

Resumen Capítulos, 3, 4 y 5 del libro RESTful Web Services de O'Reilly

CAPÍTULO 3

Que hace a los servicios RESTful diferentes?

A pesar de que este es un libro acerca de los RESTful web services , la mayoría de los servicios que les he mostrado son REST-RPC híbridos como el API del.icio.us; servicios que no han terminado bien.

Es por esto que en este momento no hay muchos servicios RESTful conocidos que funcionen bien en la Web.

Introduciendo Simple Storage Service (S3)

Dos web services populares pueden responder a este llamado, el Atom Publishing Protocol (APP) y el Amazon's Simple Storage Service.

El APP más que un servicio es un conjunto de instrucciones para construir un servicio, por eso vamos a comenzar con el S3.

S3 es una forma de almacenar cualquier información , estructurada de la forma que quieras. La información puede permanecer privada o ser accesible por cualquiera vía web o un cliente de BitTorrent.

Amazon alberga el almacenamiento y el ancho de banda, y cobra por ambos.

Hay dos usos principales para S3:

Servidor de Backup

Se pueden almacenar datos a través de S3 y no dar a nadie el acceso a la misma.

En vez de comprar un disco para hacer las copia de seguridad, se le alquila el espacio en disco a Amazon.

Data Host

Se pueden almacenar los datos en S3 y dar a otros el acceso a los mismos. Amazon ofrece sus datos a través de HTTP o BitTorrent. En vez de pagar un proveedor de Internet de ancho de banda, se usa Amazon. En función de sus costos de banda ancha existentes, esto le puede ahorrar mucho dinero. Muchos web startups hoy en día utilizan S3 para compartir los archivos de datos.

A diferencia de los servicios que se han mostrado hasta ahora, S3 no se inspira en cualquier sitio web existente. La API del.icio.us se basa en el sitio web del.icio.us, y los servicios de búsqueda de Yahoo! son basados en los sitios web correspondientes, pero no hay página Web en amazon.com donde rellenar formularios HTML para subir archivos a S3. S3 es sólo para un uso programático.

Amazon ofrece librerías para Ruby, Python, Java, C # y Perl (véase <http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryID=47>). Hay también bibliotecas de terceros, como AWS::S3 de Ruby (<http://amazon.rubyforge.org/>), que incluye el shell s3sh.

Diseño Orientado a Objetos de S3

S3 está basado en 2 conceptos: S3 buckets y S3 objects.

Un objeto es un conjunto de datos identificado con un nombre que está acompañado de alguna metadata. Un bucket es un contenedor de objetos identificado con un nombre.

Un bucket es análogo al sistema de archivos de un disco duro y el objeto a uno de los archivos en ese sistema de archivos.

Es posible comparar un bucket a un directorio en un sistema de archivos, pero los directorios se pueden anidar y los buckets no. Si se desea una estructura de directorio dentro de un bucket, se debe simular uno dando a los objetos nombres como “directorio/subdirectorio/archivo-objeto”.

Unas pocas palabras acerca de los buckets

Un bucket tiene una pieza de información asociada a él, el nombre.

El nombre solo puede contener los caracteres de la A-Z, a-z , 0-9, guión bajo, punto y guión. No es recomendable usar mayúsculas en los nombres.

Como anteriormente se mencionó, los buckets no pueden contener otros buckets, solo objetos.

Cada usuario de S3 está limitado a 100 buckets, y los nombres de los mismos deben ser únicos.

Unas pocas palabras acerca de los objetos

Un objeto consta de 4 partes:

- Una referencia a su bucket
- Los datos almacenados en el objeto (S3 llama a esto el “valor”)
- Un nombre (en S3 la “clave”)
- Un conjunto de pares clave-valor de metadatos asociados al objeto. Esto por lo general es metadata, pero a veces puedo incluir valores para el standard de cabecera HTTP Content-Type y Content-Disposition.

Qué pasa si S3 fuera una biblioteca independiente?

Si S3 fuera implementado como una biblioteca de código orientado a objetos en lugar de un servicio web , habría de tener 2 clases, S3Bucket y S3Object.

Tendrían métodos getter y setter para sus objetos: S3Bucket#name, Object.value=, S3Bucket#addObject y similares.

La clase S3Bucket debería tener un método de instancia S3Bucket#getObjects que retorna una lista de instancias de S3Objects y un método de clase S3Bucket que retorne todos los buckets.

El siguiente ejemplo en Ruby muestra como debería ser la clase.

```
class S3Bucket

  # A class method to fetch all of your buckets.
  def self.getBuckets
  end

  # An instance method to fetch the objects in a bucket.
  def getObjects
  end

  ...
end

class S3Object

  # Fetch the data associated with this object.
  def data
  end

  # Set the data associated with this object.
  def data=(new_value)
  end

  ...
end
```

Recursos

Amazon expone S3 como dos web services diferentes, un servicio RESTful basado en mensajes planos HTTP y un servicio al estilo RPC basado en mensajes SOAP.

El estilo de servicio RPC expone funciones muy similares a los métodos del ejemplo anterior.

El servicio REST S3 expone toda la funcionalidad del servicio RPC, pero en lugar de hacerlo con las funciones habituales, lo hace con objetos HTTP estándar llamados recursos.

En lugar de responder a los nombres de los métodos personalizados como getObjects, un recurso responde a una o más de los seis métodos HTTP estándar: GET, HEAD, POST, PUT, DELETE y OPTIONS.

El servicio RESTful S3 ofrece tres tipos de recursos.:

- La lista de los buckets (<https://s3.amazonaws.com/>). Sólo hay un recurso de este tipo.
- Un bucket particular (<https://s3.amazonaws.com/{nombre-de-bucket}/>). Puede haber hasta 100 recursos de este tipo.
- Un objeto S3 particular dentro de un bucket (<https://s3.amazonaws.com/{nombre-de-bucket}/{nombre-de-objeto}>). Puede haber infinitos recursos de este tipo.

Cada método de la hipotética biblioteca de objetos S3 corresponde a uno de los seis métodos estándar en uno de estos 3 tipos de recursos.

El metodo getter `S3Object#name` corresponde a una solicitud GET a un recurso objeto S3 y el método setter `S3Object#value=` corresponde a una solicitud PUT en el mismo recurso. Métodos de fábrica como el `S3Bucket.getBuckets` y métodos relacionales como el `S3Bucket#getObjects` corresponden a metodos GET en la “bucket list” y recursos “bucket”.

Cada recurso expone la misma interface y trabaja de la misma manera. Para obtener el valor de un objeto se envía una solicitud GET al URI de ese objeto. Para obtener sólo los metadatos de un objeto se envía una solicitud HEAD para la misma URI. Para crear un bucket, se envía una solicitud PUT a una URI que incorpora el nombre del bucket. Para añadir un objeto a un bucket, se envía a un URI que incorpora el nombre de bucket y el nombre del objeto. Para eliminar un bucket o un objeto, se envía una solicitud DELETE a su URI.

Los diseñadores S3 no diseñaron esto. De acuerdo con la norma HTTP esto es para lo son GET, HEAD, PUT y DELETE. Estos cuatro métodos (además de POST y OPTIONS, que S3 no utiliza) son suficientes para describir todas las interacciones con los recursos de la Web. Para exponer sus programas, servicios web, no es necesario inventar nuevos vocabularios o el nombres de métodos en URI, o hacer otra cosa que pensar cuidadosamente acerca de el diseño de su recurso. Todos los servicios web REST, sin importar su complejidad, soporta las mismas operaciones básicas. Toda la complejidad reside en los recursos.

Comparando esto con la interfaz SOAP de estilo RPC S3, para obtener la lista de buckets a través de SOAP el nombre del método es `ListAllMyBuckets`. Para obtener el contenido de un bucket el nombre del método es `ListBucket`.

Con la interfaz REST, siempre es GET. En un servicio RESTful, el URI designa un objeto y los nombres de los métodos son estandarizados. Los mismos métodos funcionan de la misma manera a través de los recursos y servicios.

Códigos de respuesta HTTP.

Otra característica definitoria de una arquitectura REST es el uso de códigos de respuesta HTTP. Si se envía una solicitud a S3, y S3 lo maneja sin ningún problema, es probable que vuelva un código de respuesta HTTP 200 (“OK”). Si algo sale mal, el código de respuesta estará en el rango de 3xx, 4xx o 5xx, por ejemplo, 500 (“Internal Server Error.”). Un código de respuesta de error es una señal para el cliente que la metadata y el cuerpo no deben interpretarse como una respuesta

a la solicitud.

No es lo que el cliente preguntó, es el intento del servidor de decirle al cliente acerca de un problema. Desde que el código de respuesta no es parte del documento o de la metadata, el cliente puede ver si se produjo un error con solo mirar los 3 primeros bytes de la respuesta.

Los códigos de respuesta no son bien utilizados en la web. El navegador no muestra el código de respuesta HTTP cuando se solicita una página, porque nadie quiere ver un código numérico cuando se puede mirar el documento para ver si algo ha ido mal. Cuando se produce un error en una aplicación web, la mayoría de las aplicaciones envían 200 ("OK"), junto con un documento legible que habla sobre el error.

En la web programable es todo lo contrario. Los programas de ordenador son buenos tomando caminos diferentes en función del valor de una variable numérica, y muy malos en averiguar lo que un documento quiere decir. En ausencia de normas preestablecidas, no hay manera para que un programa pueda determinar si un documento XML contiene datos o describe un error. Los códigos de respuesta HTTP son las reglas: convenciones básicas de cómo el cliente debe acercarse a una respuesta HTTP. Porque no son parte del cuerpo o los metadatos, un cliente puede entender lo que pasó, incluso si no tiene ni idea de cómo leer la respuesta. S3 utiliza una variedad de códigos de respuesta, además de 200 ("OK") y 404 ("Not Found"). El más común es 403 ("Forbidden") que se utiliza cuando el cliente realiza una solicitud sin proporcionar las credenciales adecuadas. También utiliza algunos otros, incluyendo 400 ("Bad Request"), lo que indica que el servidor no pudo entender los datos que el cliente envió, y 409 ("Conflict"), se envía cuando el cliente intenta eliminar un bucket que no está vacío.

Un cliente S3

Las librerías de muestras de Amazon, y las contribuciones de terceros como AWS::S3, eliminan gran parte de la necesidad de bibliotecas de cliente S3 personalizadas. Vamos a usar S3 para ilustrar la teoría detrás de REST, con un ejemplo de cliente Ruby S3 y vamos a ir analizándolo. La biblioteca implementará una interfaz orientada a objetos. El resultado se verá como ActiveRecord o algún otro mapeador objeto-relacional. En lugar de hacer llamadas SQL para almacenar los datos en una base de datos, se harán peticiones HTTP para almacenar datos en el servicio S3. En lugar de dar nombres de método de recursos específicos como getBuckets y getObject, se van a usar nombres que reflejen la interfaz RESTful: get, put, y así sucesivamente. Se crea un pequeño módulo de Ruby llamado S3::Authorized, que sólo las otras clases S3 pueden incluirlo.

Ejemplo 3.3. Cliente Ruby, código inicial:

```
#!/usr/bin/ruby -w
# S3lib.rb

# Libraries necessary for making HTTP requests and parsing responses.
require 'rubygems'
require 'rest-open-uri'
require 'rexml/document'

# Libraries necessary for request signing
require 'openssl'
require 'digest/sha1'
require 'base64'
require 'uri'

module S3 # This is the beginning of a big, all-encompassing module.

  module Authorized
    # Enter your public key (Amazon calls it an "Access Key ID") and
    # your private key (Amazon calls it a "Secret Access Key"). This is
    # so you can sign your S3 requests and Amazon will know who to
    # charge.
    @@public_key = ''
    @@private_key = ''

    if @@public_key.empty? or @@private_key.empty?
      raise "You need to set your S3 keys."
    end

    # You shouldn't need to change this unless you're using an S3 clone like
    # Park Place.
    HOST = 'https://s3.amazonaws.com/'
  end
end
```

El único aspecto interesante de `S3::Authorized` es que es en donde se debe conectar las dos claves criptográficas asociadas con la cuenta de los servicios web de Amazon. Toda solicitud S3 que se realiza incluye la clave pública (Amazon lo llama "clave de acceso ID") para que Amazon puede identificarlo. Cada petición que se haga debe ser criptográficamente firmado con su clave privada (Amazon lo llama una "clave de acceso secreta") para que Amazon sepa que es realmente usted. Es privado, en el sentido de que nunca se debe revelar a nadie. Si lo hace, la persona a la que le revele la clave será capaz de realizar solicitudes a S3 y Amazon le cobrará por ello.

Bucket List

El ejemplo 3-4 muestra una clase orientada a objetos para el primer recurso, la lista de los buckets.

Se llama a la clase para este recurso S3::BucketList.

Ejemplo 3-4. S3 cliente Ruby: clase S3::BucketList

```
# The bucket list.
class BucketList
  include Authorized

  # Fetch all the buckets this user has defined.
  def get
    buckets = []

    # GET the bucket list URI and read an XML document from it.
    doc = REXML::Document.new(open(HOST).read)

    # For every bucket...
    REXML::XPath.each(doc, "//Bucket/Name") do |e|
      # ...create a new Bucket object and add it to the list.
      buckets << Bucket.new(e.text) if e.text
    end

    return buckets
  end
end
```

Explicacion XPath

Leyendo de derecha a izquierda, la expresión Xpath //Bucket/Name significa:

Encuentra todos los nombres de tag Name
Es el hijo directo de un tag bucket Bucket/
Cualquier lugar en el documento //

Ahora el archivo es un cliente de servicios web real. Si llamamos S3::BucketList#get hacemos una solicitud HTTP GET segura a <https://s3.amazonaws.com/>, que pasa a ser el URI del recurso , "una lista de los buckets.". El servicio S3 envía un documento XML que luce como el ejemplo 3-5. Esta es una representación del recurso, " lista de buckets." Es sólo un poco de información sobre el estado actual de la lista. La etiqueta **Owner** deja claro que lista de buckets es , y la etiqueta **Bucket** contiene una serie de etiquetas Bucket describiendo los buckets (en este caso, hay una etiqueta Bucket y un bucket).

Ejemplo 3.5: Una muestra de “lista de buckets”

```
<?xml version='1.0' encoding='UTF-8'?>
<ListAllMyBucketsResult xmlns='http://s3.amazonaws.com/doc/2006-03-01/'>
  <Owner>
    <ID>c0363f7260f2f5fcf38d48039f4fb5cab21b060577817310be5170e7774aad70</ID>
  >
    <DisplayName>leonardr28</DisplayName>
  </Owner>
  <Buckets>
    <Bucket>
      <Name>crummy.com</Name>
      <CreationDate>2006-10-26T18:46:45.000Z</CreationDate>
    </Bucket>
  </Buckets>
</ListAllMyBucketsResult>
```

Para efectos de esta pequeña aplicación cliente, el **Name** es el único aspecto de un bucket en el que estamos interesados. La expresión XPath //Bucket/Name da el nombre de cada bucket, que es todo lo que se necesita para crear objetos bucket.

Una cosa que falta en este documento XML son los links. El documento da el nombre de cada bucket, pero no dice nada acerca de donde pueden ser encontrados los buckets en la Web.

Bucket

Ahora, como se muestra en el Ejemplo 3-6, vamos a escribir la clase S3::Bucket, de modo que S3::BucketList.get tendrá algo para instanciar.

Ejemplo 3-6. Cliente Ruby: la clase S3::Bucket

```
# A bucket that you've stored (or will store) on the S3 application.
class Bucket
  include Authorized
  attr_accessor :name

  def initialize(name)
    @name = name
  end

  # The URI to a bucket is the service root plus the bucket name.
  def uri
    HOST + URI.escape(name)
  end

  # Stores this bucket on S3. Analagous to ActiveRecord::Base#save,
  # which stores an object in the database. See below in the
```

```
# book text for a discussion of acl_policy.
def put(acl_policy=nil)
  # Set the HTTP method as an argument to open(). Also set the S3
  # access policy for this bucket, if one was provided.
  args = {:method => :put}
  args["x-amz-acl"] = acl_policy if acl_policy

  # Send a PUT request to this bucket's URI.
  open(uri, args)
  return self
end

# Deletes this bucket. This will fail with HTTP status code 409
# ("Conflict") unless the bucket is empty.
def delete
  # Send a DELETE request to this bucket's URI.
  open(uri, :method => :delete)
end
```

Aquí hay otros dos métodos de servicio web: `S3::Bucket#put` y `S3::Bucket#delete`. Desde que el URI de un bucket lo identifica de forma exclusiva, la eliminación es simple: se envía una solicitud DELETE al URI del bucket, y se elimina. Desde que el nombre de un bucket esta en su URI, y un bucket no tiene otras propiedades configurables, también es fácil de crear: solos se envía una solicitud PUT a su URI. En `S3::Object` se ve que una solicitud PUT es más complicada cuando no todos los datos se pueden almacenar en el URI.

Anteriormente se han comparado las clases `S3` con las clases de `ActiveRecord`, pero `S3::Bucket#put` trabaja un poco diferente que una implementación **save** de `ActiveRecord`. Una fila en una tabla de base de datos de registros controlados tiene un ID único. Si se toma un objeto `ActiveRecord` con ID 23 y se cambia su nombre, su cambio se refleja como un cambio en el registro de base de datos con ID 23:

```
SET name="newname" WHERE id=23
```

El ID permanente de bucket `S3` es su URI y el URI incluye el nombre. Si cambia el nombre de un bucket y llama `put`, el cliente no renombra el viejo bucket en `S3`: crea uno nuevo vacío en un nuevo URI con el nuevo nombre. Esto es resultado de las decisiones de diseño tomadas por los programadores de `S3`.

El último método de `S3::Bucket`, que se ha llamado `get`, como `S3::BucketList.get`, este método realiza una solicitud GET a la URI de un recurso (en este caso, a un recurso bucket), recupera un documento XML, y lo parsea dentro de nuevas instancias de una clase Ruby (véase el Ejemplo 3-7). Este método es compatible con una variedad de maneras de filtrar el contenido de buckets `S3`. Por ejemplo, puede utilizar: `:Prefix` para recuperar sólo los objetos cuyas claves empiezan con una determinada cadena.

Ejemplo 3-7. cliente Ruby S3: clase S3::Bucket (terminada)

```

# Get the objects in this bucket: all of them, or some subset.
#
# If S3 decides not to return the whole bucket/subset, the second
# return value will be set to true. To get the rest of the objects,
# you'll need to manipulate the subset options (not covered in the
# book text).
#
# The subset options are :Prefix, :Marker, :Delimiter, :MaxKeys.
# For details, see the S3 docs on "Listing Keys".
def get(options={})
  # Get the base URI to this bucket, and append any subset options
  # onto the query string.
  uri = uri()
  suffix = '?'

  # For every option the user provided...
  options.each do |param, value|
    # ...if it's one of the S3 subset options...
    if [:Prefix, :Marker, :Delimiter, :MaxKeys].member? :param
      # ...add it to the URI.
      uri << suffix << param.to_s << '=' << URI.escape(value)
    end
  end
  suffix = '&'
end

# Now we've built up our URI. Make a GET request to that URI and
# read an XML document that lists objects in the bucket.
doc = REXML::Document.new(open(uri).read)
there_are_more = REXML::XPath.first(doc, "//IsTruncated").text == "true"

# Build a list of S3::Object objects.
objects = []
# For every object in the bucket...
REXML::XPath.each(doc, "//Contents/Key") do |e|
  # ...build an S3::Object object and append it to the list.
  objects << Object.new(self, e.text) if e.text
end
return objects, there_are_more
end
end

```

Xpath Explicacion

Leyendo de derecha a izquierda, la expresión Xpath //IsTruncated significa:

Buscar todos los tag IsTruncated IsTruncated
En cualquier lugar del documento //

Si se hace una solicitud GET a la raíz de la URI de la aplicación, se obtiene una representación del recurso "lista de buckets.". Al hacer una petición GET a la URI de un recurso "bucket", se obtiene una representación del bucket: un documento XML como el de ejemplo 3-8, que contiene una etiqueta Contents para cada elemento del bucket.

Example 3-8. Ejemplo de representación de bucket

```

<?xml version='1.0' encoding='UTF-8'?>
<ListBucketResult xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
  <Name>crummy.com</Name>
  <Prefix></Prefix>
  <Marker></Marker>
  <MaxKeys>1000</MaxKeys>
  <IsTruncated>>false</IsTruncated>
  <Contents>
    <Key>mydocument</Key>
    <LastModified>2006-10-27T16:01:19.000Z</LastModified>
    <ETag>"93bede57fd3818f93eedce0def329cc7"</ETag>
    <Size>22</Size>
    <Owner>
      <ID>
c0363f7260f2f5fcf38d48039f4fb5cab21b060577817310be5170e7774aad70</ID>
      <DisplayName>leonardr28</DisplayName>
    </Owner>
    <StorageClass>STANDARD</StorageClass>
  </Contents>
</ListBucketResult>

```

El objeto S3

Un objeto S3 es sólo una cadena de datos que se le ha dado un nombre (una llave) y un conjunto de pares de metadatos clave-valor (como Content-Type = "text / html"). Cuando se envía una solicitud GET a la lista de buckets, o a un bucket, S3 sirve un documento XML que hay que analizar. Cuando se envía una solicitud GET a un objeto, S3 sirve cualquier cadena de datos que puso (PUT) anteriormente byte a byte. Ejemplo 3-9 muestra el principio de un S3::objet.

