm&has-iap=yes&start=21"/>
</airport-listing>
```

Esto es un buen uso de REST, porque el recurso XML contiene su propia información de transición (como un enlace a la siguiente página). Sin embargo, es increíblemente feo. Presumiblemente, el mismo tipo de paginación podría funcionar con la colección entera cuando no hay parámetros de búsqueda, de manera que:

```
http://www.example.org/airports/
```

o

```
http://www.example.org/airports/?start=1
```

Deberían devolver las 20 primeras referencias a aeropuertos, seguidas de un link a `http://www.example.org/airports/?start=21`, que devolverá las siguientes 20 entradas y así sucesivamente. La potencia de REST y XLink está clara: es posible comenzar con una única URL y descubrir todos los recursos disponibles automáticamente, al contrario que WS-*, se realiza sin tener que tratar con voluminosas especificaciones como UDDI y WSDL.

5.8.1.4 El significado de un enlace

El verdadero corazón de REST en su sentido original (todo debe tener una URL) y su sentido más popular y amplio (HTTP + XML como alternativa a los servicios Web), es "**enlazar**". REST insiste en que cualquier información que se pueda recuperar debe tener una única dirección que podamos hacer circular, de la misma manera que podemos pasar un número de teléfono o una dirección de e-mail. Esas direcciones hacen posible enlazar recursos (páginas HTML o en el futuro, ficheros de datos XML) en la Web, de manera que las personas o los agentes software puedan descubrir páginas nuevas siguiendo enlaces desde otros existentes.

5.8.1.4.1 La vieja escuela del hipertexto

Pero, ¿Qué significa enlace? Esta cuestión afecta mucho a cualquiera que escriba software REST de propósito general, como motores de búsqueda, navegadores, o herramientas de bases de datos, que no están diseñadas para trabajar con un simple vocabulario de marcas XML. Los especialistas de hipertexto pre-HTML creían que los enlaces podían tener diferentes significados, querían proporcionar una manera de que el autor los especificara; escondidos en la sombra durante la revolución de la Web de los años 1990s, la vieja escuela trató de añadir a XLink el universalmente ignorado atributo xlink:type. ¿Necesitamos xlink:type para el procesado de datos XML en un ambiente de trabajo REST? La respuesta es que no, de hecho, si nos fijamos, enlazar un recurso externo desde un documento HTML siempre significa la misma cosa:

Es muy difícil pensar en excepciones. Por ejemplo, consideremos estos enlaces de un documento HTML:

```
<p>During the <a href="http://en.wikipedia.org/wiki/Renacimiento">Renacimiento</a>
...</p>
<img alt="Illustration of Galileo" src="galileo.jpg"/>
<script src="validate-form.js"/>
```

En cada caso, el elemento que contiene el atributo del enlace, proporciona algo de algún otro lugar. Obviamente, causa distintos comportamientos del navegador. El dibujo se

insertará en el documento mostrado automáticamente, mientras que la entrada Renacimiento de Wikipedia no lo hará. Pero en ambos casos, las cosas que se enlazan representan algo más completo: el artículo de Wikipedia sobre el Renacimiento es más completo que la frase "Renacimiento" y la imagen galileo.jpg es más completa que el texto alternativo "Ilustración de Galileo"

5.8.1.4.2 La nueva escuela XML

Exactamente los mismos principios se aplicarán a los enlaces en los ficheros de datos XML, como en este ejemplo:

```
<person xml:base="http://www.example.org/people/e40957.xml"
xmlns:xlink="http://www.w3.org/1999/xlink">
  <name>Jane Smith</name>
  <date-of-birth>1970-10-11</date-of-birth>
  <employer xlink:href="http://www.example.org/companies/acme.xml">ACME Widgets,
Inc.</employer>
  <country-of-birth xlink:href="http://www.example.org/countries/ca.xml">Canada</country-
of-birth>
</person>
```

Toda la información disponible para el nombre de la persona es la cadena "Jane Smith", y toda la información disponible para la fecha de nacimiento es la cadena "1970-10-11"; sin embargo, hay información más completa sobre el empleado en `http://www.example.org/companies/acme.xml`, y hay información más completa sobre el país de nacimiento en `http://www.example.org/countries/ca.xml`. Parece que los enlaces unidireccionales como estos que se usan en la Web siempre conducen hacia un incremento de información canónica. Si un elemento XML tiene un atributo de enlace, entonces, ¿Podemos asumir que el documento XML entero representa una versión más pequeña de la información disponible externamente por el enlace? De ser cierto, ¿Podemos ganar mucho añadiendo xlink:role a la mezcla?

5.8.1.4.3 Futuro

Si realmente enlazar resulta tan sencillo, podremos hacer grandes cosas con datos XML y REST incluso si no estamos de acuerdo con una codificación común del contenido. Esto podría ser enormemente valioso: los documentos Web tuvieron una gran acogida precisamente porque la codificación del contenido se estandarizó mediante HTML. Si pudiésemos hacer lo mismo con los documentos Web XML sin tener que forzar a todo el mundo a encajar sus datos en algo como RDF o XTM, se podría conseguir suficiente gente que lo usara para que la idea funcionase.

5.8.1.5 ¿Cuánta normalización?

¿Cuánto se deben normalizar los ficheros de datos XML devueltos por una aplicación Web REST? Por ejemplo, si una aplicación devuelve información sobre la película Sixteen Candles, ¿Debería ponerse toda la información importante en un único fichero XML como este?