Example 3-9. S3 Cliente Ruby: clase S3::Object

```

# An S3 object, associated with a bucket, containing a value and
metadata.

```

```

class Object
  include Authorized

  # The client can see which Bucket this Object is in.
  attr_reader :bucket

  # The client can read and write the name of this Object.
  attr_accessor :name

  # The client can write this Object's metadata and value.
  # I'll define the corresponding "read" methods later.
  attr_writer :metadata, :value

  def initialize(bucket, name, value=nil, metadata=nil)
    @bucket, @name, @value, @metadata = bucket, name, value, metadata
  end

  # The URI to an Object is the URI to its Bucket, and then its name.
  def uri
    @bucket.uri + '/' + URI.escape(name)
  end
end

```

Lo que sigue es mi primera aplicación de una solicitud HTTP HEAD. Yo lo uso para buscar metadatos pares clave-valor de un objeto y rellenar el hash metadatos con ella.

Ejemplo 3-10. Cliente Ruby S3: El método `S3::Object#metadata`

```

# Retrieves the metadata hash for this Object, possibly fetching
# it from S3.
def metadata
  # If there's no metadata yet...
  unless @metadata
    # Make a HEAD request to this Object's URI, and read the metadata
    # from the HTTP headers in the response.
    begin
      store_metadata(open(uri, :method => :head).meta)
    rescue OpenURI::HTTPError => e
      if e.io.status == ["404", "Not Found"]
        # If the Object doesn't exist, there's no metadata and this is not
        # an error.
        @metadata = {}
      else
        # Otherwise, this is an error.
        raise e
      end
    end
  end
end

```

```

end
return @metadata
end

```

El objetivo es buscar los metadatos de un objeto sin ir a buscar el objeto en sí. Esta distinción entre metadatos y la representación no es única a S3, y es la solución en general para todos los servicios web de recursos orientados. El método HEAD da a cualquier cliente una manera de ir a buscar los metadatos de cualquier recurso, sin también ir a buscar su representación. Se ha puesto la petición GET en el método de acceso S3::Object#value, en el Ejemplo 3-11.

Ejemplo 3-11. S3 Cliente Ruby : El metodo S3::Object#value

```

# Retrieves the value of this Object, possibly fetching it
# (along with the metadata) from S3.
def value
  # If there's no value yet...
  unless @value
    # Make a GET request to this Object's URI.
    response = open(uri)
    # Read the metadata from the HTTP headers in the response.
    store_metadata(response.meta) unless @metadata
    # Read the value from the entity-body
    @value = response.read
  end
  return @value
end

```

El cliente almacena los objetos en el servicio S3 de la misma manera en que almacena buckets: mediante el envío de una solicitud PUT a un determinado URI. El PUT bucket es trivial, ya que un bucket no tiene identidad mas que su nombre, que va a la URI de la solicitud PUT.

Un objeto PUT es más complejo. Aquí es donde el cliente HTTP especifica una metadata de objetos (como Content-Type) y el valor. Esta información estará disponible en el futuro para HEAD y peticiones GET.

Afortunadamente, la creación de la petición PUT no es muy complicada, ya que el valor de un objeto es lo que el cliente dice que es. No hay que envolver el valor del objeto en un documento XML.

Ejemplo 3-12. Cliente Ruby S3: El metodo S3::Object#put

```

# Store this Object on S3.
def put(acl_policy=nil)
  # Start from a copy of the original metadata, or an empty hash if
  # there is no metadata yet.
  args = @metadata ? @metadata.clone : {}

```

```
# Set the HTTP method, the entity-body, and some additional HTTP
# headers.
args[:method] = :put
args["x-amz-acl"] = acl_policy if acl_policy
if @value
  args["Content-Length"] = @value.size.to_s
  args[:body] = @value
end

# Make a PUT request to this Object's URI.
open(uri, args)
return self
end
```

Implementacion de S3::Object#delete (ver Ejemplo 3-13) es idéntico al S3::Bucket#delete.
Ejemplo 3-13. Cliente Ruby S3: El método S3::Object#delete

```
# Deletes this Object.
def delete
  # Make a DELETE request to this Object's URI.
  open(uri, :method => :delete)
end
```

El ejemplo 3-14 muestra el método para convertir los encabezados de respuesta HTTP en metadatos de un objeto S3. A excepción de Content-Type, debe aparecer delante todos los encabezados de metadatos que se ha establecido la cadena "x-amz-meta-". De lo contrario no van a hacer el viaje de vuelta al servidor S3 y de nuevo a un cliente de servicios web.

Ejemplo 3-14. Cliente Ruby S3: El método privado
S3::Object#store_metadata

```
# Given a hash of headers from a HTTP response, picks out the
# headers that are relevant to an S3 Object, and stores them in the
# instance variable @metadata.
def store_metadata(new_metadata)
  @metadata = {}
  new_metadata.each do |h,v|
    if RELEVANT_HEADERS.member?(h) || h.index('x-amz-meta') == 0
      @metadata[h] = v
    end
  end
end

RELEVANT_HEADERS = ['content-type', 'content-disposition',
  'content-range',
  'x-amz-missing-meta']
end
```

Solicitud de firma y control de acceso

Ahora es el momento para hacer frente a la autenticación de S3.

El código que se ha mostrado hasta ahora hace peticiones HTTP bien, pero S3 los rechaza ya que no contienen la cabecera de Autorización que es de suma importancia. S3 no tiene pruebas que usted es el dueño de sus propios buckets. Se recuerda que Amazon cobra por el almacenamiento de los datos en sus servidores y el ancho de banda utilizado en la transferencia de dichos datos. Si S3 acepta solicitudes a sus buckets sin autorización, cualquier persona puede almacenar datos en sus buckets y luego usted tendría que pagar por ello.

La mayoría de los servicios web que requieren autenticación utilizan un mecanismo HTTP estándar para asegurarse de que usted es quien dice ser. Sin embargo, las necesidades de S3 son más complicadas. Como la mayoría de los servicios web no se quiere que cualquiera utilice sus datos. Uno de los usos de S3 es ofrecer un servicio de alojamiento. Es posible que desee servir una película en S3, y que cualquiera la pueda descargar con su cliente de BitTorrent.

O podría ser la venta de acceso a los archivos de películas almacenados en S3. Su e-commerce obtiene el pago del cliente y le da una URI S3 que se puede utilizar para descargar la película. Así se está delegando a un tercero el derecho a llamar un servicio web en particular (una solicitud GET) como tú, y luego cargarlo a tu cuenta.

Los mecanismos estándar para la autenticación HTTP no pueden garantizar seguridad para este tipo de aplicación. Normalmente, la persona que está enviando la petición HTTP necesita saber la contraseña real. Cada vez que haga una solicitud a S3, se utiliza su llave "privada" para firmar las partes importantes de la solicitud. Eso sería la URI, el método HTTP que está usando, y algunas de las cabeceras HTTP. Sólo alguien con la key "privada" puede crear estas firmas para sus peticiones. Pero una vez que se ha firmado una solicitud, se puede enviar la firma a un tercero sin revelar su clave "privada". La tercera parte es libre de enviar una solicitud HTTP idéntica a la que tu firmaste.

El modulo Ruby S3::Authorized va a tener la capacidad para interceptar las llamadas al método *open*, y firmar las peticiones HTTP antes de que sean hechas. Desde que S3::Bucketlist, S3::Bucket y S3::Object han incluido este módulo, van a heredar esta capacidad.

Ejemplo 3-15. Cliente Ruby S3: El modulo Authorized S3::Authorized

```
module Authorized
  # These are the standard HTTP headers that S3 considers interesting
  # for purposes of request signing.
  INTERESTING_HEADERS = ['content-type', 'content-md5', 'date']

  # This is the prefix for custom metadata headers. All such headers
  # are considered interesting for purposes of request signing.
  AMAZON_HEADER_PREFIX = 'x-amz-'

  # An S3-specific wrapper for rest-open-uri's implementation of
  # open(). This implementation sets some HTTP headers before making
```

```
# the request. Most important of these is the Authorization header,
# which contains the information Amazon will use to decide who to
# charge for this request.
def open(uri, headers_and_options={}, *args, &block)
  headers_and_options = headers_and_options.dup
  headers_and_options['Date'] ||= Time.now.httpdate
  headers_and_options['Content-Type'] ||= ''
  signed = signature(uri, headers_and_options[:method] || :get,
    headers_and_options)
  headers_and_options['Authorization'] = "AWS #{@public_key}:#{signed}"
  Kernel::open(uri, headers_and_options, *args, &block)
end
```

El trabajo complicado está en la firma del método, aún no definido. Este método necesita construir una cadena cifrada para entrar en la cabecera de una petición de autorización.

Ejemplo 3-16. Cliente Ruby S3 : el modulo Authorized#signature

```
# Builds the cryptographic signature for an HTTP request. This is
# the signature (signed with your private key) of a "canonical
# string" containing all interesting information about the request.
def signature(uri, method=:get, headers={}, expires=nil)
  # Accept the URI either as a string, or as a Ruby URI object.
  if uri.respond_to? :path
    path = uri.path
  else
    uri = URI.parse(uri)
    path = uri.path + (uri.query ? "?" + query : "")
  end
  # Build the canonical string, then sign it.
  signed_string = sign(canonical_string(method, path, headers, expires))
end
```

Resulta una solicitud HTTP en una cadena que se ve como el Ejemplo 3-17. El dato interesante es el método HTTP (PUT), el Content-Type ("text / plain"), una fecha, unas cuantas cabeceras HTTP ("x-amz-metadata"), y la parte de la URI ("/crummy.com/myobject"). Esta es la cadena que hay que firmar.

Ejemplo 3-17. muestra de canonical string

```
PUT
text/plain
Fri, 27 Oct 2006 21:22:41 GMT
x-amz-metadata:Here's some metadata for the myobject object.
/crummy.com/myobject
```

Cuando el servidor de Amazon recibe la solicitud HTTP, que genera la cadena canónica, la firma y ve si las dos

firmas coinciden. Así es como funciona la autenticación S3. Si las firmas coinciden, la solicitud sigue adelante. De lo contrario, se obtiene un código de respuesta 403 ("Forbidden").

Ejemplo 3-18 muestra el código para generar el canonical string.

Ejemplo 3-18. Cliente Ruby S3: el metodo `Authorized#canonical_string`

```
# Turns the elements of an HTTP request into a string that can be
# signed to prove a request comes from your web service account.
def canonical_string(method, path, headers, expires=nil)

  # Start out with default values for all the interesting headers.
  sign_headers = {}
  INTERESTING_HEADERS.each { |header| sign_headers[header] = '' }

  # Copy in any actual values, including values for custom S3
  # headers.
  headers.each do |header, value|
    if header.respond_to? :to_str
      header = header.downcase
      # If it's a custom header, or one Amazon thinks is interesting...
      if INTERESTING_HEADERS.member?(header) ||
        header.index(AMAZON_HEADER_PREFIX) == 0
        # Add it to the header has.
        sign_headers[header] = value.to_s.strip
      end
    end
  end

  # This library eliminates the need for the x-amz-date header that
  # Amazon defines, but someone might set it anyway. If they do,
  # we'll do without HTTP's standard Date header.
  sign_headers['date'] = '' if sign_headers.has_key? 'x-amz-date'

  # If an expiration time was provided, it overrides any Date
  # header. This signature will be valid until the expiration time,
  # not only during the single second designated by the Date header.
  sign_headers['date'] = expires.to_s if expires

  # Now we start building the canonical string for this request. We
  # start with the HTTP method.
  canonical = method.to_s.upcase + "\n"

  # Sort the headers by name, and append them (or just their values)
```

```

# to the string to be signed.
sign_headers.sort_by { |h| h[0] }.each do |header, value|
  canonical << header << ":" if header.index(AMAZON_HEADER_PREFIX) == 0
  canonical << value << "\n"
end

# The final part of the string to be signed is the URI path. We
# strip off the query string, and (if neccessary) tack one of the
# special S3 query parameters back on: 'acl', 'torrent', or
# 'logging'.
canonical << path.gsub(/\?.*$/, '')

for param in ['acl', 'torrent', 'logging']
  if path =~ Regexp.new("[&?]{#{param}}($|&|=)")
    canonical << "?" << param
    break
  end
end
return canonical
end

```

Ejemplo 3-19. Cliente Ruby S3: el metodo Authorized#sign

```

# Signs a string with the client's secret access key, and encodes the
# resulting binary string into plain ASCII with base64.
def sign(str)
  digest_generator = OpenSSL::Digest::Digest.new('sha1')
  digest = OpenSSL::HMAC.digest(digest_generator, @@private_key, str)
  return Base64.encode64(digest).strip
end

```

La firma de un URI

S3 permite firmar una petición HTTP y dar la URI a otra persona, dejando que hagan la solicitud como si fueran usted. El método que permite hacer esto es `signed_uri`. En lugar de hacer una petición HTTP con `open`, se pasan los argumentos `open` dentro de este método, y le devuelve un URI firmado que cualquiera puede utilizar. Para limitar el abuso, un URI firmado sólo funciona durante un tiempo limitado. Se puede personalizar ese tiempo pasando un objeto `Time` con un argumento `:expires`.

Ejemplo 3-20. Cliente Ruby S3: el metodo Authorized#signed_uri

```

# Given information about an HTTP request, returns a URI you can
# give to anyone else, to let them make that particular HTTP
# request as you. The URI will be valid for 15 minutes, or until the

```

```
# Time passed in as the :expires option.
def signed_uri(headers_and_options={})
  expires = headers_and_options[:expires] || (Time.now.to_i + (15 * 60))
  expires = expires.to_i if expires.respond_to? :to_i
  headers_and_options.delete(:expires)
  signature = URI.escape(signature(uri, headers_and_options[:method],
  headers_and_options, nil))
  q = (uri.index("?") ? "&" : "?")
  "#{uri}#{q}Signature=#{signature}&Expires=#{expires}&AWSAccessKeyId=#{@@
public_key}"
end
end
end
```

Remember the all-encompassing S3 module? This is the end.

He aquí cómo funciona. Supongamos que yo quiero dar a un cliente acceso a mi archivo albergado en <https://s3.amazonaws.com/BobProductions/KomodoDragon.avi>. Puedo ejecutar el código en el Ejemplo 3-21 para generar un URI para mi cliente.

Example 3-21. Generando una URI firmada

```
#!/usr/bin/ruby1.9
# s3-signed-uri.rb
require 'S3lib'
bucket = S3::Bucket.new("BobProductions")
object = S3::Object.new(bucket, "KomodoDragon.avi")
puts object.signed_uri
# "https://s3.amazonaws.com/BobProductions/KomodoDragon.avi
# ?Signature=J%2Fu6kxT3j0zHaFXjsLbowgpzExQ%3D
# &Expires=1162156499&AWSAccessKeyId=0F9DBXKB5274JKTJ8DG2"
```

Esta URI será válida durante 15 minutos, el valor predeterminado para mi aplicación signed_uri. Incorpora mi clave pública (AWSAccessKeyId), la fecha de caducidad (Expires) y la firma (Signature) criptográfica. El cliente puede visitar este URI y descargar el archivo de película. Amazon me va a cobrar por el uso de mi cliente de su ancho de banda. Si mi cliente modifica cualquier parte de la URI, el servicio S3 rechazará su solicitud.

Configuración de la directiva de acceso

¿Y si quiero hacer un objeto accesible al público? Quiero servir a mis archivos al mundo y dejar que Amazon resuelva la administración de servidores. Esto se logra permitiendo el acceso anónimo.

Usted puede hacer esto mediante el establecimiento de la política de acceso a un bucket o un objeto, diciéndole a S3 que responda sin firmar las solicitudes.

Esto se hace mediante el envío de la cabecera x-amz-acl junto con el PUT que crea el bucket o el objeto.

Eso es lo que hacer el argumento acl_policy en Bucket#put y Object#put. Si usted desea hacer un

bucket o un objeto , publico o escribible, se pasa un valor apropiado por `acl_policy`. Mi cliente envía ese valor como parte del encabezado personalizado de la solicitud HTTP `X-amz-acl`. Amazon S3 lee este encabezado de la solicitud y establece las reglas para que el bucket o objeto accedan apropiadamente.

El cliente en el Ejemplo 3-22 crea un objeto S3 que cualquiera puede leer visitando el URI en `https://s3.amazonaws.com/BobProductions/KomodoDragon-Trailer.avi`.

Ejemplo 3-22. Creando un objeto `public-read`

```
#!/usr/bin/ruby -w
# s3-public-object.rb
require 'S3lib'
bucket = S3::Bucket.new("BobProductions")
object = S3::Object.new(bucket, "KomodoDragon-Trailer.avi")
object.put("public-read")
```

S3 comprende cuatro políticas de acceso:

`private`

El valor por defecto. Sólo peticiones firmadas por su llave "privada" son aceptadas.

`public-read`

Solicitudes GET sin firmar se aceptan: cualquier persona puede descargar un objeto o una lista bucket.

`public-write`

Se aceptan peticiones GET y PUT sin firmar. Cualquier persona puede modificar un objeto o agregar objetos a un bucket.

`authenticated-read`

Peticiones sin firmar se rechazaron, pero las solicitudes de lectura pueden ser firmados por la clave "privada" de cualquier usuario S3, no sólo la suya. Básicamente, cualquier persona con una cuenta de S3 puede descargar su objeto o listar los buckets.

Usando la librería de cliente S3

Voy a mostrar una librería de cliente en Ruby, que puede acceder a casi todas las capacidades del servicio S3 de Amazon.

El Ejemplo 3-23 es un cliente S3 de línea de comandos simple que puede crear un bucket y un objeto, luego lista el contenido de los buckets.

Ejemplo 3-23. Un ejemplo de cliente S3

```
#!/usr/bin/ruby -w
# s3-sample-client.rb
require 'S3lib'

# Gather command-line arguments
bucket_name, object_name, object_value = ARGV
unless bucket_name
  puts "Usage: #{ $0 } [bucket name] [object name] [object value]"
  exit
end

# Find or create the bucket.
buckets = S3::BucketList.new.get
# GET /
bucket = buckets.detect { |b| b.name == bucket_name }
if bucket
  puts "Found bucket #{bucket_name}."
else
  puts "Could not find bucket #{bucket_name}, creating it."
  bucket = S3::Bucket.new(bucket_name)
  bucket.put
  # PUT /{bucket}
end

# Create the object.
object = S3::Object.new(bucket, object_name)
object.metadata['content-type'] = 'text/plain'
object.value = object_value
object.put
# PUT /{bucket}/{object}

# For each object in the bucket...
bucket.get[0].each do |o|
  # GET /{bucket}
  # ...print out information about the object.
  puts "Name: #{o.name}"
  puts "Value: #{o.value}"
end
```

```
# GET /{bucket}/{object}
puts "Metadata hash: #{o.metadata.inspect}"
puts
end
```

Creando un Webservice sencillo

El web service va a ser un simple block de notas: una manera de enviarme notas todo el tiempo a mi mismo. Tengo Rails 1.2 instalado en mi pc, asique puedo correr el servicio de la siguiente manera

```
$ rails notebook
```

```
$ cd notebook
```

Cree una base de datos en mi pc llamada notebook_development, y edite el archivo *notebook/config/database.yml* Para darle a Rails la información necesaria para que se conecte a la base.

Ahora creare una aplicación Rails, y generare el código para un simple Webservice RESTful con scaffold_resource generator. Quiero que mis notas contengan un timestamp y un texto, así que voy a correr el siguiente comando:

```
$ ruby script/generate scaffold_resource note date:date body:text
create app/views/notes
create app/views/notes/index.rhtml
create app/views/notes/show.rhtml
create app/views/notes/new.rhtml
create app/views/notes/edit.rhtml
create app/views/layouts/notes.rhtml
create public/stylesheets/scaffold.css
create app/models/note.rb
create app/controllers/notes_controller.rb
create test/functional/notes_controller_test.rb
create app/helpers/notes_helper.rb
create test/unit/note_test.rb
create test/fixtures/notes.yml
create db/migrate
create db/migrate/001_create_notes.rb
route map.resources :notes
```

Rails ha generado un set completo de código (modelo, vista, controlador) para mi objeto “nota”. En db/migrate/001_create_notes.rb encontramos código que crea una tabla llamada notas en la base de datos con 3 campos (ID único, fecha y un campo de texto).

El código generado en `app/models/note.rb` provee una interfaz activa a la tabla de la base de datos. El controlador en `app/controllers/notes_controller.rb` expone esa interfaz al mundo a través de HTTP, y las vistas en `app/views/notes` definen la interfaz de usuario.

Antes de iniciar el WS, inicializamos la base de datos:

```
$ rake db:migrate
== CreateNotes: migrating
=====
-- create_table(:notes)
-> 0.0119s
== CreateNotes: migrated (0.0142s)
=====
```

Ahora puedo activar la aplicación de la pc y comenzar a usar el servicio.

```
$ script/server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
```

Un cliente ActiveResource

La aplicación que acabo de generar no es mucho más que una demo, pero esta demo nos muestra algunas cosas bastante interesantes. Primero que nada es una aplicación y un servicio web. Puedo visitar <http://localhost:3000/notes> en un navegador, y crear notas a través de la interfaz web. Después de un rato la página <http://localhost:3000/notes> se verá como esta:

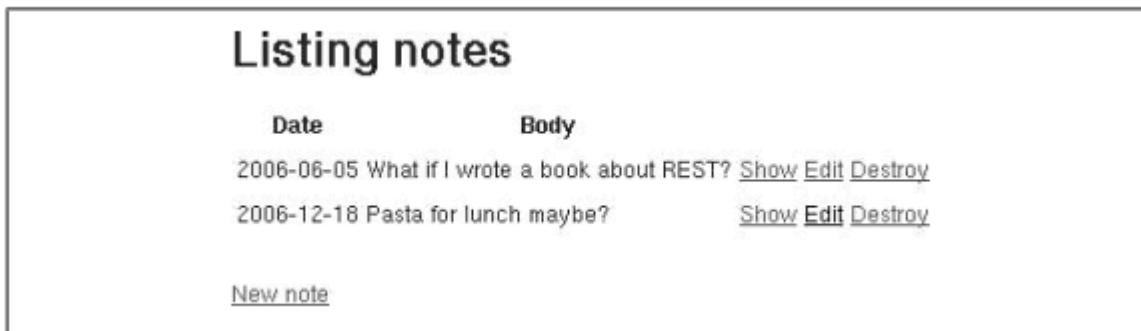


Figure 3-1. La aplicación de notas, con unas pocas notas.

Si alguna vez haz escrito o visto una aplicación Rail esto te debe sonar familiar, pero en Rails 1.2 el modelo y controlador generados pueden actuar también como un RESTful Webservice.

Tanto un cliente programado, como un navegador pueden acceder fácilmente.

Desafortunadamente, el cliente ActiveResource en sí mismo no fue puesto en libertad junto con Rails 1.2.

En este momento todavía está siendo desarrollado; Para obtener el código tenemos que descargar el repositorio del svn

```
$ svn co http://dev.rubyonrails.org/svn/rails/trunk activeresource_client
$ cd activeresource_client
```

Ahora estamos listos para escribir el cliente ActiveResource para el webservice.

El ejemplo 3-24 es un cliente que crea una nota, la modifica, lista las notas existentes y borra la nota que acaba de crear.

Ejemplo 3-24.

```
#!/usr/bin/ruby -w
# activeresource-notebook-manipulation.rb
require 'activesupport/lib/active_support'
require 'activeresource/lib/active_resource'
# Define a model for the objects exposed by the site
class Note < ActiveResource::Base
  self.site = 'http://localhost:3000/'
end
def show_notes
```

```

notes = Note.find :all # GET /notes.xml
puts "I see #{notes.size} note(s):"
notes.each do |note|
puts "  #{note.date}: #{note.body}"
end
end
new_note = Note.new(:date => Time.now, :body => "A test note")
new_note.save # POST /notes.xml
new_note.body = "This note has been modified."
new_note.save # PUT /notes/{id}.xml
show_notes
new_note.destroy # DELETE /notes/{id}.xml
puts
show_notes

```

El ejemplo 3-25 Muestra la salida cuando corro el programa:

Ejemplo 3-25

```

I see 3 note(s):
  2006-06-05: What if I wrote a book about REST?
  2006-12-18: Pasta for lunch maybe?
  2006-12-18: This note has been modified.
I see 2 note(s):
  2006-06-05: What if I wrote a book about REST?
  2006-12-18: Pasta for lunch maybe?

```

Si estas familiarizado con ActiveRecord, el mapeo de objetos que conecta a Rails con la base de datos te darás cuenta que la interfaz de ActiveRecord es prácticamente igual. Ambas librerías proveen una interfaz orientada a objetos.

Con ActiveRecord, el objeto vive en una base de datos y es expuesto a través de SQL, con SELECT, INSERT, UPDATE y DELETE. Con ActiveRecord los objetos viven en una aplicación Rails y son expuestos a través de HTTP con los métodos GET, POST, PUT y DELETE.

El ejemplo 3-26 es un extracto de los registros del servidor Rails en el momento en que corrí mi cliente ActiveRecord. Las peticiones GET, POST, PUT y DELETE corresponden a las líneas de código comentadas en el Ejemplo 3-24.

Ejemplo 3-26. El pedido HTTP hecho por activereource-notebook-manipulation.rb

```

"POST /notes.xml HTTP/1.1" 201
"PUT /notes/5.xml HTTP/1.1" 200
"GET /notes.xml HTTP/1.1" 200
"DELETE /notes/5.xml HTTP/1.1" 200
"GET /notes.xml HTTP/1.1" 200

```

¿Qué está pasando en estas peticiones? Lo mismo que pasa en las peticiones a S3: accedemos a los recursos a través de la interfaz uniforme de HTTP. Mi servicio expone dos tipos de recursos:

- La lista de notas (/notes.xml). Similar a un bucket S3, que es una lista de objetos.
- Una nota (/notes/{id}.xml). Similar a un objeto S3.

Estos recursos se exponen con GET, PUT y DELETE, al igual que los recursos S3. La lista de notas también soporta POST para crear una nueva nota. Eso es un poco diferente de S3, donde los objetos se crean con PUT.

Cuando se ejecuta el cliente, los documentos XML se transfieren de manera invisible entre el cliente y el servidor. Se parecen a los documentos del Ejemplo 3-27 o 3-28: representaciones simples de las subyacentes filas de la base de datos.

Ejemplo 3-27. Respuesta a un pedido GET /notes.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<notes>
  <note>
    <body>What if I wrote a book about REST?</body>
    <date type="date">2006-06-05</date>
    <id type="integer">2</id>
  </note>
  <note>
    <body>Pasta for lunch maybe?</body>
    <date type="date">2006-12-18</date>
    <id type="integer">3</id>
  </note>
</notes>
```

Ejemplo 3-28. Pedido enviado como parte de una petición PUT a /notes/5.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <body>This note has been modified.</body>
</note>
```

Cliente Python para un Webservice sencillo

En este momento la única biblioteca de cliente ActiveResource es la biblioteca de Ruby y Rails es el único framework que expone servicios compatibles con ActiveResource. Pero no pasa nada aquí excepto las peticiones HTTP que pasan documentos XML en determinado URI y obtienen documentos XML de vuelta. No hay ninguna razón para que un cliente en otro lenguaje no pueda enviar los documentos XML, o qué otro framework no pueda exponer los mismos URIs.

El ejemplo 3-29 es un cliente Python para el ejemplo 3-24. Es más largo que el programa de Ruby, porque no confía en ActiveResource. Entonces tiene que crear sus propios documentos XML y hacer sus propias peticiones HTTP, pero su estructura es casi exactamente la misma.

Ejemplo 3-29. Cliente Python para un servicio ActiveResource

```
#!/usr/bin/python
# activeresource-notebook-manipulation.py

from elementtree.ElementTree import Element, SubElement,
    tostring
from elementtree import ElementTree
import httplib2
import time
BASE = "http://localhost:3000/"
client = httplib2.Http(".cache")

def showNotes():
    headers, xml = client.request(BASE + "notes.xml")
    doc = ElementTree.fromstring(xml)
    for note in doc.findall('note'):
        print "%s: %s" % (note.find('date').text, note.find('body').text)

newNote = Element("note")
date = SubElement(newNote, "date")
date.attrib['type'] = "date"
date.text = time.strftime("%Y-%m-%d", time.localtime())
body = SubElement(newNote, "body")
body.text = "A test note"

headers, ignore = client.request(BASE + "notes.xml", "POST",
    body=tostring(newNote),
    headers={'content-type' : 'application/xml'})

newURI = headers['location']

modifiedBody = Element("note")

body = SubElement(modifiedBody, "body")
body.text = "This note has been modified"

client.request(newURI, "PUT",
    body=tostring(modifiedBody),
    headers={'content-type' : 'application/xml'})
```

```
showNotes()  
client.request(newURI, "DELETE")  
  
print  
showNotes()
```

Palabras de despedida

Como los webServices RESTful tienen interfaces simples y bien definidas, no es difícil clonarlos o intercambiar una implementación por otra. Park Place (<http://code.whytheluckystiff.net/Parkplace>) es una aplicación hecha en Ruby que expone la misma HTTP como interfaz S3. Podemos usar Park Place para alojar nuestra propia versión del S3. Las librerías y clientes S3 trabajarán contra nuestro servidor Place Park al igual que lo hacen ahora contra <https://s3.amazonaws.com/>.

Mientras tanto, la escritura de un cliente único ActiveResource-compatible no es más difícil que escribir un cliente para cualquier otro servicio RESTful.

A estas alturas ya se debe sentir cómodo con la idea de escribir un cliente para cualquier servicio RESTful o REST-RPC híbrido, si sirve XML, HTML, JSON, o alguna mezcla. Son todas solicitudes HTTP y parseo de documentos.

También debe tener idea de lo que diferencia a los servicios web RESTful S3 (ej: servicio de búsqueda de Yahoo!) de servicios de tipo RPC e híbridos como Flickr y la API de del.icio.us. Esto no es un juicio sobre el contenido del servicio, sólo sobre su arquitectura. En el trabajo de la madera es importante trabajar con la veta de la madera. En la Web, los webservice RESTful son una de las vetas importantes.

En los próximos capítulos vamos a ver cómo se pueden crear servicios Web que se parezcan más a S3 y menos a la API de del.icio.us. Esto culmina en el capítulo 7, que reinventa del.icio.us como un servicio web RESTful.

CAPITULO 4

Arquitectura Orientada a Recursos (ROA)

Hemos mostrado el poder de REST, pero no hemos mostrado de forma sistemática como este se estructura, o la forma de exponerlo. En este capítulo describimos la estructura concreta de un WebService RESTful: la arquitectura orientada a recursos (ROA). La ROA es una manera de convertir un problema en un WebService RESTful: un arreglo de URIs, HTTP y XML que funcionan como el resto de la web, y que los programadores podrán disfrutar de usar.

En el capítulo 1 clasificamos webServices RESTful por sus respuestas a dos preguntas. Estas respuestas corresponden a dos de las cuatro características que definen un servicio REST:

- La información de alcance ("¿por qué debe el servidor enviar estos datos en lugar de aquellos datos? ") se mantiene en el URI. Este es el principio de direccionamiento.
- La forma de la información ("¿por qué el servidor debe enviar estos datos en lugar de eliminarlos? ") se mantiene en el método HTTP. Hay sólo unos pocos métodos HTTP, y todo el mundo sabe de antemano lo que hacen. Este es el principio de la interfaz uniforme.

En este capítulo presentaremos las partes móviles de la arquitectura orientada a recursos: recursos (por supuesto), sus nombres, sus representaciones, y los vínculos entre ellos.

Explicaremos y promoveremos las propiedades de la ROA: direccionamiento, la conexión, y la interfaz uniforme. Mostraremos cómo las tecnologías de la web (HTTP, URI, y XML) implementan las piezas móviles para hacer estas propiedades posibles.

En los capítulos anteriores hemos ilustrado conceptos señalando a los servicios web existentes, como S3. Seguimos la tradición en este capítulo, pero también vamos a ilustrar conceptos señalando sitios web existentes. Esperamos haberte convencido ya de que los sitios web son servicios web, y que muchas aplicaciones web (como los motores de búsqueda) son webServices RESTful. Cuando hablamos de conceptos abstractos como direccionamiento, es útil para mostrarnos URIs reales, que se pueden escribir en el navegador para ver los conceptos en acción.

Recursos orientados, y ahora?

¿Por qué llegar a un nuevo término, 'Arquitectura orientada a recursos'? ¿Por qué no decir REST?

La Arquitectura orientada a recursos también es un servicio RESTful. Pero REST no es una arquitectura: es un conjunto de criterios de diseño. Se puede decir que una arquitectura cumple los criterios mejor que la otra, pero no hay ninguna "arquitectura REST."

Hasta ahora, la gente ha tendido en mente una única arquitectura para el diseño de sus servicios, de acuerdo a su propia comprensión de REST. El resultado más evidente de esto es la gran variedad de RPC en servicios web REST híbridos que sus creadores afirman que son RESTful. Estamos tratando de poner fin a esto presentando una serie de normas concretas para la construcción de servicios web que realmente van a ser RESTful. En los dos capítulos siguientes Incluso te mostraremos sencillos procedimientos que pueden seguir para convertir las necesidades en recursos. Si no te gustan estas reglas, por lo menos tendrás una idea de lo que puedes cambiar sin perder la idea de RESTful.

Son un conjunto de criterios de diseño, REST es muy general. En particular, no está ligado a la

Web. Nada acerca de REST depende de la mecánica de HTTP o de la estructura de URIs. Pero estamos hablando de los servicios web, así que me ato explícitamente a la arquitectura orientada a recursos de las tecnologías de la Web. Vamos a hablar acerca de cómo hacer REST con HTTP y URIs, en lenguajes de programación específicos. Si en el futuro se producen arquitecturas RESTful que no se ejecuten en la parte superior de la Web, sus mejores prácticas probablemente sean similares a la ROA, pero los detalles serán diferentes. Cruzaremos ese puente cuando lleguemos a él. La definición tradicional de REST deja mucho espacio abierto, que los profesionales tienen sembrado con el folclore.

Que es un recurso?

Un recurso es cualquier cosa que sea lo suficientemente importante como para ser referenciado como una cosa en sí misma. Si los usuarios desearan "crear un enlace de hipertexto a algo, hacer o refutar afirmaciones sobre algo, recuperar o almacenar en caché una representación de algo, anotar o realizar otras operaciones sobre un elemento", entonces usted debe convertir es 'algo' en un recurso.

Por lo general, un recurso es algo que se puede almacenar en un ordenador y representar como una corriente de bits: un documento, una fila en una base de datos, o el resultado de ejecutar un algoritmo.

Un recurso puede ser un objeto físico como una manzana o un concepto abstracto como el valor, pero (como veremos más adelante) las representaciones de estos recursos están destinados a ser decepcionante.

Aquí están algunos recursos posibles:

- La versión 1.0.3 liberada del software
- La última versión liberada del software
- Un mapa de carreteras de Little Rock, Arkansas
- Parte de la información sobre las medusas
- Un directorio de los recursos existentes en las medusas
- El siguiente número primo después de 1024
- Los próximos cinco números primos después de 1024
- Los números de ventas para Q42004
- La relación entre dos conocidos, Alice y Bob
- Una lista de los informes de error de la base de datos de bug

URIs

¿Qué hace que un recurso sea un recurso? Tiene que tener al menos un URI. El URI es el nombre y la dirección de un recurso. Si una parte de la información no tiene un URI, entonces no es un recurso y no está realmente en la Web, es solo bits de datos que describen algún otro recurso.

Vamos a suponer que esto es una petición HTTP 0.9 para <http://www.example.com/hello.txt> :

Solicitud del cliente

GET /hello.txt

Respuesta del servidor

Hola, mundo!

Un cliente HTTP manipula un recurso mediante la conexión con el servidor que lo aloja (en este caso, www.example.com), y enviando al servidor un método ("GET") y una ruta de acceso al recurso ("/hello.txt"). Hoy HTTP 1.1 es un poco más complejo que 0.9, pero funciona de la misma manera. Tanto el servidor y la ruta provienen de URI del recurso.

Solicitud del cliente

GET /hello.txt HTTP/1.1

Host: www.example.com

Respuesta del servidor

200 OK

Content-Type: text/plain

Hello, world!

El URI es la tecnología fundamental de la Web. Había sistemas de hipertexto antes del HTML, y protocolos de Internet antes de HTTP, pero no se comunicaban entre sí. El URI interconecta todos estos protocolos de Internet en un sitio Web, la forma TCP/IP interconecta redes como Usenet, Bitnet y CompuServe en un solo Internet. Entonces la Web adoptó estos protocolos y a los demás los mató, al igual que Internet hizo con las redes privadas.

La web mata a otros protocolos debido a que tiene algo que la mayoría de los protocolos carecen: una manera sencilla de etiquetar todos los elementos disponibles. Cada recurso en la Web tiene al menos un URI. Usted puede pegar un URI en una cartelera.

La gente puede ver ese cartel, escribir esa URI en sus navegadores web, e ir derecho al recurso que quiera mostrarles. Puede parecer extraño, pero esta interacción cotidiana era imposible antes de que el URI se inventara.

Las URIs deben ser descriptivas

Este es el primer punto en el que se basa la ROA para realizar las recomendaciones sobre los servicios REST. Propongo que un recurso y su URI deben tener una correspondencia intuitiva. Aquí están algunas buenas URIs para los recursos que se enumeran más arriba:

- <http://www.example.com/software/releases/1.0.3.tar.gz>
- <http://www.example.com/software/releases/latest.tar.gz>
- <http://www.example.com/weblog/2006/10/24/0>
- http://www.example.com/map/roads/USA/AR/Little_Rock
- <http://www.example.com/wiki/Jellyfish>
- <http://www.example.com/search/Jellyfish>
- <http://www.example.com/nextprime/1024>
- <http://www.example.com/next-5-primes/1024>

- <http://www.example.com/sales/2004/Q4>
- <http://www.example.com/relationships/Alice;Bob>
- <http://www.example.com/bugs/by-state/open>

Las URI deben tener una estructura. Estas deben variar de manera predecible: no hay que ir a '/search/Medusas' por medusas y '/i-want-to-know-about/Mice' para ratones. Si el cliente conoce la estructura del URI del servicio, puede crear sus propios puntos de entrada en el servicio. Esto hace que sea fácil para los clientes utilizar su servicio de maneras que no se habían pensado. Esto no es una regla absoluta de REST, como veremos en la sección "Nombre de Recursos". Las URIs técnicamente no tienen que tener ninguna estructura o la previsibilidad, pero sería bueno que así fuera. Esta es una de las reglas de un buen diseño web, y eso se nota en REST y los servicios híbridos REST-RPC.

La relación entre URIs y recursos

Vamos a considerar algunos casos extremos. ¿Pueden dos recursos ser el mismo? ¿Pueden dos URI designar al mismo recurso? ¿Puede un solo URI designar dos recursos?

Por definición, dos recursos no pueden ser el mismo. Si fueran el mismo, entonces sólo tendríamos un recurso. Sin embargo, en algún momento en el tiempo dos recursos diferentes pueden compartir los mismos datos. Si la versión de software actual es 1.0.3, entonces

<http://www.example.com/software/releases/1.0.3.tar.gz> y

<http://www.example.com/software/releases/latest.tar.gz> harán referencia al mismo archivo por un tiempo. Pero la idea detrás de cada URI es diferente: una de ellas siempre apunta a una versión en particular, y la otra a la versión más reciente del producto. Esto son dos conceptos y dos recursos distintos.

Un recurso puede tener uno o varios URIs. Por ejemplo las cifras de ventas disponibles en

<http://www.example.com/sales/2004/Q4> también pueden estar disponibles en

<http://www.example.com/sales/Q42004>. Si un recurso tiene múltiples URIs, es más fácil que los clientes vean el recurso.

La desventaja es que cada URI adicional diluye el valor de todas las demás.

Algunos clientes utilizan una URI, algunos utilizan otras, y no hay forma automática para verificar que todos los URI se refieren a un mismo recurso.

Una forma de evitar esto es exponer múltiples URI para el mismo recurso, pero que uno de ellos sea el URI "canónico" de este recurso.

Cuando un cliente solicita el URI canónico, el servidor envía los datos apropiados, junto con código de respuesta 200 ("OK"). Cuando un cliente solicita otra URI, el servidor envía un código de respuesta 303 ("Vea también"), junto con la URI canónica. El cliente no puede determinar si dos URIs apuntan al mismo recurso, pero puede hacer dos peticiones HEAD y ver si una URI redirige a la otra o si ambas se redirigen a una tercer URI.

Otra forma es responder a todos los URI como si fueran el mismo, pero dar la URI "canónica" en el encabezado en respuesta cada vez que alguien solicita una URI no canónica.

Usar './sales/2004/Q4' podría retornar el mismo bytestream que usar './sales/Q42004', porque son diferentes URIs para el mismo recurso: "las ventas para el último trimestre de 2004."

También usar `.../releases/1.0.3.tar.gz` podría retornar exactamente el mismo `bytestream` que al usar `.../releases/latest.tar.gz`, aunque sean diferentes recursos, ya que representan diferentes casos: "Versión 1.0.3" y "la última versión."

Cada URI identifica exactamente un recurso. Si identificara más de uno, no sería un identificador universal de recursos. Sin embargo, al buscar una URI el servidor puede enviarle información sobre varios recursos: la que ha solicitado y otras relacionadas. Cuando entramos a una página web, por lo general esta transmite cierta información de ella misma, pero también tiene enlaces a otras páginas web. Al recuperar un bucket S3 con un cliente de Amazon S3, se obtiene un documento que contiene información sobre el bucket, y información sobre los recursos relacionados con los objetos en el bucket.