```
<film>
  <title>Sixteen Candles</title>
  <director>John Hughes</director>
  <year>1984</year>
  <production-companies>
    <company>Channel Pictures</company>
    <company>Universal Pictures</company>
  </production-companies>
</film>
```

O ¿debería enlazar a diferentes documentos XML que contengan información sobre las personas, compañías y el resto de datos?

```
<film                                   xml:base="http://www.example.org/objects/014002.xml"
xmlns:xlink="http://www.w3.org/1999/xlink">
  <title>Sixteen Candles</title>
  <director xlink:href="487847.xml"/>
  <year>1984</year>
  <production-companies>
    <company xlink:href="559366.xml"/>
    <company xlink:href="039548.xml"/>
  </production-companies>
</film>
```

Presumiblemente, el servidor REST está creando la información XML desde una base de datos relacional que esté normalizada, por lo que el mantenimiento de los datos no es un problema. Aún así, cada ejemplo tiene sus desventajas.

- En el primer ejemplo, la aplicación cliente no puede estar segura de que dos películas diferentes estén referidas al mismo director o productor, o de que alguna otra pueda tener el mismo nombre. También será difícil que el servidor pueda manejar una petición PUT para actualizar la base de datos normalizada.

- En el segundo ejemplo, la aplicación cliente tendrá que realizar un gran número de peticiones GET para juntar suficiente información incluso para la aplicación más básica.

¿Es la mejor solución imitar a HTML? Los enlaces HTML normalmente incluyen una referencia a recursos externos y una breve descripción de ese recurso (por ejemplo, el texto azul en el que solemos pinchar para acceder al enlace). No hay ninguna razón por la que los ficheros XML de una aplicación REST no puedan hacer lo mismo, combinar las ventajas de las aproximaciones normalizadas y no-normalizadas, como en este ejemplo:

```
<film xml:base="http://www.example.org/objects/014002.xml"
xmlns:xlink="http://www.w3.org/1999/xlink">
  <title>Sixteen Candles</title>
  <director xlink:href="487847.xml">John Hughes</director>
  <year>1984</year>
  <production-companies>
    <company xlink:href="559366.xml">Channel Pictures</company>
    <company xlink:href="039548.xml">Universal Pictures</company>
  </production-companies>
</film>
```

Ahora, una aplicación cliente REST no necesita solicitar ficheros de datos extra solamente para encontrar el nombre del director o de la productora, pero si sabe donde debe mirar para conocer información más completa. También puede usar la URL del enlace como identificador para evitar la ambigüedad en la gente y las compañías (las que son las mismas o sólo coincide el nombre). La aproximación además les será familiar a los desarrolladores Web que decidan usar REST para solicitar datos.

5.8.1.6 La palabra “C” (contenido)

Ahora que estamos cerrando las cuestiones de diseño, debemos hablar sobre el mayor problema de REST: el contenido.

Los principios de REST nos cuentan como manejar los recursos, pero no lo que podemos hacer con ellos. Este problema no lo comparten otras aproximaciones de red: XML-RPC define lo que significa el contenido XML hasta el punto de que se puede serializar y deserializar automática e invisiblemente; SOAP permite en principio cualquier tipo de carga XML (asumiendo que se envuelve en el sobre SOAP), aunque la mayoría de la gente usa la codificación por defecto de SOAP (que también puede ser serializada y deserializada automáticamente). REST, por otra parte, es pura arquitectura y no menciona directamente al contenido. Los adeptos a REST presumen de que ya hay aplicaciones REST funcionando como son Amazon, eBay o Flickr, pero los desarrolladores rápidamente entienden que no van a conseguir ningún beneficio: cada aplicación REST requiere sus propios mecanismos, porque usa diferentes formatos de

contenido. Si usaran XML-RPC o SOAP, habría muchas librerías estándar que simplificarían el trabajo de los desarrolladores y se podría compartir el código.

En términos prácticos, ¿Es REST sólo una palabra de marketing? Los defensores de REST pueden argumentar que la carencia de una estandarización del contenido es algo beneficioso, porque hace que la arquitectura sea más flexible a la hora de tratar con cualquier tipo de recurso, desde un fichero XML hasta una imagen, un video o una página HTML. Por otra parte, la carencia de un estándar en el formato del contenido dificulta hacer cosas útiles con los recursos una vez que los hemos solicitado. Ya se han propuesto candidatos para la codificación XML estándar de REST, incluyendo RDF y XTM, aunque parece que ninguno tendrá éxito porque RDF (el líder) nunca ha funcionado correctamente con la mayoría de especificaciones basadas en XML como XQuery o XSLT.

5.8.1.6.1 Estandarizando el contenido XML de REST en bits y piezas

La alternativa es estandarizar el contenido en bits y piezas. En vez de tratar con un formato exhaustivo de codificación de datos, podemos intentar usar un estándar de marcas para los bits, que podamos usar en cualquier tipo de documentos XML. Ahora vamos a exponer algunas posibilidades:

xlink:ref y xml:id para los enlaces

Usando el atributo xlink:href es posible que programas como los "spider" localicen fácilmente los enlaces sin importar el tipo de documento. Junto con xlink:ref, xml:id permite a los enlaces apuntar a fragmentos de los documentos XML de manera sencilla.