Direccionamiento

Ahora que hemos presentado a los recursos y sus URIs, podemos profundizar en dos de las características del ROA: direccionamiento y la statelessness.

Una aplicación es direccionable si expone los aspectos interesantes de su conjunto de datos como recursos.

Dado que los recursos están expuestos a través de URIs, una aplicación direccionable expone un URI para cada pieza de información que posiblemente podría servir. Esto suele ser un número infinito de URIs.

Desde la perspectiva del usuario final, el direccionamiento es el aspecto más importante de cualquier sitio o aplicación web. Los usuarios son ingeniosos, y van a tratar de corregir o rodear casi cualquier deficiencia, si los datos son lo suficientemente interesantes; pero es muy difícil solucionar la falta de direccionamiento.

Considere una URI real que da nombre a un recurso en el "Directorio de recursos sobre el género jellyfish": <http://www.google.com/search?q=jellyfish>. Esta búsqueda de medusas se parece bastante a esta otra URI (también real) <http://www.google.com>.

Si HTTP no fuera direccionable, o si el motor de búsqueda de Google no fuera una aplicación web direccionable, no seríamos capaces de publicar esta URI en un libro. Tendríamos que decir: "Abra una conexión web a google.com, escriba 'jellyfish' en el cuadro de búsqueda y haga clic en el botón 'Buscar con Google'."

Esto no era una preocupación académica hasta mediados de la década de los 90, cuando <ftp://URI> se hizo popular para describir los archivos en sitios FTP, entonces la gente tenía que escribir cosas como: "Inicio de una sesión de FTP anónimo en ftp.example.com ahora cambie al directorio pub .../files/ y descargue archivo.txt".

Las URI hicieron a FTP tan direccionable como HTTP. Ahora la gente debe escribir: "Download <ftp://ftp.example.com/pub/files/archivo.txt>". Los pasos son los mismos, pero ahora pueden ser llevados a cabo por la máquina.

Pero HTTP y Google son ambos direccionables, así que podemos imprimir esa URI en un libro. Usted puede leerla y escribirla en el navegador. Cuando lo haga llegará al mismo lugar que si se accediera desde la aplicación de google. También puedes marcar esta página y volver a ella más tarde. Podemos poner un link a esta página en nuestra propia aplicación web, podemos enviar la URI a otra persona, etc. Si HTTP no fuera direccionable, tendríamos que descargar toda la página y enviar el archivo HTML como archivo adjunto.

Para ahorrar ancho de banda, puede configurar un proxy-caché HTTP en la red local. La primera

vez que alguien solicita <http://www.google.com/search?q=jellyfish>, la caché guardara una copia local del documento. La próxima vez que alguien solicite esta URI, la memoria caché podría mostrar la copia guardada en lugar de descargarla de nuevo. Estas cosas son posibles sólo si cada página tiene una cadena de identificación única: una dirección.

Es incluso posible que la URI utilice una URI como entrada a otra. Podemos usar un webService externo para validar el HTML de una página, o para traducir el texto a otro idioma. Estos webServices esperan una URI como entrada. Si HTTP no fuera direccionable, no habría forma de decirles a estos con que recursos tienen que operar.

El servicio S3 de Amazon es direccionable por cada bucket y cada objeto tiene su propia URI. Buckets y objetos que aún no existen también tienen su propia URI: se puede crear un recurso mediante el envío de una solicitud PUT a su URI.

El sistema de ficheros en el ordenador de casa es otro sistema direccionable. De la línea de comandos las aplicaciones pueden tomar una ruta a un archivo y hacer cosas extrañas a la misma. Las celdas de una hoja de cálculo también son direccionables, se puede conectar el nombre de una celda en una fórmula y la fórmula utilizará cualquier valor que se encuentra en esa celda.

Las URIs serían los path a los archivos y nombres de celdas en la WEB.

La direccionabilidad es una de las mejores cosas acerca de las aplicaciones web. Esto hace que sea fácil para los clientes utilizar los sitios web de manera que los diseñadores originales nunca imaginaron.

Esta es la razón por la que los servicios REST-RPC son tan comunes: combinan direccionamiento con la programación del modelo “procedimiento en guardia”.

Los recursos son el tipo de cosa que es direccionable.

Esto parece natural, la forma en que la Web debe trabajar. Desafortunadamente, muchas aplicaciones web no funcionan de esta manera. Esto es especialmente cierto sobretodo en las aplicaciones ajax. Como veremos en el Capítulo 11, la mayoría de las aplicaciones ajax son sólo clientes de webServices RESTful o híbridos.

Sin embargo, cuando se utilizan estos clientes como si fueran sitios web, notamos que no se sienten como sitios web.

No hay necesidad de meterse con los pequeños, vamos a continuar nuestro recorrido por las propiedades de Google, teniendo en cuenta el servicio de correo electrónico Gmail en línea.

Desde la perspectiva del usuario final, sólo hay una URI Gmail: <https://mail.google.com/>. Hagas lo que hagas, toda la información que envías o recibes de Gmail, nunca verás una URI diferente. El recurso "mensajes de correo electrónico acerca de jellyfish" no es direccionable, sin embargo el camino "web de Google páginas sobre jellyfish" sí lo es. Sin embargo, detrás de las escenas, como vemos en el capítulo 11, es un sitio web que es direccionable. La lista de los mensajes de correo electrónico acerca de jellyfish tiene una URI: es <https://mail.google.com/mail/?q=jellyfish&search=query&view=tl>. El problema es que no eres el consumidor real de ese sitio web. El sitio web es en realidad un servicio web y el consumidor real es un programa JavaScript que se ejecuta dentro del navegador web. El servicio web de Gmail es direccionable, pero la aplicación web de Gmail que utiliza no lo es.

Statelessness

La direccionabilidad es una de las cuatro características principales de la ROA. La segunda es statelessness. Vamos a dar dos definiciones de statelessness: una definición algo general y una más práctica orientada hacia el ROA.

Statelessness significa que cada petición HTTP ocurre en completo aislamiento. Cuando el cliente realiza una petición HTTP, esta incluye toda la información necesaria para que el servidor pueda cumplir con esa solicitud. El servidor nunca se basa en información de solicitudes anteriores. Si la información era importante, el cliente la habría enviado de nuevo en esta solicitud.

Más prácticamente, considere el statelessness en términos de direccionamiento. La direccionabilidad dice que cada pieza interesante de información que el servidor puede proporcionar debe ser expuesta como un recurso, y tener su propia URI. El statelessness dice que los estados posibles del servidor también son recursos, y se les debe dar su propio URI. El cliente no debe tener que convencer al servidor en un cierto estado para que sea receptivo a una determinada petición.

En la red humana, a menudo nos encontramos con situaciones en las que el botón “Atrás” de su navegador no funciona correctamente y no se puede ir adelante ni atrás en el historial del navegador; a veces esto se debe a que hemos realizado una acción irrevocable, como una publicación en un blog o la compra de un libro, pero a menudo es porque estás en un sitio web que viola el principio del statlessness. Este sitio espera que usted haga una solicitud en un cierto orden: A, B, C. Usted se confunde al hacer solicitud B por segunda vez en lugar de pasar a solicitud de C.

Tomemos el ejemplo de búsqueda de nuevo. Un motor de búsqueda es un servicio web con un número infinito de posibles estados: al menos uno por cada cadena que podría buscar. Cada estado tiene su propio URI. Podemos solicitar el servicio de un directorio de recursos sobre ratones (mice): <http://www.google.com/search?q=mice>. Podemos solicitar un directorio de recursos sobre las medusas (jellyfish) : <http://www.google.com/search?q=jellyfish>. Si no te sentís cómodo creando una URI desde cero, puedes solicitar el servicio rellenando un formulario de google. Cuando solicitamos un directorio de recursos sobre ratones o medusas, no recibimos el directorio completo. Recibimos una página del directorio; una lista de los 10 o X artículos que el motor de búsqueda considera son los mejores resultados para nuestra consulta. Para obtener más del directorio debemos efectuar más solicitudes HTTP. La segunda y siguientes páginas son distintos estados de la solicitud, y ellos necesitan tener su propio URI: algo así como <http://www.google.com/search?q=jellyfish> [HYPERLINK "http://www.google.com/search?q=jellyfish&start=10"](http://www.google.com/search?q=jellyfish&start=10) [HYPERLINK "http://www.google.com/search?q=jellyfish&start=10"](http://www.google.com/search?q=jellyfish&start=10) [start=10](http://www.google.com/search?q=jellyfish&start=10). Al igual que con cualquier recurso direccionable, que pueda transmitir ese estado de la solicitud a otra persona, caché, o un marcador y volver a ella más tarde.

La Figura 4-1 es un diagrama de estado simple que muestra cómo un cliente HTTP puede interactuar con cuatro estados de un motor de búsqueda.

Se trata de una aplicación sin estado, porque cada vez que el cliente realiza una solicitud, termina de vuelta en donde comenzó. Cada solicitud está totalmente desconectada de las otras. El cliente puede realizar solicitudes relacionadas con estos recursos cualquier cantidad de veces, y en cualquier orden. Se puede solicitar la página 2 de "ratones" antes de solicitar la página 1, y al servidor no le molestara.

A modo de contraste, la Figura 4-2 muestra los mismos estados dispuestos, con los estados que lleva sensiblemente entre sí. La mayoría de las aplicaciones de escritorio están diseñadas de esta manera.

Esto es mucho mas organizado, y si HTTP fuera diseñado para permitir interacciones con estado, las peticiones HTTP podrían ser mucho más simples. Cuando el cliente inicia una sesión en el motor de búsqueda este puede alimentar de forma automática al buscador. No tiene que enviar los datos de petición en absoluto, ya que la primera respuesta podría estar predeterminada. Si el cliente estaba mirando a las 10 primeras entradas en el directorio de ratones y quería ver las entradas de 11 a 20, sólo tendría que enviar una solicitud de que "start = 10". No tendría que enviar /search?q=mice&start=10, repitiendo todos los parámetros.

FTP funciona de esta manera: tiene una noción de un "directorio de trabajo" que se mantiene constante en el transcurso de una sesión a menos que se cambie. Es posible conectarse a un servidor FTP, hacer cd a un cierto directorio, y obtener un archivo de ese directorio. Usted puede obtener otro archivo del mismo directorio, sin tener que emitir una segunda orden cd. ¿Por qué HTTP no soporta esto?

El estado hace que las solicitudes HTTP individuales sean muy simples, pero esto haría que el protocolo HTTP fuera mucho más complicado. Un cliente FTP es mucho más complicado que un cliente HTTP, precisamente porque el estado de la sesión se debe mantener en sincronización entre el cliente y el servidor. Esta es una tarea compleja, incluso en una red confiable, que la Internet no es.

Eliminar el estado de un protocolo significa eliminar una gran cantidad de condiciones de fallo. El servidor no tiene que preocuparse por la sincronización con el cliente, porque no hay interacciones que duren más que una única solicitud:

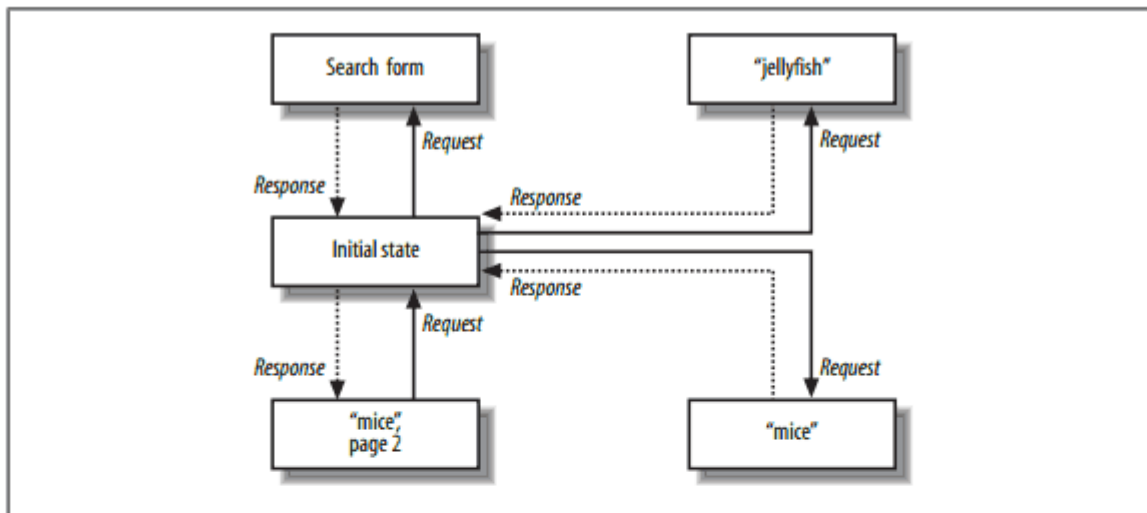


Figure 4-1. A stateless search engine

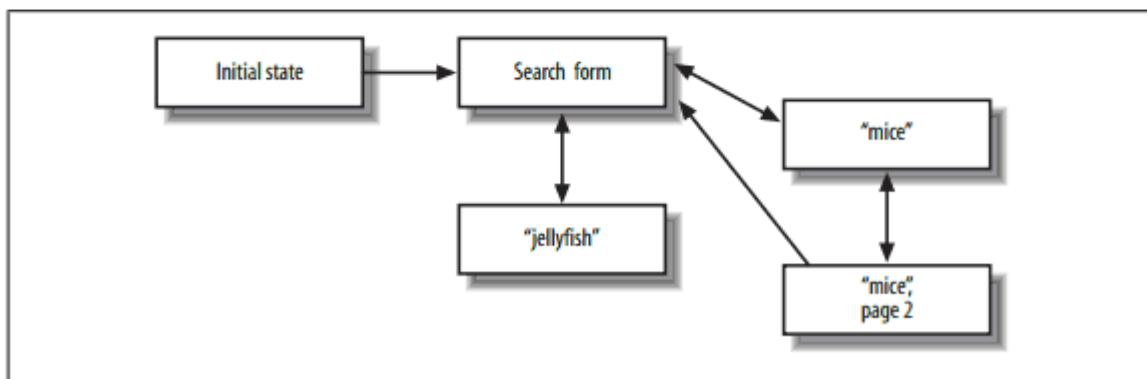


Figure 4-2. A stateful search engine

El servidor nunca pierde la noción de "donde" esta cada cliente en la aplicación, ya que el cliente envía toda la información necesaria con cada solicitud. El cliente nunca termina de realizar una acción en un mal "directorio de trabajo" debido a que el servidor mantiene un estado ahí sin decirle al cliente.

Statelessness también trae nuevas características. Es más fácil de distribuir una aplicación sin estado a través de servidores con equilibrio de carga. Como no hay solicitudes que dependan de otras, estas pueden ser manejadas por dos servidores diferentes que nunca coordinen con los otros. Una aplicación sin estado es fácil de almacenar en el caché: una pieza de software puede decidir si desea o no almacenar en el caché el resultado de una solicitud HTTP con sólo mirar esa petición. El cliente se beneficia de ser stateless también. Un cliente puede procesar el directorio "ratones" desde la página 50, buscando `/search?q=mice&start=50`, y volver una semana más tarde sin tener que navegar a través de docenas de estados anteriores. Un URI que funciona cuando estás horas inmerso en una sesión HTTP funcionará de la misma manera la primera vez que los envías en una nueva sesión.

Para hacer nuestro servicio direccionable tenemos que realizar cierto trabajo, debemos diseccionar los datos de la aplicación en conjuntos de recursos.

HTTP es un protocolo sin estado intrínsecamente, por lo que al escribir servicios web, se obtiene el statelessness por defecto. Tenemos que hacer algo para romperlo, la forma más común de

romper el statelessness es utilizar la versión de su marco de sesiones HTTP. La primera vez que un usuario visita un sitio web, este recibe una cadena única que identifica su sesión en el sitio. La cadena puede ser mantenida en una cookie, o el sitio puede propagar una cadena única a través de todos los URIs que le sirven a un cliente en particular. He aquí una cookie de sesión establecida por una aplicación Rails:

Set-Cookie: _session_id=c1c934bbe6168dcb904d21a7f5644a2d; path=/

Esta URI propaga el identificador de sesión en una aplicación PHP:

<http://www.example.com/forums?PHPSESSID=27314962133>.

Lo importante es que el número hexadecimal sin sentido no es el Estado. Es una llave en una estructura de datos en el servidor, y esta estructura de datos contiene el estado. No hay nada que no sea RESTful sobre una URI con estado: así es como el servidor le comunica los posibles próximos estados al cliente. Sin embargo, hay algo que no es RESTful en las cookies, como explicamos en "El problema con las cookies." Para usar una analogía, las cookies rompen el botón 'Atrás' de un cliente de servicios web.

Piense en la petición variable start=10 de una URI, incrustada en una página HTML devuelta por el motor de búsqueda de Google. Ese es el servidor enviando un posible estado próximo para el cliente.

Pero estos URIs necesitan tener el estado, no sólo proporcionan una clave para obtener el estado almacenado en el servidor. start=10 significa algo en sí mismo, y PHPSESSID=27314962133 no. RESTful requiere que el estado permanezca en el lado del cliente, y se transmita al servidor en cada solicitud que lo necesite. El servidor puede cambiar el estado del cliente mediante el envío de enlaces con estado para que el cliente siga, pero no puede mantener un estado propio.

Decidiendo entre Representaciones

Si un servidor ofrece múltiples representaciones de un recurso, ¿cómo averiguar el que el cliente está pidiendo?

Por ejemplo, un comunicado de prensa puede exponerse en Inglés y Español.

Cuál es el que hay que darle al cliente?

Hay un número de maneras de resolver esto dentro de las limitaciones de REST. La más simple, y la que el autor recomienda para la arquitectura orientada a recursos, es dar una URI distinta a cada representación de un recurso.

<http://www.example.com/releases/104.en> puede designar la representación de Inglés de la nota de prensa y <http://www.example.com/releases/104.es> podrían designar la representación en español.

El autor recomienda esta técnica para aplicaciones ROA porque la URI contiene toda la información necesaria para el servidor para atender la solicitud.

La desventaja de exponer múltiples URIs para el mismo recurso es que puede generarse confusión ya que hay varios artículos expuestos de diferentes formas que hablan de lo mismo y el consumidor puede pensar que hay múltiples informaciones de un mismo tema y no la misma información expuesta de forma diferente.

La forma alternativa se llama negociación de contenido. En este escenario, la URI sólo es expuesta en la forma platónica, <http://www.example.com/releases/104>, cuando un cliente realiza

una solicitud de esa URI, proporciona cabeceras especiales en la petición HTTP que indican qué tipo de representaciones el cliente está dispuesto a aceptar.

Su navegador Web tiene un ajuste con las preferencias de idiomas en las que prefiere obtener las páginas web. El navegador presenta esta información con cada petición HTTP, en el cabezal "Accept-Language". El servidor suele ignorar esta información porque la mayoría de las páginas web están disponibles en un solo idioma. Pero encaja con lo que estamos tratando de hacer aquí: exponer representaciones diferentes del mismo recurso. Cuando un cliente llama a <http://www.example.com/releases/104>, el servidor puede decidir si se debe seleccionar la representación en Inglés o en español del recurso basado en la información que viene en el cabezal.

El motor de búsqueda de Google es un buen lugar para probar esto. Usted puede obtener resultados en casi todos los idiomas cambiando el idioma del navegador o manipulando la variable de consulta *hl* en la URI (por ejemplo, *hl = tr* de Turquía). El motor de búsqueda soporta ambos contenidos de negociación y diferentes URIs para diferentes representaciones.

Un cliente también puede establecer el encabezado el formato de archivo que prefiera para las representaciones.

Un cliente puede decir que prefiere XHTML a HTML, SVG o de cualquier otro formato gráfico. El servidor permite usar cualquiera de estas solicitudes de metadatos y decidir que representación enviar. Otros tipos de solicitud de metadatos incluyen información de pago, credenciales de autenticación, directivas de almacenamiento en caché y demás. Todo esto podría hacer una diferencia en las decisiones del servidor de qué datos incluir en la representación, el idioma y el formato que debe usar e incluso denegar el acceso.

Es RESTful para mantener esta información en las cabeceras HTTP, y es RESTful para ponerlo en el URI. El autor recomienda mantener la mayor cantidad de información posible en la URI, y tan poco como sea posible en la solicitud de metadatos. El autor cree que las URIs son más útiles que los metadatos.

Las URIs van pasando de persona a persona y de programa a programa. la solicitud de metadatos casi siempre se pierde en la transición.

He aquí un ejemplo simple de este dilema: el validador HTML del W3C, un servicio web disponible en <http://validator.w3.org/>. Aquí hay una URI a un recurso en el sitio del W3C, un reporte de validación, la versión en Inglés de un comunicado de prensa hipotético:

<http://validator.w3.org/check?uri=http%3A%2F%2Fwww.example.com%2Freleases%2F104.en>. Aquí hay otro recurso: un informe de validación de la versión española del comunicado de prensa:

<http://validator.w3.org/check?uri=http%3A%2F%2Fwww.example.com%2Freleases%2F104.es>.

Cada URI en tu sitio se convierte en un recurso en la aplicación W3C, sin importar si lo es o no se designa un recurso distinto en el sitio. Si su comunicado de prensa tiene un URI diferente para cada representación, se pueden obtener dos recursos del W3C: informes de validación de la versión en español y de la versión en Inglés.

Pero si sólo se expone la forma platónica de URI, para servir a ambas representaciones desde esa URI, sólo se puede obtener un recurso de la W3C. Eso sería un informe de validación de la versión predeterminada de la nota de prensa (probablemente el Inglés). Uno no tiene forma de saber si hay o no errores de formato HTML en la representación en español. Si el servidor no expone el comunicado de prensa española con su propia URI, no va a estar el recurso disponible en el sitio W3C. Esto no quiere decir que no se puede exponer esa forma platónica de URI: sólo que no debe ser el único URI que utilice.

A diferencia de los humanos, los programas de computadora son muy malos en el trato con las representaciones que no esperan. El autor cree que un cliente web automatizado debe ser lo más explícito posible sobre la representación que se quiere. Esto casi siempre significa especificar una representación en la URL.

Enlaces y conectividad

A veces, las representaciones no son más que estructuras de datos serializados. Están destinados a ser utilizados y se descartan. Pero en la mayoría de los servicios REST, las representaciones son hipermedia: documentos que no contienen sólo datos, sino enlaces a otros recursos.

Tomemos el ejemplo de búsqueda de nuevo. Si vamos al directorio de documentos sobre medusas (<http://www.google.com/search?q=jellyfish>) de Google, verá algunos resultados de búsqueda, y un conjunto de enlaces internos a otras páginas. La Figura 4-3 muestra una muestra representativa de la página.

En la página hay datos y enlaces. Los datos dicen que en algún lugar en la Web, alguien dijo tal y tal cosa acerca de las medusas, con énfasis en dos especies de medusas hawaianas. Los enlaces dan acceso a otros recursos: algunos dentro de la búsqueda de Google "servicio web", y algunos otros lugares en la web:

- La página web externa que habla de las medusas: <http://www.aloha.com/~lifeguards/jellyfish.html>. El punto principal de este servicio web, por supuesto, es el de presentar enlaces de este tipo.
- Un enlace a un cache proporcionado por Google de la página externa. Estos enlaces siempre tienen una URI extensa que apuntan a direcciones IP del propietario (Google), como <http://209.85.165.104/search?q=cache:FQrLzPU0tKQJ>
- Un enlace a un directorio de páginas que Google piensa que están relacionados con la página externa (<http://www.google.com/search?q=related:www.aloha.com/~lifeguards/jellyfish.html>, como "Páginas similares"). Este es otro caso de un servicio web que toma una URI como parámetro de entrada.
- Un conjunto de vínculos de navegación que te llevan a diferentes páginas del directorio "medusas": <http://www.google.com/search?>

`q=jellyfish&start=10,http://www.google.com/search?q=jellyfish&start=20`, y así sucesivamente.

Al principio de este capítulo, me mostró lo que podría suceder si HTTP fuera un protocolo *statful* como FTP. La Figura 4-2 muestra las rutas que un cliente HTTP *stateful* puede tomar durante una "sesión" en `www.google.com`. HTTP en realidad no funciona de esa manera, pero esa imagen muestra de buena manera como usamos los humanos la web. Para utilizar un motor de búsqueda empezamos llenando un formulario en la página principal para hacer una búsqueda, y hacemos clic en los enlaces para ir a las siguientes páginas de resultados. No escribir una URI tras otra para obtener resultados: seguimos vínculos y rellenamos formularios. Si usted ha leído acerca de REST antes, es posible que haya encontrado un axioma de la Fielding disertación: "hipermedia como el motor de estado de la aplicación." Esto es lo que ese axioma significa: el estado actual de una "sesión" HTTP no se almacena en el servidor como un estado de los recursos, pero es visto por el cliente como un estado de la aplicación, y creado por el camino que el cliente lleva a través de la Web. El servidor dirige la trayectoria del cliente por servir "hipermedia": vínculos y formas dentro de las representaciones de hipertexto.

El servidor envía las directrices del cliente acerca de qué estados están cerca de la actual. El enlace "Siguiente" en `http://www.google.com/search?q=jellyfish` es un cambio de estado: se le muestra cómo llegar desde el estado actual a otro relacionado. Esto es muy poderoso. Un documento que contiene un URI apunta a otro posible estado de la aplicación: "página dos," o "relacionado con este URI," o ". Una versión en caché de la URI" O puede estar apuntando a un posible estado de una aplicación totalmente diferente.

La web es fácil de usar para los humano, ya que está bien conectada. Cualquier usuario con experiencia sabe cómo escribir URI en la barra de direcciones del navegador, y la forma de saltar de un sitio a otro mediante la modificación de la URI, pero muchos usuarios lo hacen todo navegando desde un único punto de inicio de Internet: la página de inicio del navegador establecido por su ISP. Esto es posible debido a que la Web está muy bien comunicada. Las páginas se vinculan entre sí, incluso a través de los sitios.

Pero la mayoría de los servicios web no están conectados internamente, y mucho menos conectados entre sí. Amazon S3 es un servicio web RESTful que es direccionable y sin estado, pero no conectado. Representaciones S3 no son los URI. Para conseguir un bucket S3, usted tiene que conocer las reglas para la construcción de URI del bucket. No se puede obtener la lista de buckets y seguir un enlace al bucket que desee.

Ejemplo 4-1 muestra una lista de bucket S3 que he cambiado para que esté conectado. Comparar con el Ejemplo 3-5, que no tiene ninguna etiqueta URI. Esta es sólo una forma de introducir la URI en una representación XML. Dado que los recursos se vuelven mejor conectados, las relaciones entre ellos se hace más evidente (ver Figura 4-4).

Example 4-1. A connected "list of your buckets"

```
<?xml version='1.0' encoding='UTF-8'?>
```

```

<ListAllMyBucketsResult xmlns='http://s3.amazonaws.com/doc/2006-03-01/'>
<Owner>
<ID>c0363f7260f2f5fcf38d48039f4fb5cab21b060577817310be5170e7774aad70</ID>
<DisplayName>leonardr28</DisplayName>
</Owner>
<Buckets>
<Bucket>
<Name>crummy.com</Name>
<URI>https://s3.amazonaws.com/crummy.com</URI>
<CreationDate>2006-10-26T18:46:45.000Z</CreationDate>
</Bucket>
</Buckets>
</ListAllMyBucketsResult>

```

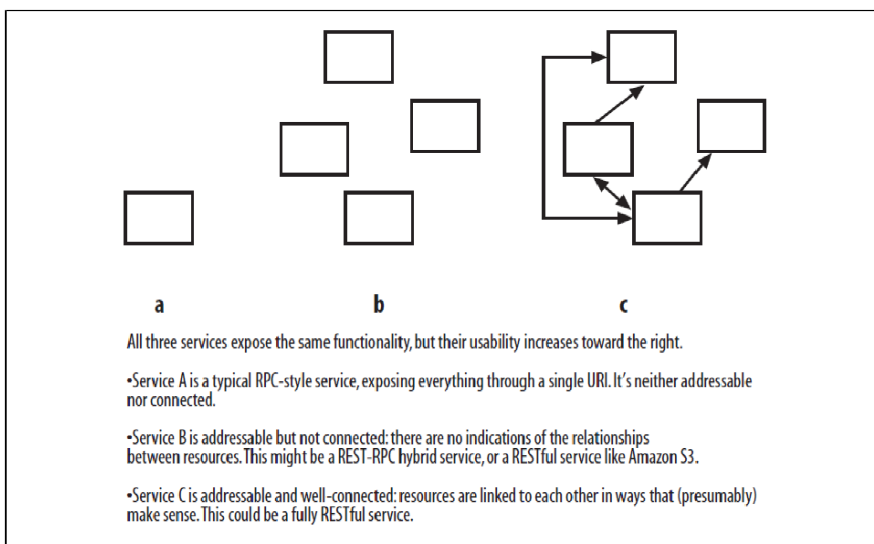


Figure 4-4. One service three ways

La interfaz uniforme

Hay algunas cosas básicas que usted puede hacer sobre un recurso. HTTP proporciona cuatro métodos básicos para las cuatro operaciones más comunes:

- Recuperar una representación de un recurso: HTTP GET
- Crear un nuevo recurso: HTTP PUT a una nueva URI, o HTTP POST a un URI existente (consulte la sección "POST" más abajo)
- Modificar un recurso existente: HTTP PUT a una URI existente
- Eliminar un recurso existente: HTTP DELETE

Voy a explicar cómo se utilizan estos cuatro para representar casi cualquier operación que se pueda imaginar. También me ocuparé de dos métodos HTTP para dos operaciones menos comunes: HEAD y OPTIONS.

GET, PUT y DELETE

Estos tres deben ser familiares para usted en el ejemplo S3 en el capítulo 3. Para recuperar o eliminar un recurso, el cliente simplemente envía una solicitud GET o DELETE a su URI. En el caso de una petición GET, el servidor devuelve una representación en la respuesta de la entidad y el cuerpo. Para obtener una solicitud DELETE, la respuesta de la entidad-cuerpo puede contener un mensaje de estado, o nada en absoluto.

Para crear o modificar un recurso, el cliente envía una solicitud PUT que generalmente incluye un cuerpo de la entidad. El cuerpo de la entidad contiene la propuesta de nueva representación del cliente del recurso. ¿Qué datos son, y en qué formato se encuentra? Depende del servicio. Sea lo que parece, este es el punto en el que el estado de aplicación se mueve en el servidor y se convierte en el estado del recurso.

Una vez más, pensar en el servicio S3, donde hay dos tipos de recursos que se pueden crear: buckets y objetos. Para crear un objeto, se envía una solicitud de PUT a su URI e incluye el contenido del objeto de la entidad-cuerpo de la convocatoria. Usted hace lo mismo para modificar un objeto: el nuevo contenido sobrescribe cualquier contenido anterior.

La creación de un bucket es un poco diferente, ya que no tiene que especificar un cuerpo de la entidad en la solicitud PUT. Un bucket no tiene ningún estado de los recursos a excepción de su nombre, y el nombre es parte de la URI.

Sin embargo, todos los objetos de S3 es un recurso propio, así que no hay necesidad de manipular un objeto a través de su bucket. Cada objeto expone la interfaz uniforme y se puede manipular por separado.

Las peticiones PUT para la mayoría de los recursos no incluyen un cuerpo de la entidad que contiene una representación, pero como se puede ver no es un requisito.

HEAD y OPTIONS

Hay otros tres métodos HTTP que considero parte de la interfaz uniforme. Dos de ellos son simples métodos de utilidad, así que los voy a cubrir en primer lugar.

- Obtener una representación sólo de metadatos: HTTP HEAD
- Compruebe que los métodos HTTP soporta un recurso en particular: OPTION HTTP

Vimos el método HEAD expuesto por los recursos del servicio S3 en el capítulo 3. Un cliente de S3 utiliza HEAD para buscar metadatos de un recurso sin tener que descargar posiblemente la enorme entidad del cuerpo. Un cliente puede utilizar HEAD para comprobar si existe un recurso, o averiguar otra información sobre el recurso, sin ir a buscar toda su representación. HEAD le da exactamente lo que una petición GET te daría, pero sin el cuerpo de la entidad.

Hay dos métodos HTTP estándar que no cubren en este libro:

TRACE y CONNECT. TRACE se utiliza para depurar proxies, y CONNECT se utiliza para reenviar algún otro protocolo a través de un proxy HTTP.

El método OPTIONS permite al cliente descubrir lo que le está permitido hacer a un recurso.

La respuesta a una solicitud de OPTIONS contiene el encabezado HTTP Allow, que establece el subconjunto de la interfaz uniforme apoya este recurso. Aquí está una muestra de encabezado Allow:

Allow: GET, HEAD

Esa cabecera especial significa que el cliente puede esperar que el servidor actúe razonablemente a una petición GET o HEAD de este recurso, pero que el recurso no admite ningún otro métodos HTTP. Efectivamente, este recurso es de sólo lectura.

Las cabeceras que el cliente envía en la solicitud pueden afectar a la cabecera Allow que el servidor envía en respuesta. Por ejemplo, si envía un encabezado de autorización adecuada, junto con una solicitud OPTIONS, usted puede encontrar que usted está autorizado a hacer peticiones GET, HEAD, PUT y DELETE contra un URI particular. Si envía la misma petición, OPTIONS y se omite el encabezado de autorización, es posible que sólo se permitan hacer solicitudes GET y HEAD. El método OPTIONS permite al cliente hacer el control de acceso sencillo.

En teoría, el servidor puede enviar información adicional en respuesta a una solicitud de OPTIONS, y el cliente puede enviar peticiones OPTIONS que hacen preguntas muy específicas sobre capacidades del servidor. Muy bien, excepto que no hay estándares aceptados de lo que es un cliente puede pedir en una solicitud de OPTIONS. Además de la cabecera Allow no hay normas aceptadas para lo que un servidor puede enviar la respuesta. La mayoría de los servidores web y marcos cuentan con muy poco apoyo para las opciones.

POST

Ahora llegamos al mas incomprendido de los métodos HTTP: POST. Este método básicamente tiene dos propósitos: uno que se ajusta a las restricciones de REST, y uno que sale a la calle REST e introduce un elemento de estilo RPC. En casos complejos como este lo mejor es volver al texto original. Esto es lo que el RFC 2616, el estándar HTTP, dice acerca de POST.

POST está diseñado para permitir un método uniforme para cubrir las siguientes funciones:

- Anotación de los recursos existentes;
- Publicar un mensaje en un tablón de anuncios, grupos de noticias, listas de correo, o grupo similar de artículos;
- Proporcionar un bloque de datos, tales como el resultado de la presentación de una forma, a un proceso de manejo de datos;
- Extensión de una base de datos a través de una operación de agregación..

La función real realizado por el método POST se determina por el servidor y eso por lo general depende de la Request-URI. La entidad está subordinada a la URI de la misma manera que un archivo está subordinado a un directorio que lo contiene, un artículo de noticias está subordinado a un grupo de noticias al cual está publicado, o un registro está subordinado a una base de datos.

Seguridad e idempotencia

Cuando se utilizan correctamente las peticiones GET y HEAD son seguras. GET, HEAD, PUT y DELETE son idempotentes.

Seguridad

Una petición GET o HEAD es una solicitud para leer cierta información, no una petición de cambiar algún estado del servidor. El cliente puede hacer una solicitud GET o HEAD 10 veces y es lo mismo que si lo hiciera por primera vez, o que nunca lo hubiera hecho. Cuando vayas a <http://www.google.com/search?q=jellyfish>, no va a cambiar nada sobre el directorio de recursos de jellyfish. Solo estaras obteniendo una representación del mismo.

Un cliente debe ser capaz de enviar una solicitud GET o HEAD para una URI desconocida y sentirse seguro de que nada catastrófico va a suceder.

Esto no quiere decir que las peticiones GET y HEAD no pueden tener efectos secundarios. La mayoría de los servidores web guardan un log de cada petición de entrada a un archivo de registro. Estos son los efectos secundarios: el estado del servidor, e incluso el estado de los recursos, cambia en respuesta a una petición GET. Pero el cliente no pidió los efectos secundarios, y no es responsable de los mismos. Un cliente nunca debe hacer una solicitud GET o HEAD sólo por los efectos secundarios, y los efectos secundarios no deben ser tan grandes como para que el cliente se arrepienta de haber hecho la solicitud.

La idempotencia es una noción un poco engañosa. La idea viene de las matemáticas, y si no estás familiarizado con la idempotencia, un ejemplo matemático puede ayudar. Una operación idempotente en matemáticas es la que tiene el mismo efecto si se aplica una, o más veces.

Multiplicar un número por cero es idempotente: $4 \times 0 \times 0 \times 0$ es lo mismo que 4×0 . Por analogía, una operación en un recurso es idempotente si al hacer una solicitud no cambia el recurso. Multiplicar un número por uno es a la vez seguro e idempotente: $4 \times 1 \times 1 \times 1$ es lo mismo que 4×1 , que es el mismo que 4. La multiplicación por cero no es segura, porque 4×0 no es el mismo que 4. Multiplicar por cualquier otro número no es ni seguro ni idempotente. Al hacer una serie de solicitudes idénticas. La segunda y siguientes solicitudes salen del estado del recurso con exactamente el mismo estado en que lo hizo la primera solicitud.

PUT y DELETE son idempotentes. Si elimino un recurso este desaparece, si lo elimino de nuevo continúa desaparecido. Si se crea un nuevo recurso con PUT y a continuación vuelva a enviar la petición PUT, el recurso todavía esta allí y conserva las mismas propiedades que se le dieron cuando se creó. Si se utiliza PUT para cambiar el estado de un recurso, se puede enviar la solicitud PUT y el estado de los recursos no va a cambiar de nuevo.

El resultado práctico de esto es que no se debe permitir a los clientes hacer PUT de representaciones que cambian el estado de un recurso en términos relativos. Si un recurso mantiene un valor numérico como parte de su estado de recurso, un cliente puede utilizar PUT para cambiar el valor a 4 o 0 o -50, pero no para incrementar ese valor por 1. Si el valor inicial es 0, enviando dos peticiones PUT que dicen "establecer el valor a 4" deja el valor en 4. Si el valor inicial es 0, enviando dos peticiones PUT que dicen "incrementar el valor en 1" deja el valor no en 1, sino en 2. Eso no es idempotente.

¿Por qué es importante la seguridad e idempotencia?

Seguridad e idempotencia es dejar a un cliente realizar solicitudes HTTP seguras a través de una red no confiable. Si usted hace una petición GET y no obtiene una respuesta, simplemente hace otra.

Es seguro: Aunque una petición anterior haya fallado, esta no tiene ningún efecto real sobre el servidor. Si hacemos una solicitud PUT y no obtenemos una respuesta, simplemente hacemos otra. Si una solicitud anterior recibió respuesta, la segunda solicitud no tendrá ningún efecto adicional.

POST no es ni seguro ni idempotente. Hacer dos solicitudes POST idénticas a un recurso de "fábrica" probablemente dará lugar a dos recursos subordinados que contienen la misma información.

El mal uso más común de la interfaz uniforme es exponer las condiciones inseguras a través de GET. Las API's de Flickr y del.icio.us hacen esto. Cuando vas a

<https://api.del.icio.us/posts/delete>, no estás trayendo una representación, estás modificando el conjunto de datos de del.icio.us.

¿Por qué es malo? Bueno, aquí una historia. En 2005 Google lanzó una herramienta de almacenamiento en caché del lado del cliente llamado Web Accelerator. Funciona en combinación con el navegador web y "pre-obtiene" las páginas enlazadas desde cualquier página que estás viendo. Si por casualidad usted hace clic en uno de estos enlaces, la página en el otro lado se cargará más rápido, debido a que su equipo ya la ha ido a buscar.

Web Accelerator fue un desastre. No a causa de algún problema en el software en sí mismo, sino debido a que la Web está llena de aplicaciones que utilizan mal el método GET. Web Accelerator supuso que las operaciones GET eran seguras, que los clientes podían hacerlas antes de tiempo por si un ser humano quisiera ver las representaciones correspondientes. Pero cuando se hicieron aquellas solicitudes sobre URI's reales GET, esto cambió los conjuntos de datos. Las personas perdieron datos.

La culpa es repartida: los programadores no deben exponer las acciones inseguras a través de GET, y Google no debería haber lanzado una herramienta que no funcionó con la web en el mundo real. La versión actual de Web Accelerator ignora todas las URI's que contienen variables de consulta. Esto resuelve parte del problema, sino que también impide que muchos recursos seguros de usar a través de GET.

Muchos de los servicios y aplicaciones web que utilizan URI's como entrada, lo primero que hacen es enviar una petición GET a buscar una representación de un recurso. Estos servicios no pretenden provocar efectos secundarios catastróficos, pero no depende de ellos. Está hasta el servicio para manejar una petición GET de una manera que cumpla con el estándar HTTP.

¿Por qué los problemas de interfaz uniforme?

Lo importante de REST no es que utiliza la interfaz uniforme específica que HTTP define. REST especifica una interfaz uniforme, pero no dice que interfaz uniforme. GET, PUT, y el resto no son una interfaz perfecta para todos los tiempos. Lo que es importante es la uniformidad: que cada servicio la interfaz de HTTP de la misma manera.

El punto no es que GET sea la mejor opción para una operación de lectura, pero GET significa "leer" a través de la Web, sin importar el recurso en el que se está usando. Dada una URI de un recurso, no hay duda de cómo se obtiene una representación: se envía una solicitud HTTP GET

a la URI. La interfaz uniforme hace que cualquiera de los dos servicios sea similar a cualquiera de los dos sitios web. Sin la interfaz uniforme, usted tiene que aprender cómo espera recibir y enviar información. Las reglas pueden incluso ser diferentes para los distintos recursos dentro de un mismo servicio.