```
<data>
  <person xml:id="dpm">
    <name>David Megginson</name>
  <person>

  <weblog>
    <title>Quoderat</title>
    <author xlink:href="#dpm"/>
  </weblog>
</data>
```

Esta característica es muy importante porque REST está basado en los enlaces. La carencia de un mecanismo estándar para los enlaces acabaría con REST incluso antes de que comenzase.

xml:base para la identificación de documentos

De manera similar, el atributo xml:base proporciona un identificador y localizador de documentos de datos XML. Un atributo xml:base adjunto al elemento principal, nos da una URL base para resolver enlaces relativos en los documentos y un identificador global para el documento en sí.

```
<data xml:base="http://www.example.org/data/foo.xml">
  ...
</data>
```

xsi:type para los tipos de datos

¿Necesitamos tipos de datos para todo en XML? El uso de esquemas externos suele ser una mala idea por razones de funcionamiento y seguridad. Si queremos crear un tipo para todo (al menos para los tipos de datos simples), debemos hacerlo en el propio documento usando algo similar al atributo xsi:type.

```
<start-date xsi:type="xsd:date">2005-02-23</start-date>
```

Hacerlo de esta manera suele ser algo inofensivo.

5.8.2 ebPML.org

ebPML es un sitio de Internet dedicado a los estándares, tecnologías y computación orientada a Servicios.

El grupo ebPML.org [12] hizo una crítica a REST en la que determinaba que no era válido para soportar aplicaciones B2B por las siguientes razones:

State Alignment: REST no ofrece ninguna pista para conseguir la alineación del estado. Eso no importa mucho en la Web, sin embargo, no funciona en los escenarios de negocios reales. Siempre que Amazon utilice una API basada en cliente/servidor, las cosas funcionarán bien porque un cliente recibiendo una respuesta es suficiente para conseguir una alineación del estado. Pero tan pronto como Amazon cambie para poder soportar escenarios de negocio más complejo, esto se convertirá en un problema más grande y SOAP será necesario.

Interacciones Peer-to-Peer: REST ofrece esencialmente un modelo cliente/servidor con tantas interacciones sin estado como sea posible entre el cliente y el servidor. Las interacciones de negocios son Peer-to-Peer, como es el caso de las definidas en OASIS/ebBP. Una interacción de Amazon involucra por lo menos a cuatro partes

(comprador, Amazon, transporte y vendedor). Esas partes necesitan compartir el nivel de contexto del estado de la interacción. Esas ideas están totalmente fuera del ámbito de REST.

Estado parcial: REST asume que la representación del estado se transfiere libremente entre el cliente y el servidor. Esto puede que sea cierto en las páginas Web, pero no es cierto en los negocios Web. Es tremendamente raro que una parte transfiera completamente el estado de un objeto de negocio y la otra parte lo adquiera y actualice su estado. En este punto es en el que se hacen necesarios nuevos verbos. Esto es lo que realmente está haciendo Amazon con los carros de la compra.

En resumen, REST no es una arquitectura válida para los servicios B2B (business to business).

5.9 Conclusiones

En este capítulo ha quedado recogida una visión actual del estilo de arquitectura REST. La base del capítulo ha sido la disertación de Roy Thomas Fielding, aunque también se han añadido algunos detalles del funcionamiento que surgieron posteriormente. En el siguiente capítulo se intentará dar una visión general del debate que ha existido en la Web entorno a la comparación entre REST y SOAP. Se expondrán los artículos más relevantes que se han escrito hasta el momento.