Usted puede programar una computadora para que entienda lo que significa GET y que la comprensión se aplicará a todos los servicios web RESTful. No hay mucho que entender. El código `servicespecific` puede vivir en el manejo de la representación. Sin la interfaz uniforme se obtiene una multiplicidad de métodos que toman el lugar de GET: `doSearch`, `getPage` y `NextPrime`. Cada servicio habla un idioma diferente. Esta es también la razón por la que no me gusta sobrecargar mucho el método POST.

Algunas aplicaciones extienden la interfaz uniforme de HTTP. El caso más evidente es el Web-DAV, que añade ocho nuevos métodos HTTP como MOVE, COPY y SEARCH. El uso de estos métodos en un servicio web no infringe ningún precepto REST. Su uso violaría mi arquitectura orientada a recursos (he atado explícitamente el ROA a los métodos HTTP estándar), pero el servicio aún podría ser un recurso orientado en un sentido general.

La verdadera razón para no utilizar los métodos WebDAV es que esto hace que su servicio sea incompatible con otros servicios RESTful. Su servicio sería utilizar una interfaz uniforme diferente a la mayoría de los otros servicios. Hay servicios web como Subversion que utilizan los métodos WebDAV, por lo que su servicio no estaría solo. Pero sería parte de una red mucho más pequeña. Esta es la razón por lo que hacer sus propios métodos HTTP son una muy mala idea: su vocabulario personalizado lo pone en una comunidad de uno.

Otra interfaz uniforme consiste únicamente en HTTP GET y POST sobrecargado. Para buscar una representación de un recurso, se envía GET a una URI. Para crear, modificar o eliminar un recurso, se envía POST. Esta interfaz es perfectamente RESTful, pero, de nuevo, no se ajusta a mi arquitectura orientada a recursos. Esta interfaz es lo suficientemente rica como para distinguir entre las operaciones seguras y no seguras. Una aplicación web orientada a recursos podría utilizar esta interfaz, porque el HTML de hoy sólo admite GET y POST.

Eso es todo!

Esa es la arquitectura orientada a recursos. Son tan sólo cuatro conceptos:

1. Recursos
2. Sus nombres (URI)
3. Sus representaciones
4. Los vínculos entre ellos

y cuatro propiedades:

1. Direccionalidad
2. Statelessness
3. conectividad
4. Una interfaz uniforme

Por supuesto, todavía hay muchas preguntas abiertas. ¿Cómo puede un conjunto de datos reales ser dividido en recursos, y cómo deben ser distribuidos estos recursos? ¿Qué debe ir en las peticiones y respuestas HTTP reales? Voy a pasar la mayor parte del resto del libro explorando temas como estos.

CAPÍTULO 5

Diseñando Recursos de Solo-Lectura orientados a Objetos.

Tenemos algo de información que queremos exponer a la gente en otros lugares en la red. Queremos llegar a la mayor combinación posible de clientes. Cada lenguaje de programación tiene una biblioteca de HTTP, por lo que la opción natural es exponer los datos a través de HTTP. Cada lenguaje de programación tiene una biblioteca de análisis de XML, para que podamos dar formato a los datos con XML y siempre ser entendido.

La solución es obvia, por lo que los programadores se pusieron a trabajar. A pesar de sus fallas, esta técnica da resultados sorprendentemente buenos. La mayoría de las personas están intuitivamente familiarizados con lo que hace que una buena página web y un buen servicio web funciona de la misma manera.

Por desgracia, este enfoque instintivo combina las corazonadas de todo el mundo en una ensalada de servicios web que son por lo general no RESTful (son híbridos REST-RPC) y que trabaje sólo de manera superficial. Si usted entiende por qué funciona REST, usted puede hacer sus servicios más seguros, fáciles de usar y accesibles a través de herramientas estándar.

Algunos "servicios web" nunca fueron destinados a ser utilizados como tales, y tienen cualidades RESTful por accidente. En esta categoría entran los muchos sitios web bien diseñados que han sido tomadas por screen-scraped en los últimos años. Así que no muchos proveedores de imágenes: por ejemplo, los mosaicos de mapas estáticos sirven a la aplicación Google Maps, donde se cambia el URI para hacer frente a una parte diferente de la Tierra. Un ejemplo divertido es las imágenes de productos de Amazon, que pueden ser manipulados de forma divertida poniendo cadenas adicionales en el URI.

No es casualidad que muchos sitios web sean RESTful. Un sitio web bien diseñado presenta representaciones ordenadas de los recursos con nombres sensatos, accesibles a través de HTTP GET. Representaciones ordenadas son fáciles de analizar o de screen-scrape, y los recursos nombrados con sensatez son fáciles de direccionar mediante programación. Usando GET para recuperar una representación de una interfaz uniforme de HTTP. Diseña un sitio web con estas reglas, encajara bien con mi Arquitectura Orientada a Recursos(ROA).

Ahora que he introducido los principios de REST, dentro de ROA, mostraré cómo utilizar el ROA para diseñar servicios programáticos que sirven datos a través de la red. Estos servicios simples proporcionan acceso cliente a un conjunto de datos. Incluso pueden permitir a los clientes filtrar o buscar en los datos. Pero no deje que los clientes puedan modificar los datos o añadir en ellos. En el capítulo 6 se habla de servicios web que permiten almacenar y modificar información en el servidor. Por ahora estoy concentrado en dejar que los clientes recuperen y busquen en un conjunto de datos.

He dividido la discusión debido a que muchos servicios web excelentes no hacen más que mandar información útil a las personas que lo necesitan. No se trata de servicios de juguete. Cualquier búsqueda de base de datos basada en la web entra en estas categorías: búsquedas

en la web, el libro de búsquedas, incluso las acciones estereotipadas de servicios web. Después de todo, un servicio web que permite a los clientes modificar la información también debe permitir recuperarla.

En este capítulo se diseñó un servicio web que ofrece información sobre mapas. Está inspirado en las aplicaciones web como Google Maps. Al igual que con cualquier sitio web bien diseñado, puede consumir de Google Maps, cuadros de imágenes como un servicio web, pero sólo un poco ilícitamente y con dificultad. El servicio fantasía que yo diseñé aquí es una vía de programación amigable que se usa para recuperar datos de los mapas para cualquier propósito, incluyendo una aplicación basada en el navegador como Google Maps Ajax.

Este capítulo tiene como objetivo enseñar la forma de ver un problema desde el punto de vista orientado a recursos pueden representar un servicio distribuido muy potente y bastante complejo.

Espero demostrar las reglas simples y la interfaz uniforme de ROA pueden representar un servicio distribuido extremadamente potente y bastante complejo.

Diseño del Recurso

La técnica de diseño estándar para los programas orientados a objetos es la de romper un sistema en sus partes móviles: sus sustantivos.

Un objeto es algo. Cada sustantivo ("lector", "Columna", "Historia", "Comentario") tiene su propia clase, y el comportamiento de la interacción con los otros nombres. Por el contrario, una buena técnica de diseño para una arquitectura de estilo RPC es romper el sistema en sus propuestas: los verbos. Un procedimiento hace algo ("Suscribir a", "Leer", "Comentar").

Un recurso es algo, por lo que tome un enfoque orientado a objetos para el diseño de recursos. Un lenguaje de programación puede exponer cualquier número de métodos y darles cualquier nombre, pero un recurso HTTP expone una interfaz uniforme, a lo sumo seis métodos HTTP.

Estos métodos permiten sólo las operaciones básicas: crear (PUT o POST), modificar (PUT), lectura (GET) y borrar (DELETE). Si es necesario, se puede ampliar esta interfaz por la sobrecarga de POST, convirtiéndose un recurso en un pequeño procesador de mensajes al estilo RPC, pero usted no debería tener que hacer eso muy a menudo.

Un servicio puede exponer un recurso histórico, y una historia puede existir en forma de draft o publicación, pero un cliente no puede publicar un draft de historia en el sitio. No con tantas palabras, de todos modos: "publicar" no es una de las seis acciones. Un cliente puede poner una nueva representación de la historia que describe cómo publicada. El recurso a continuación, puede estar disponible en un nuevo URI, y puede ya no requerir la autenticación de leer. Esta es una distinción sutil, pero que le impide cometer errores de diseño peligrosas como la exposición de una de estilo RPC especial "publicar este artículo" URI a través de GET.

Convirtiendo los requerimientos en recursos de solo lectura:

He vuelto con un procedimiento a seguir una vez que se tenga una idea de lo que usted quiere que su programa haga. Se produce un conjunto de recursos que respondan a un subconjunto de sólo lectura de la interfaz uniforme de HTTP: GET y posiblemente HEAD. Una vez que llegues al final de este procedimiento, usted debería estar listo para implementar sus recursos en cualquier idioma y framework que quiera.

1. Averiguar el conjunto de datos
2. Dividir el conjunto de datos sobre los recursos

Para cada tipo de recurso:

3. Nombra los recursos con los URI
4. Exponer un subconjunto de la interfaz uniforme
5. El diseño de la representación(es) aceptada por el cliente
6. El diseño de la representación(es) que sirve al cliente
7. Integrar este recurso dentro de los recursos existentes, el uso de enlaces y formas hipermedia
8. Considere el típico curso de los acontecimientos: que se supone que suceda?
9. Considere la posibilidad de condiciones de error: qué podría salir mal?

Como encontramos el conjunto de datos

Un web service comienza con un conjunto de datos, o por lo menos una idea para uno. Este es el conjunto de datos que se va a exponer. Antes he dicho que mi conjunto de datos serían mapas. Pero que mapas? Mi web service imaginario servirá mapas en todas las proyecciones y en todas las escalas.

No se trata de cualquier tipo de mapa. Sólo sirvo mapas que utilizan un sistema de coordenadas en 2D estándar: una forma de identificar cualquier punto en el mapa. El mapa no tiene que ser exacto, pero debe ser direccionable con la latitud y la longitud.

Cada mapa tiene un número infinito de puntos pero para mi conjunto de datos no es necesario guardarlos todos. Sólo necesito unos datos de imagen y un par de piezas básicas de información sobre el mapa. Así que mi conjunto de datos incluye no sólo los mapas y los puntos en los mapas, también los mismos planetas y todos los puntos en los planetas. Puede parecer arrogante tratar a todo el planeta Tierra como un recurso, pero recuerda que no estoy obligado a dar un informe completo del estado de cualquier recurso. Si mi representación de la "tierra" es una lista de mis mapas de la Tierra, está bien. Lo importante es que el cliente puede decir "cuéntame sobre la Tierra", en contraposición a "cuéntame sobre el mapa político de la Tierra", y yo puedo dar una respuesta. Hablando de la ciudad de Nueva York y el Océano Pacífico, algunos puntos sobre un planeta son más interesantes que otros. La mayoría de los puntos no tienen nada más por debajo de ellos. Algunos de los puntos corresponden a un campo de maíz o llanura plana, y otros corresponden a una ciudad o un cráter de meteorito. Algunos puntos sobre un planeta son lugares. Mis usuarios serán desproporcionadamente interesados en estos puntos en los planetas y los puntos correspondientes en mis mapas. No van a querer especificar estos lugares como pares de latitud y longitud. De hecho, muchos de mis usuarios se tratando de averiguar dónde está algo: van a estar tratando de convertir un lugar conocido en un punto de un planeta. Para que sea más fácil para mis usuarios identificar los lugares, mi conjunto de datos incluirá un mapeo de los nombres de lugares en el correspondiente puntos en los planetas.

¿Qué pasa con lugares que no son puntos, como las ciudades, los países y los ríos? Programas de cartografía SIG representan áreas tales como las listas de puntos, que forman líneas o polígonos.

Resumen

Este es un primer paso en cualquier análisis estándar. A veces uno tiene que elegir el conjunto de datos y a veces se está tratando de exponer los datos que ya se tienen. Se puede volver a este paso para ver la mejor manera de exponer el conjunto de datos como recursos.

Me presenté los resultados de una operación de búsqueda ("lugares de la Tierra llamado Springfield") como parte del conjunto de datos. Un análisis RPC orientada trataría a estos como acciones que el cliente invoca, recordar de GoogleSearch del servicio de Google SOAP.

Comparar esto con como funciona el sitio web de Google: en un análisis de los recursos, formas de ver los datos son en sí mismos piezas de datos. Si se considera la salida de un algoritmo como un recurso, ejecutar el algoritmo puede ser tan simple como el envío de un GET a dicho recurso. Hasta ahora no he dicho nada acerca de cómo un cliente de servicios web puede acceder a este conjunto de datos a través de HTTP. Ahora mismo sólo estoy reuniendo todo en un solo lugar. También estoy haciendo caso omiso de cualquier consideración de cómo deben aplicarse estas características. Si realmente he planeado proporcionar este servicio, las características que he

anunciado hasta ahora tendrían un efecto profundo en la estructura de mi base de datos, y pude empezar a diseñar la parte de la aplicación también.

Dividir el Conjunto de Datos en los Recursos.

Una vez que se tenga un conjunto de datos en mente, el siguiente paso es decidir cómo exponer los datos como recursos HTTP. Recuerde que un recurso es cualquier cosa lo suficientemente interesante como para ser objetivo de un link de hipertexto. Cualquier cosa que pueda ser identificado por el nombre debe tener un nombre. Servicios web comúnmente exponen tres tipos de recursos:

- ***Recursos predefinidos especialmente importantes de aspectos de la aplicación.***

La mayoría de los servicios exponen pocos o ningún recurso de una sola vez.

Ejemplo: página de inicio de un sitio web. Es un recurso one-of-a-kind, en una URI conocida, que actúa como un portal a otros recursos.

El URI raíz del servicio S3 de Amazon (<https://s3.amazonaws.com/>) ofrece una lista de sus buckets S3. Sólo hay un recurso de este tipo en el S3. Usted puede obtener este recurso, pero no puede eliminarlo, y no se puede modificar directamente: se modifica sólo mediante una operación en sus buckets. Es un recurso predefinido que actúa como un directorio de recursos secundarios (los buckets).

- ***Un recurso para todos los objetos expuestos a través del servicio.***

Un servicio puede exponer muchos tipos de objetos, cada uno con su propio conjunto de recursos. La mayoría de los servicios exponen un número grande o infinito de estos recursos. Ejemplo: Cada bucket S3 que se crea se expone como un recurso. Usted puede crear un máximo de 100 buckets, y pueden tener casi cualquier nombre que desee (sus nombres no pueden entrar en conflicto con nadie más). Usted puede obtener y eliminar estos recursos, pero una vez que los ha creado no se puede modificar directamente: están modificados sólo actuando sobre los objetos que contienen. Cada objeto S3 se crea y se expone como un recurso. Un bucket tiene espacio para cualquier número de objetos. Puede aplicar GET, PUT y DELETE sobre esos recursos como mejor le parezca.

- ***Recursos que representan los resultados de los algoritmos aplicados al conjunto de datos.***

Esto incluye la recolección de recursos, que son por lo general los resultados de las consultas. La mayoría de los servicios, ya sea exponer una infinidad de recursos algorítmicos, o que no exponen ninguna.

Ejemplo: Un motor de búsqueda expone un número infinito de recursos algorítmicos. Hay uno para cada solicitud de búsqueda que posiblemente podría hacer.

El motor de búsqueda de Google expone un recurso en <http://google.com/search?q=jellyfish> (eso sería "un directorio de recursos sobre las medusas") y otro en [http://google.com/search?q =](http://google.com/search?q=)

chocolate ("un directorio de recursos sobre el chocolate"). Ninguno de estos recursos se definieron explícitamente antes de tiempo: Google traduce cualquier URI con el formato `http:// google.com / search q = {query}` en un recurso algorítmico? "Un directorio de recursos sobre {query}."

No entre en mayor detalle en el Capítulo 3, pero S3 también expone un número infinito de recursos algorítmicos. Si a usted le interesa mirar hacia atrás el Ejemplo 3-7 y la implementación de S3 :: Bucket # getObject.

Algunos recursos algorítmicos de S3 funcionan como un motor de búsqueda de los objetos en un bucket. Si sólo está interesado en los objetos cuyos nombres comienzan con la cadena "movies/" , hay un recurso que: se expone a través del `https://s3.amazonaws.com/MyBucket URI? Prefix = movies/`.

Usted puede GET este recurso, pero no se puede manipularlo directamente: es sólo una visión del conjunto de datos subyacente.

Vamos a aplicar estas categorías al servicio de mapas de fantasía. Necesito un recurso especial que muestra los planetas, al igual que S3 tiene un recurso de nivel superior que muestra los buckets. Es razonable vincular a esto la lista de planetas el recurso "la lista de planetas.": Cada planeta es un recurso, es razonable vincular Venus como un recurso. Cada mapa de un planeta es un recurso , es razonable vincular al "mapa de radar de Venus." como un recurso. La lista de los planetas es un recurso del primer tipo, ya que sólo hay uno de ellos. Los planetas y los mapas son también los recursos de una sola vez: mi servicio servirá un pequeño número de mapas para un pequeño número de planetas.

Éstos son algunos de los recursos:

- La lista de planetas
- Marte
- Tierra
- El mapa de satélite de Marte
- El mapa de radar de Venus
- El mapa topográfico de la Tierra
- El mapa político de la Tierra

Pero no puedo servir mapas completos y dejar que nuestros clientes averigüen el resto. Luego estaría corriendo un servicio de hosting para enormes archivos de mapas estáticos: un servicio RESTful, pero no uno muy interesante. También tengo que servir partes de mapas, orientación sobre los puntos específicos y lugares.

Cada punto de un planeta es potencialmente interesante, y así debe ser un recurso. Un punto de podría representar una casa, una montaña, o la ubicación actual de un barco. Estos son los recursos del segundo tipo, debido a que hay un número infinito de puntos en cualquier planeta. Por cada punto en un planeta hay un punto correspondiente en uno o varios mapas.

Por eso me limité a mapas direccionables. Cuando el mapa puede ser direccionado por la latitud y longitud, es fácil de convertir un punto del planeta en un punto en el mapa.

Éstos son algunos más de los recursos hasta el momento:

- 24.9195N 17.821E en la Tierra
- 24.9195N 17.821E en el mapa político de la Tierra

- 24.9195N 17.821E en Marte
- 44N 0V en el mapa geológico de la Tierra

Yo también sirvo lugares: puntos en un planeta identificado por su nombre en lugar de sus coordenadas.

Mi base de datos de fantasía contiene un gran número, pero finito de lugares. Cada lugar tiene un tipo, una latitud y longitud, y cada uno también pueden tener datos asociados adicionales. Por ejemplo, una zona de alta contaminación debe "saber", que contaminante hay y cual es la concentración. Al igual que con los puntos identificados por la latitud y la longitud, el cliente debe ser capaz de moverse de un lugar en el planeta hasta el punto correspondiente en cualquier mapa.

¿Los Lugares son Recursos?

¿Son los lugares realmente recursos propios o son simplemente nombres alternativos para los recursos "punto del planeta " que yo definí? Después de todo, cada lugar es un punto geográfico. Quizás tengo una situación en la que un solo recurso tiene dos nombres: uno basado en latitud y longitud, y otro basado en un nombre legible por humanos.

Bien, considere un lugar que representa la ubicación actual de un barco. Coincide con un punto específico en el mapa, y podría ser considerado simplemente un nombre alternativo para ese punto. Pero en una hora, que va a coincidir con otro punto en el mapa.

Una empresa también puede moverse en el tiempo de un punto del mapa a otro. Un lugar en el que este servicio no es la ubicación: es algo que tiene ubicación. Un lugar tiene una vida independiente desde el punto en el mapa que ocupa.

Esto es análogo a la discusión en el capítulo 4 sobre si "la versión 1.0.3" y "la última versión" apuntan a un mismo recurso. Puede suceder que apuntan a los mismos datos en este momento, pero hay dos cosas diferentes allí, y cada uno puede ser el destino del link de hipertexto. Yo podría vincular a uno y decir "Versión 1.0.3 tiene un fallo." Yo podría vincular a la otra y decir "descargar la última versión." Del mismo modo, podría vincular a un punto de la Tierra y decir "El tesoro está enterrado aquí . "O podría vincular a un lugar llamado " USS Mutiny "en las mismas coordenadas y decir" Nuestro barco está aquí. "las plazas son sus propios recursos, y son recursos de segundo tipo: cada uno corresponde a un objeto expuesto por el servicio.

Dije antes que los nombres de lugares son ambiguos. Hay cerca de 6.000 ciudades y pueblos en los Estados Unidos (aprox.) llamado Springfield. Si un nombre de lugar es inusual sólo puede decir de qué planeta es el, y es tan buena como la que especifica la latitud y longitud.

Si el nombre del lugar es común, es posible que tenga que especificar más información de alcance: dar un continente, país, o la ciudad junto con el nombre del lugar.

Éstos son algunos recursos de muestra:

- El Cleopatra es un cráter en Venus.
- El cráter Ubehebe está en la Tierra.
- 1005 Gravenstein Highway Norte, Sebastopol, CA.
- La sede de O'Reilly Media, Inc.
- El lugar llamado Springfield en Massachusetts, en los Estados Unidos de América, en la Tierra.

Hasta el momento, se trata de algo bastante general. Los usuarios quieren saber que mapas tenemos, por eso nosotros exponemos un recurso único que muestra los planetas. Cada planeta es también un único recurso que se vincula con los mapas disponibles. Un punto geográfico del planeta es direccionable por latitud y longitud, así que tiene sentido exponer cada punto como un recurso direccionable.

Cada punto en un planeta corresponde a un punto en uno o varios mapas. Algunos puntos son interesantes y tienen nombres, por lo que a los lugares en el planeta también se puede acceder por su nombre: un cliente puede encontrarlo en el planeta y luego ver ese punto en un mapa.

Todo lo que he hecho hasta ahora es describir las interacciones entre las partes de un conjunto de datos predefinido. Todavía no he expuesto los recursos generados algorítmicamente, pero es bastante fácil de agregar. El tipo más común de recursos algorítmicos es una lista de resultados de búsqueda. Yo permito que mis clientes busquen lugares en un planeta que tienen ciertos nombres o que indique los criterios específicos de un lugar.

Aquí hay algunos ejemplos de recursos algorítmicos:

- Lugares de la Tierra llamado Springfield
- Los buques portacontenedores de la Tierra
- Cráteres en Marte más de 1 km de diámetro
- Lugares en la Luna llamado antes de 1900

Los resultados de la búsqueda se pueden limitar a un área en particular, y no sólo a un planeta. Algunos recursos mas de muestra:

- Lugares en los Estados Unidos llamados Springfield
- Los sitios de aguas termales en Colorado
- Los petroleros o buques de contenedores cerca de Indonesia
- Pizzerías en Worcester, MA
- Restaurantes cerca de Mount Rushmore
- Áreas de alto arsénico cerca 24.9195N 17.821E
- Ciudades en Francia con población inferior a 1.000

Estos son todos los recursos generados algorítmicamente, porque se basan en el cliente proporcionando una cadena de búsqueda arbitraria ("Springfield") o la combinación de elementos no vinculados ("monte Rushmore" + restaurantes, o "Francia" + + "población <1.000").

Podría llegar a nuevos tipos de recursos durante todo el día (de hecho, eso es lo que hice al escribir esto). Sin embargo, todos los recursos que he pensado hasta ahora caben en cinco tipos básicos, lo suficiente para hacer la fantasía interesante. Ejemplo 5-1 da la lista maestra de los tipos de recurso.

Ejemplo 5-1. Los cinco tipos de recursos:

1. La lista de los planetas
2. Un lugar en un planeta, posiblemente todo el planeta-identificado por su nombre
3. Un punto geográfico en el planeta, identificado por latitud y longitud
4. Una lista de lugares en un planeta que coinciden con algunos de los criterios de búsqueda

5. Un mapa de un planeta, en centrado alrededor de un punto particular

Un web service en la vida real podría definir recursos adicionales. Sitios web reales como Google Maps exponen obviamente una funcionalidad que no he mencionado: cómo llegar. Si quisiera mejorar mi servicio podría exponer un nuevo recurso algorítmico que trata de un conjunto de direcciones de manejo como una relación entre dos lugares.

Un servicio web RESTful expone tanto sus datos y sus algoritmos a través de los recursos. Por lo general hay una jerarquía que comienza con cosas pequeñas y se ramifica en un número infinito de nodos.

La lista de planetas contiene los planetas, que contienen puntos y lugares, que contienen mapas. La lista de buckets S3 contiene los buckets individuales, que contienen los objetos. Se toma un tiempo para conseguir la caída de la exposición de un algoritmo como un conjunto de recursos. En lugar de pensar en términos de acciones ("hacer una búsqueda de lugares en el mapa"), es necesario pensar en términos de los resultados de esa acción (la "lista de lugares en el mapa que coincidan con un criterio de búsqueda").

Nombrando los recursos

Los recursos son nombrados por URI's. Estas URI's necesitan responder preguntas como:

¿Por qué debería utilizar el servidor en este mapa, en lugar de ese mapa?"

¿Por qué debe funcionar el servidor en este lugar en vez de ese lugar?

Ruteo mi servicio web en <http://maps.example.com/>. Para ser breves, a veces uso URIs relativas en este capítulo y el siguiente, entienden que están en relación con [http://maps.example.com /](http://maps.example.com/). Si digo [/Earth/political](http://maps.example.com/Earth/political), lo que quiero decir es <http://maps.example.com/Earth/political>.

Ahora vamos a considerar los recursos. El recurso más básico es la lista de planetas. Tiene sentido poner esto en la URI de la raíz, <http://maps.example.com/>.

Desde la lista de planetas se abarca todo el servicio.

Para el resto de los recursos lo ideal es que se organicen las URI's para tener al alcance la información de una manera natural.

Hay tres reglas básicas para el diseño de URI, nacidos de la experiencia colectiva:

1. Utilice rutas variables para codificar jerarquía: [/parent/child](#)
2. Ponga los signos de puntuación en las rutas variables para evitar la implicación de la jerarquía donde no existe: [/parent/child1;child2](#)
3. Utilice variables de consulta que implican elementos de entrada en un algoritmo, por ejemplo: [/search?q=jellyfish&start=20](#)

Codificar Jerarquías en Rutas Variables

Hagamos URI's para la segunda clase de recursos: planetas y lugares en los planetas. Hay un elemento de información al alcance aquí: ¿qué planeta estamos viendo? (¿Tierra? ¿Venus? ¿Ganímedes?) Esta información adapta de forma natural en una jerarquía: la lista de los planetas está en la parte superior, y debajo de ella está cada planeta en particular.

Estos son los URI a algunos de mis planetas.

Muestro la jerarquía utilizando el carácter de barra de piezas separadas de información.

- <http://maps.example.com/Venus>
- <http://maps.example.com/Earth>
- <http://maps.example.com/Mars>

Para identificar los lugares vamos a extender la jerarquía hacia la derecha. Nos daremos cuenta de que tenemos un buen diseño de URI's si es fácil de extender.

Ejemplos:

- <http://maps.example.com/Venus>
- <http://maps.example.com/Venus/Cleopatra>
- <http://maps.example.com/Earth/France/Paris>
- <http://maps.example.com/Earth/Paris,%20France>
- <http://maps.example.com/Earth/Little%20Rock,AR>
- <http://maps.example.com/Earth/USA/Mount%20Rushmore>
- <http://maps.example.com/Earth/1005%20Gravenstein%20Highway%20North,%20Sebastopol,%20CA%2095472>

Ahora estamos en territorio de web service.

Enviando un GET a uno de estos URI, invoca una operación remota que toma un número variable de argumentos, y puede encontrar un lugar en un planeta con un grado de precisión deseado.

Pero los URI por si mismo parecen sitios web normales que puedes marcar, cachear y pasar a otros servicios como entrada, porque eso es lo que son.

Las rutas variables son la mejor manera de organizar la información de alcance que se puede organizar jerárquicamente. La misma estructura que se ve en un sistema de archivos, o en un sitio web estático.

¿Sin Jerarquía? Utilice comas o puntos y comas.

Los siguientes recursos que necesito nombrar son los puntos geográficos en el mundo, representados por la latitud y longitud. Latitud y longitud están unidas entre sí, por lo que una jerarquía no es apropiado.

Un URI como /Earth/24.9195/17.821 no tiene sentido. La barra hace que parezca que la longitud es un concepto subordinado a la latitud.

En lugar de utilizar la barra al poner dos piezas de información en una jerarquía, recomiendo la combinación de ellos en el mismo nivel de una jerarquía con un carácter de puntuación: por lo

general el punto y coma o la coma. Voy a utilizar una coma para latitud y longitud por separado.

- <http://maps.example.com/Earth/24.9195,17.821>
- <http://maps.example.com/Venus/3,-80>

Latitud y longitud también se pueden utilizar como información preliminar para identificar un lugar convenido. Un humano probablemente identificaría Monte Rushmore como */Earth/USA/Mount%20Rushmore* o */v1/Earth/USA/SD/Mount%20Rushmore*, pero ***/v1/Earth/43.9;-103.46/Mount%20Rushmore*** sería más preciso.

Se recomienda el uso de comas cuando el orden de la información es importante y el punto y coma cuando no lo es.

URI maps

Ahora que tenemos diseñado el URI de un punto geográfico del planeta. ¿qué pasa con el punto correspondiente en una hoja de ruta o mapa de satélite? Después de todo, el punto principal de este servicio es la de servir mapas.

Antes he dicho que expondría un recurso para todos los puntos en un mapa. Por razones de simplicidad, no estoy exponiendo mapas de lugares nombrados, sólo puntos de latitud y longitud. Además de un conjunto de coordenadas o el nombre de un lugar, necesito el nombre del planeta y el tipo de mapa (mapa de satélite, hoja de ruta, o lo que sea). Éstos son algunos de los URI a los mapas de los planetas, los lugares y puntos:

- <http://maps.example.com/radar/Venus>
- <http://maps.example.com/radar/Venus/65.9,7.00>
- <http://maps.example.com/geologic/Earth/43.9,-103.46>

Escala

Un URI como */satellite/Earth/41,-112* no dice nada sobre el grado de detalle del mapa. Un ejemplo para especificar la escala puede ser la siguiente:

```
/satellite.10/Earth/41,-112  
/satellite.5/Earth/41,-112  
/satellite.1/Earth/41,-112:
```

¿Recursos algorítmicos? Uso de las variables de consulta.

La mayoría de las aplicaciones web no almacenan mucho del estado en las rutas variables: utilizan variables de consulta en su lugar.

Es posible que hayan visto los URI de esta manera:

- <http://www.example.com/colorpair?color1=red&color2=blue>
- <http://www.example.com/articles?start=20061201&end=20071201>
- <http://www.example.com/weblog?post=My-Opinion-About-Taxes>

Los URI's se verían mejor sin las variables de consultas:

- <http://www.example.com/colorpair?color1=red&color2=blue>
- <http://www.example.com/articles?start=20061201&end=20071201>
- <http://www.example.com/weblog?post=My-Opinion-About-Taxes>

En algunos casos las variables de consultas son apropiadas. Aquí hay una búsqueda de Google URI: <http://www.google.com/search?q=jellyfish>.

Si la aplicación web Google utiliza variables de ruta, el URI se vería más como directorios y menos como el resultado de ejecutar un algoritmo:

Ej: <http://www.google.com/search/jellyfish>.

Esta percepción de variables de consulta se refuerza cada vez que utilizamos la Web.

Al rellenar un formulario HTML en un navegador web, los datos que ingresa son convertidos en variables de consulta. No hay manera de escribir "jellyfish" en un formulario y luego enviar a <http://www.google.com/search/jellyfish>.

El destino de un formulario HTML es codificado para <http://www.google.com/search/> y cuando llene el formulario terminas en <http://www.google.com/search?q=jellyfish>. Su navegador sabe cómo rutear las variables de consulta en un URI base. No sabe cómo sustituir las variables en un URI genérico como <http://www.google.com/search/{q}>.

Debido a este precedente, una gran cantidad de servicios híbridos REST-RPC utiliza variables de consulta, cuando sería más idiomática utilizar variables de ruta.

Incluso cuando un servicio híbrido pasa a exponer los recursos REST, los recursos tienen URIs que se ven como llamadas de función: URI's como

<http://api.flickr.com/services/rest/?method=flickr.photos.search&tags=penguin>.

Compare esto con la URI correspondiente URI en el sitio Flickr:

<http://flickr.com/photos/tags/penguin>.

He aquí cómo un cliente puede utilizar programa para construir URI para algunos de los recursos.

- <http://maps.example.com/Earth?show=Springfield>
- <http://maps.example.com/Mars?show=craters+bigger+than+1km>
- <http://maps.example.com/Earth/Indonesia?show=oil+tankers&show=container+ships>
- <http://maps.example.com/Earth/USA/Mount%20Rushmore?show=diners>

- <http://maps.example.com/Earth/24.9195;17.821?show=arsenic>

Notar que todos estas URI's están buscando el planeta, no cualquier mapa en particular.

URI Recap

Después de todo, este es el primer lugar donde mis recursos fantasía entran en contacto con el mundo real de HTTP. Aun así, mi servicio sólo es compatible con tres tipos básicos de URI. En resumen, aquí están:

- La lista de planetas: /.
 - Un planeta o un lugar en un planeta:/{planet}/{scoping-information}/{placename}:El valor de la variable opcional de la información de alcance será una jerarquía de los nombres de lugares como /USA/New%20England/Maine/ o será una latitud / longitud. El valor de la variable opcional {name} será el nombre del lugar. Este tipo de URI puede tener valores para mostrar en su cadena de consulta, para buscar ciudades cerca del lugar dado.
- Un mapa de un planeta, o un punto en el mapa: /{map-type}{scale}/{planet}/{scoping-information}. El valor de la variable opcional {scoping-information} siempre será latitud/longitud. El valor de la variable opcional {scale} será un dato o número.

Diseñe sus representaciones.

He decidido como se verán los recursos que estoy exponiendo, y cuál es su URI. Ahora tengo que decidir qué datos enviar, cuando un cliente solicita un recurso, así como el formato de datos a utilizar. Esto es sólo un calentamiento, ya que gran parte del capítulo 9 está dedicado a un catálogo de formatos de representación útiles. Aquí, tengo un servicio específico en mente, y tengo que decidir un formato (o un conjunto de formatos), que pueden alcanzar los objetivos de cualquier representación RESTful: para transmitir el estado actual de los recursos, y para vincular a una posible nueva aplicación y estados de recursos.

Las conversaciones representadas sobre el estado de los recursos.

El propósito principal de cualquier representación es transmitir el estado de los recursos. Recuerde que "el estado del recurso" es cualquier información acerca de los recursos que subyacen. En este caso, el estado va a responder a preguntas como: ¿que parte del mundo es gráficamente como esta? ¿Dónde esta exactamente el cráter del meteorito en cuanto a latitud y longitud? ¿Dónde están los restaurantes de la zona y cuáles son sus nombres? ¿Dónde están los barcos de contenedores en este momento?

Las representaciones de los diferentes recursos representarán diferentes puntos del estado.

La representación Enlaces a otros Estados.

El objetivo aquí es la conectividad: la capacidad de ir de un recurso a otro siguiendo los enlaces. Así es como funcionan los sitios web. No se navega por la Web escribiendo una URI tras otra. Usted puede escribir una URI para llegar a la página principal de un sitio, pero luego navegar siguiendo los enlaces y rellenar formularios.

Si un recurso se puede modificar con PUT, o puede generar nuevos recursos en respuesta a POST, su representación debe también exponer a los resortes del estado de los recursos. La representación debe proporcionar toda la información necesaria acerca de lo que la petición POST o PUT debe ser similar.

Representando la lista de planetas

La "página de inicio" de mi servicio de mapas es un buen lugar para empezar, y un buen lugar para introducir los temas detrás de la elección de un formato de representación. Básicamente, quiero mostrar una lista de enlaces a los planetas de los que tengo mapas. ¿Cual es un buen formato para la representación de una lista? Siempre hay texto sin formato. Esta representación muestra un planeta por línea: la URI y el nombre.

```
http://maps.example.com/Earth Earth
http://maps.example.com/Venus Venus
```

Esto es simple, pero requiere un analizador personalizado. Yo por lo general pienso que un formato de datos estructurado es mejor que el texto plano, especialmente las representaciones se vuelven más complejas. JSON mantiene la sencillez de texto plano, pero añade un poco de estructura.

Ejemplo 5-3. representación JSON de la lista de planetas

```
[{url="http://maps.example.com/Earth", description="Earth"},
{url="http://maps.example.com/Venus", description="Venus"},
...]
```

Lo malo es que ni JSON ni texto plano son generalmente considerados formatos "hipermedia". Otra opción popular es un vocabulario XML personalizado, ya sea con o sin una definición de esquema:

Ejemplo 5-4. Representación XML de la lista de planetas

```
<?xml version="1.0" standalone='yes'?>
<planets>
<planet href="http://maps.example.com/Earth" name="Earth" />
<planet href="http://maps.example.com/Venus" name="Venus" />
...
```

</maps>

En estos días, un vocabulario XML personalizado parece ser la opción por defecto para las representaciones de servicios web. XML es excelente para representar documentos. Los problemas básicos ya se han resuelto, y la mayoría de las veces se puede volver a utilizar un vocabulario XML existente. Da la casualidad de que ya hay un vocabulario XML para comunicar las listas de enlaces llamados Atom.

Ejemplo 5-5. El ejemplo muestra una representación XHTML de la lista de planetas..

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Planet List</title>
</head>
<body>
<ul class="planets">
<li><a href="/Earth">Earth</a></li>
<li><a href="/Venus">Venus</a></li>
...
</ul>
</body>
</html>
```

Representando mapas y puntos del mapa.

¿Qué hay de los mapas? ¿Qué sirvo, si alguien pide un mapa de satélite de la Luna? Lo obvio para enviar una imagen, ya sea en un formato gráfico tradicional como PNG o un gráfico vectorial SVG. A excepción de los mapas de mayor escala, estas imágenes serán enormes. ¿Está bien? Depende de la audiencia para mi servicio web.

Si estoy sirviendo a clientes con gran ancho de banda que esperan procesar enormes cantidades de datos del mapa, los archivos grandes son exactamente lo que quieren. Pero es más probable que mis clientes serán como los usuarios de las aplicaciones de mapas existentes, como Google y Yahoo! Mapas: clientes que quieren mapas de menor tamaño para la navegación humana. Esto es más o menos cómo funcionan los sitios de mapas en línea. Si usted visita <http://maps.google.com/>, se obtiene un mapa político centrado en los Estados Unidos: esa es la representación de si usted visita <http://maps.google.com/> "un mapa de la Tierra."

Si tu visitas <http://maps.google.com/maps?q=New+Hampshire>, se obtiene una hoja de ruta centrada en Concord, la capital. En cualquier caso, el mapa se divide en cuadrados, imágenes de 256 píxeles de lado.

Cuando el cliente solicita un punto en el mapa, voy a servir un archivo de hipermedia que incluye un enlace a una imagen de mapa pequeño (un solo cuadro, generado dinámicamente) centrado en ese punto. Cuando el cliente solicita un mapa de un planeta entero, voy a recoger un punto en ese planeta un tanto arbitraria y servir un archivo de hipermedia que se vincula a una imagen centrada en ese punto. Estos archivos hipermedia incluirán enlaces a puntos adyacentes en el mapa, que incluirá más enlaces a puntos adyacentes, y así sucesivamente. El cliente puede seguir los vínculos de navegación para formar un mapa de cualquier tamaño deseado.

Así el ejemplo siguiente es una posible representación de <http://maps.example.com/road/Earth>. Al igual que mi representación de la lista de planetas, usa XHTML para transmitir el estado del recurso y de vincular a los recursos "cercaños". El estado de los recursos es la información sobre un determinado punto en el mapa. Los recursos "cercaños" están muy cerca, en un sentido literal: son puntos cercanos.

Representación XHTML de mapa de ruta de la tierra:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Road Map of Earth</title>
</head>
<body>
...

...
<a class="map_nav" href="46.0518,-95.8">North</a>
<a class="map_nav" href="41.3776,-89.7698">Northeast</a>
<a class="map_nav" href="36.4642,-84.5187">East</a>
<a class="map_nav" href="32.3513,-90.4459">Southeast</a>
...
<a class="zoom_in" href="/road.1/Earth/37.0;-95.8">Zoom out</a>
<a class="zoom_out" href="/road.3/Earth/37.0;-95.8">Zoom in</a>
...
</body>
</html>
```

Ahora, cuando un cliente solicita el recurso "una hoja de ruta de la Tierra" en la URI `/road/Earth`, la representación que reciben no es una enorme imagen increíblemente detallada que no pueden tratar. Es un pequeño documento XHTML, que incluye enlaces a otros recursos.

Representando cuadros de los Mapas

La representación de una hoja de ruta de la Tierra, tiene un montón de enlaces en el. La mayoría son enlaces a documentos XHTML que se parecen mucho a una "hoja de ruta de la Tierra" no: son representaciones de puntos en el mapa en diferentes escalas. El vínculo más importante, sin

embargo, es el que está en la etiqueta IMG. El tag src referencia atributos de esa etiqueta por ejemplo URI <http://maps.example.com/road/Earth.8/images/37.0;-95.png>.

En estos casos, el archivo XHTML incluiría esta imagen como referencia, y también vincula a un montón de lugares en el mapa.

Otro Ejemplo: el cliente solicita /road.8/Earth/32.37,-86.30, mi servicio enviará una representación XHTML cuya etiqueta IMG referencia /road.8/Earth/images/32.37,-86.30.png.

Se trata de una hoja de ruta muy detallada centrada en 32.37 °N, 86.30 ° W en la Tierra.

Vale la pena repetir aquí que si mis clientes realmente necesitan mapas detallados multigigabyte, no tiene sentido en mí ir cortando el estado de la hoja en estas pequeñas piezas. Sería más eficiente tener la representación de /road/Earth?zoom=1 transmitir a todo el estado de la hoja con una imagen enorme.

He diseñado para clientes que sólo quieren realmente parte de un mapa, y no sabrían qué hacer con un enorme mapa de la tierra si se los doy a ellos. Los clientes que tengo en mente puede consumir los archivos XHTML, cargar las imágenes correspondientes, y seguir automáticamente los enlaces para unir un mapa que es tan grande como sea necesario.

Se puede escribir un cliente de Ajax para mi servicio web que funcionaba como la aplicación Google Maps.

Representando planetas y otros lugares.

He mostrado las representaciones de la lista de planetas, de los mapas de los planetas y los puntos en los mapas. Es de suponer que hace click en "Tierra" en la lista de planetas, enviando una solicitud GET a /Earth, y debe volver una representación de la Tierra. Esta representación incluye un montón de enlaces a mapas de la Tierra. En este punto se sigue un segundo enlace a la hoja de ruta de la Tierra. Mi representación de un planeta que contiene información útil que tengo sobre el planeta, así como una serie de enlaces a otros recursos: mapas del planeta.

Ejemplo 5-7. Representación XHTML de un lugar: El planeta Tierra

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head><title>Earth</title></head>
<body>
<dl class="place">
<dt>name</dt> <dd>Earth</dd>
<dt>maps</dt>
<dd>
<ul class="maps">
<li><a class="map" href="/road/Earth">Road</a></li>
<li><a class="map" href="/satellite/Earth">Satellite</a>
...
</ul>
</dd>
<dt>type</dt> <dd>planet</dd>
```

```

<dt>description</dt>
<dd>
Third planet from Sol. Inhabited by bipeds so amazingly primitive
that they still think digital watches are a pretty neat idea.
</dd>
</dl>
</body>
</html>

```

He optado por representar lugares como una lista de pares clave-valor. Aquí, el “lugar” es el planeta tierra en sí mismo. La tierra en este sistema es un lugar con nombre, al igual que San Francisco o Egipto. Estoy representando esto usando las etiquetas dd: un estándar de HTML para representar conjuntos de pares clave-valor. Como cualquier otro lugar, la tierra tiene un nombre, un tipo, una descripción, y una lista de mapas: enlaces a todos los recursos de este lugar. Porque estoy representando el planeta como un lugar? Porque ahora mis clientes pueden procesar la representación de un planeta con el mismo código que usan para procesar la representación de un lugar. El ejemplo 5-7 es una representación de “Mount Rushmore” en la tierra. Puedes obtener este archivo XHTML como respuesta al realizar una solicitud GET a /Earth/USA/Mount%20Rushmore.

Ejemplo 5-8. Representación de un lugar en XHTML: Mount Rushmore

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head><title>Mount Rushmore</title></head>
<body>
<ul class="places">
<li>
<dl class="place">
<dt>name</dt> <dd>Mount Rushmore</dd>
    <dt>location</dt>
    <dd>
<a class="coordinates" href="/Earth/43.9;-95.9">43.9&deg;N
95.8&deg;W</a>
    </dd>
    <dt>maps</dt>
    <ul class="maps">
<li><a class="map" href="/road/Earth/43.9;-95.9">Road</a></li>
<li><a class="map" href="/satellite/Earth/43.9;-95.9">Satellite</a>
...
</ul>
</dd>
<dt>type</dt> <dd>monument</dd>
<dt>description</dt>
<dd>
Officially dedicated in 1991. Under the jurisdiction of the
<a href="http://www.nps.gov/">National Park Service</a>.
</dd>

```

```

    </dl>
  </li>
</body>
</html>

```

En vez de ofrecer una imagen de un mapa de Mount Rushmore, o una página XHTML que contenga enlaces a esta imagen, esta representación enlaza a los recursos que he definido: mapas del punto geográfico donde Mount Rushmore está ubicado. Estos recursos se encargan de toda las imágenes y detalles de navegación. El propósito de este recurso es el de informar el estado del lugar, y como se ve en un mapa es sólo una parte de su estado.

También está su nombre, su tipo ("monumento", y su descripción. La única diferencia entre la representación de un planeta y la de un lugar es que el lugar tiene una ubicación es su definición, mientras que el planeta no. Un cliente puede procesar ambas representaciones con el mismo código.

Como habrás notado, no es necesario escribir un cliente especial para este web service en absoluto. Puedes usar un navegador web y elegir cualquier planeta, como la tierra, obteniendo la representación mostrada en el ejemplo 5-7. Este webservice funciona como un sitio web también, aunque está diseñado para ser usado por programas, esto no evita que pueda ser usado por humanos dentro de un navegador web.

Web services son tan solo sitios web para robots, mi servicio de mapas es similar a un sitio web: conecta recursos entre sí por medio de hipervínculos. No utiliza ningún elemento que un navegador estándar no soporta. Pero todos los web services RESTful son parte de la Web aunque no puedan ser usados con un navegador web.

Ejemplo 5-9 muestra una representación más: la representación de un punto en el mapa.

Ejemplo 5-9. Una representación XHTML de un punto 43.9°N 103.46°W en la tierra:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>43.9&deg;N 103.46&deg;W on Earth</title>
</head>
<body>
<p>
Welcome to
<a class="coordinates" href="/Earth/43.9;-103.46">43.9&deg;N
103.46&deg;W</a>
on scenic <a class="place" href="/Earth">Earth</a>.
</p>
<p>See this location on a map:</p>
<ul class="maps">
<li><a class="map" href="/road/Earth/43.9;-95.9">Road</a></li>

```

```

<li><a class="map" href="/satellite/Earth/43.9;-95.9">Satellite</a></li>
...
</ul>
<p>Things that are here:</p>

<ul class="places">
<li><a href="/Earth/43.9;-95.9/Mount%20Rushmore">Mount Rushmore</a></li>
</ul>
<form id="searchPlace" method="get" action="">
<p>
Show nearby places, features, or businesses:
<input name="show" repeat="template" /> <input class="submit" />
</p>
</form>
</body>
</html>

```

Esta representación consiste íntegramente de enlaces: enlaces a mapas centrados alrededor de este punto, y enlaces a lugares ubicados en este punto. No posee estado por sí mismo, sino que provee otros recursos más interesantes.

Representando Listas de Resultados de Búsquedas

He mostrado la representación de listas de planetas, un planeta, puntos y lugares en un planeta, y sus mapas. Pero qué hay de los recursos algorítmicos, los resultados de búsquedas? Cual sería una buena representación de un recurso para “restaurantes cerca de Mount Rushmore” (/Earth/USA/Mount%20Rushmore?show=diners)?

Una búsqueda debe asociar el lugar que es buscado, por lo tanto la representación de “restaurantes cerca de Mount Rushmore” debe relacionarse con el lugar “Mount Rushmore”.

Cuando el cliente busque en or entorno a un lugar, está buscando más lugares. Por más que el texto usado sea ambiguo o una descripción muy general, los resultados serán ubicaciones en el mapa: ciudades, restaurantes.

Por lo tanto, los resultados pueden ser tan sólo enlaces a otros recursos ya definidos. Y el cliente accede a ellos para obtener más información de los mismos.

Ejemplo 5-10. Representación de una lista de lugares llamados Springfield en Estados Unidos.

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head><title>Search results: "Springfield"</title></head>
<body>
<p>
Places matching <span class="searchterm">Springfield</span>

```

```

in or around
<a class="place" href="/Earth/USA">the United States of America</a>:
</p>
<ul>
<li>
<a class="place" href="/Earth/USA/IL/Springfield">Springfield, IL</a>
</li>
<li>
<a class="place" href="/Earth/USA/MA/Springfield">Springfield, MA</a>
</li>
<li>
<a class="place" href="/Earth/USA/MO/Springfield">Springfield, MO</a>
</li>
...
</body>
</html>

```

Esta representación está hecha íntegramente por enlaces a lugares. Hay un enlace al lugar que fue buscado “United States of America” (lugar de tipo “país”). También hay enlaces a otros lugares que contienen la palabra buscada (“Springfield”), siendo cada lugar un recurso.

Un cliente puede acceder a estos enlaces para obtener más información acerca de los lugares así como mapas de los mismos. La figura 5-3 muestra cómo es posible navegar desde /Earth/USA? show=Springfield.

Google Maps presenta resultados uniendo pequeñas imágenes para conformar un mapa a gran escala, y agregando indicadores gráficos en el. Es posible escribir un cliente para este web service que hiciera lo mismo, el primer paso sería crear un mapa a gran escala. El cliente accede el enlace inicial /Earth/USA y obtiene la representación como en el Ejemplo 5-4. El segundo paso sería agregar marcadores en el mapa para cada resultado. Para esto el cliente accede a los enlaces de los resultados y obtiene la información de estos tales como la latitud y longitud.

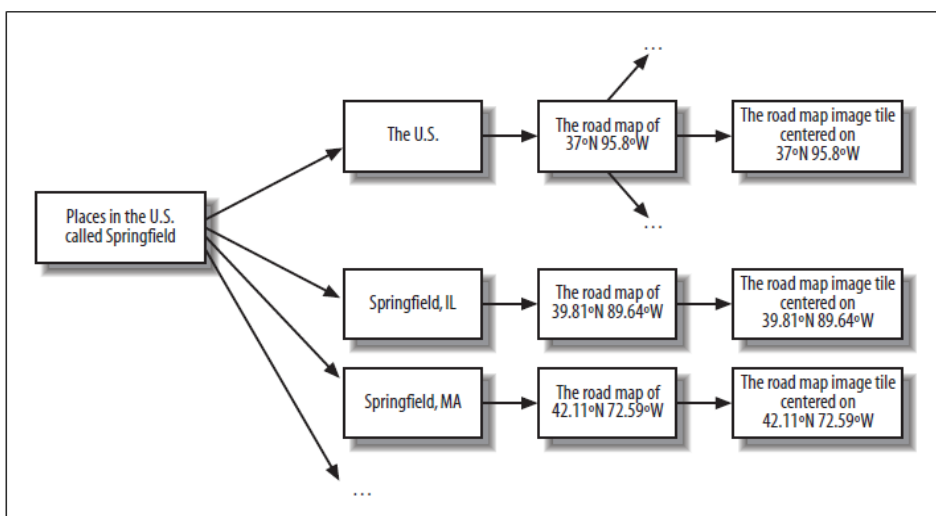


Figura 5-3. Donde puedes ir desde los resultados de una búsqueda?

Enlazar Recursos entre sí

Debido a que he diseñado todos mis recursos en paralelo, poseen varios enlaces a otros recursos. Un cliente puede obtener la lista de planetas, seguir un enlace a un planeta en específico, seguir otro link a un mapa específico y luego seguir los enlaces de navegación y zoom para ver el mapa. Un cliente podría realizar búsquedas de lugares con cierto criterio, y seguir los enlaces de los resultados para obtener más información, y luego seguir otro enlace para ubicar el lugar en el mapa.

Queremos hacer posible acceder desde `/Earth/USA/Mount%20Rushmore` a `/Earth/USA/Mount%20Rushmore?show=restaurantes`. Pero no es una buena idea hacer el enlace a específicamente a “restaurantes”, ya que es una de todas las posibles búsquedas. Y no queremos poner todas las posibles representaciones previendo que alguien desee buscar “tiendas de mascotas” o “cráteres de meteoritos” cerca de Mount Rushmore.

HTML provee formularios para resolver esto. Al enviar una form específico en una representación, podemos informar al cliente de como utilizar variables en las cadenas de búsqueda. Estos forms representan infinitas URIs, las siguen un patrón específico. Extendemos la representación de lugares (como en el ejemplo 5-8) incluyendo formularios HTML (ver Ejemplo 5-11)

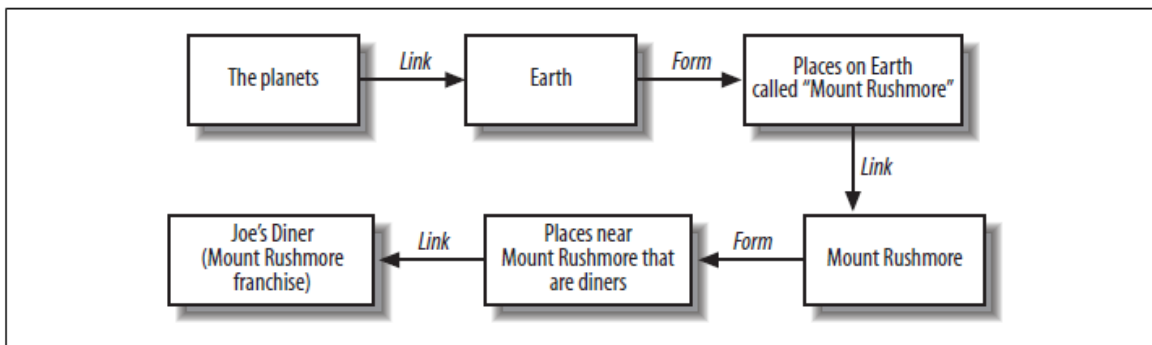


Figura 5-4. El camino desde el servicio raíz hasta “restaurantes cerca de Mount Rushmore”

Ejemplo 5-11. Un formulario HTML para la búsqueda de lugares

```

<form id="searchPlace" method="get" action="">
  <p>
    Show places, features, or businesses:
    <input id="term" repeat="template" name="show" />
    <input class="submit" />
  </p>
</form>

```

Cuando una persona vea este formulario verá un cuadro de texto, un botón y unas etiquetas. Al poner algún dato en el cuadro de texto, y presionar el botón, serán llevados a otra URI. Si estuvieran en `/Earth/USA/Mount%20Rushmore` e ingresan “restaurantes”, serían llevados a `/Earth/USA/Mount%20Rushmore?show=restaurantes`. Un cliente puede leer la definición del formulario `searchPlace`, y usar dicha definición para construir la URI `/Earth/USA/Mount%20Rushmore?show=restaurantes`.

Probablemente no hayas visto la sintaxis `repeat="template"` antes. Es una característica de XHTML5, la cual está en aun esta en diseño en el momento que este libro es publicado.

El problema es que mi servicio acepta cualquier número de valores para la variable `show`. Un cliente puede realizar una búsqueda simple como `?show=restaurantes` o una búsqueda más compleja como `?show=restaurantes&show=farmacias&show=bancos`.

Un formulario en XHTML 4 requeriría que se especificaran 4 cuadros de texto, todos llamados `show`. Con XHTML 5 es posible utilizar la característica `repetition` para definir un número arbitrario de variable sin tener que definir un formulario HTML infinitamente extenso.

Ahora mi servicio está debidamente conectado, es posible llegar desde una lista de planetas a la descripción de un restaurante cercano a Mount Rushmore (suponiendo que existe uno). La figura 5-4 ilustra dicho camino. Empezando en la raíz del servicio (`/`), el cliente selecciona el planeta Tierra (`/Earth`). El cliente usa el formulario HTML en dicha representación para buscar lugares llamados "Mount Rushmore" en la Tierra (`/Earth?show=Mount%20Rushmore`). El cliente podrá acceder al recurso perteneciente a Mount Rushmore de entre los resultados `/Earth/USA/Mount%20Rushmore`. La representación de Mount Rushmore tendrá a su vez un formulario, el cliente podrá ingresar "restaurantes" en él. Suponiendo que existe algún restaurante cercano, el cliente podrá seguir el enlace del resultado para encontrar un restaurante cerca de Mount Rushmore.

La Respuesta HTTP

El diseño está casi completo. Sabemos qué datos estamos sirviendo, qué solicitudes HTTP los clientes hacen, que datos serán representados. Aún debemos considerar la respuesta HTTP en sí. Sabemos como las posibles representaciones se vieron y que contendrán en el cuerpo, pero aún no hemos considerado los posibles códigos de respuesta o el cabezal HTTP. También debemos considerar las posibles condiciones de error, casos en los que debemos responder con un error en lugar de datos.

Que debería ocurrir?

Este paso del diseño es conceptualmente simple, pero donde se invierte bastante tiempo: asegurándote que las peticiones de tus clientes sean convertidas correctamente en respuestas.

La mayoría de los recursos tienen una manera simple de llevar a cabo esto. EL usuario envía una solicitud GET a una URI, el servidor responde con un código, por ejemplo 200 ("OK"), algunos cabeceras HTTP, y una representación. Una solicitud de cabecera funciona muy similar, pero el servidor omite la representación. La única duda es que cabeceras HTTP el cliente debe enviar, y cuales el servidor debe retornar en la respuesta.

Las cabeceras de respuestas HTTP son elementos bastante complejos, y la mayoría de ellos no son útiles para este simple servicio. En este servicio la principal respuesta en la cabecera es "Content-Type", que le dice al cliente el tipo de representación. Los tipos son `application/xhtml+xml` para las representaciones de mapas y los resultados de búsquedas; e `image/png` para las imágenes de los mapas.

HTTP GET Condicional

Los HTTP GET condicionales ahorran tiempo al cliente y al servidor. Se implementa con dos respuestas en las cabeceras (Last-Modified y ETag), y dos solicitudes en la cabecera (If-Modified-Since y If-None-Match).

Existen recursos bastante populares como “restaurantes en New York”, los cuales serán solicitados por los clientes varias veces a lo largo de su existencia. Pero estos datos no cambian tan a menudo. Las imágenes satelitales son actualizadas cada ciertos meses, y lugares como restaurantes no cambian minuto a minuto. Muchas veces las peticiones realizadas luego de la primer, son una pérdida de tiempo/recurso ya que podrían utilizar la representación de la primer petición. Pero cómo pueden saber esto?

Aquí es cuando los GET condicionales entran en juego. Cuando un servidor otorga una representación, incluye un valor de tiempo para el valor de la cabecera Last-Modified. Esta es la última vez que los datos que conforman la representación fueron cambiados. Para la caminera de Estados Unidos, el Last-Modified es probable que sea la fecha en que los mapas fueron importados por primera vez al servicio. Para “restaurantes en New York”, el Last-Modified puede ser tan solo unos días atrás: cuando un restaurante fue agregada a la base de datos de lugares.

El cliente puede almacenar este valor (Last-Modified) y usarlo más adelante. Digamos que el cliente solicita la “caminería de Estados Unidos” y obtiene la siguiente respuesta:

```
Last-Modified: Thu, 30 Nov 2006 20:00:51 GMT
La siguiente vez que el cliente realice la petición GET a este recurso,
puede proveer el valor para If-Modified-Since:
GET /road/Earth HTTP/1.1
Host: maps.example.com
If-Modified-Since: Thu, 30 Nov 2006 20:00:51 GMT
```

Si los datos cambiaron entre las dos peticiones, el servidor retorna el código 200 (“OK”) y provee la nueva representación en el cuerpo de la respuesta. Pero si los datos no cambiaron, el servidor responde con el código 304 (“Sin Modificaciones”), y omite el cuerpo. Entonces el cliente sabe que puede reutilizar las representaciones que tiene almacenadas.

Que puede ir mal?

Es necesario contemplar los códigos de respuesta para errores: 3xx, 4xx, o 5xx. También es posible contemplar datos complementarios que describen el error.

Algunos errores comunes que pueden suceder:

- El cliente puede solicitar un mapa que no existe como /road/Saturn. El servidor entiende la solicitud pero no posee datos para ella. La respuesta adecuada sería 404 (“No

Encontrado"). No es necesario enviar ninguna descripción del error.

- El cliente puede utilizar un nombre de lugar que no existe en mi base de datos. El usuario final podría haber escrito mal el nombre, o utilizar un nombre que la aplicación no reconoce. Pueden haber descrito el lugar en vez de nombrarlo, podrían tener el nombre correcto, pero el planeta equivocado. O simplemente podrían estar construyendo URI con cadenas aleatorias en los mismos.

Puedo devolver un código de respuesta 404, como en el ejemplo anterior, o puedo tratar de ser útil. Si no puedo coincidir exactamente con el nombre del lugar solicitado, como `/Earth/Mount%20Rush%20more%20National%20Monument`, podría ejecutarlo a través de mi motor de búsqueda y ver si ocurre una coincidencia. Si obtengo una coincidencia, puedo ofrecer una re dirección a ese lugar: por ejemplo, `/Earth/43.9; -95.9/Mount%20Rushmore`. El código de respuesta para ser útil en este caso sería 303 ("See Other"), y el encabezado de respuesta HTTP Location contendría el URI del recurso que creo que el cliente era el que quería pedir "realmente". Si intenta realizar una búsqueda y aún no se tiene idea del lugar que el cliente está hablando, voy a retornar un código de respuesta 404 ("Not Found").

- El cliente puede utilizar latitudes lógicamente imposibles o longitudes, como 500, -181 (500 grados de latitud norte, 181 grados de longitud oeste). 404 ("Not Found") es una buena respuesta aquí, ya que es un lugar que no existe. Sin embargo, un 400 ("Bad Request") sería más preciso.

¿Cuál es la diferencia entre los dos casos? Bueno, no hay nada obviamente mal con una solicitud de un nombre de lugar inexistente como "Tanhoidfog." Alguien podría nombrar una ciudad o una empresa "Tanholdfog" y entonces sería un lugar válido. El cliente no sabe que no hay tal lugar.

- La búsqueda de lugares en un mapa podría no devolver resultados. Puede no haber autódromos cerca de Sebastopol, CA. Puedo tratar esto como cualquier otra búsqueda: enviar un código de respuesta 200 ("OK") y una representación. La representación incluirá un enlace al lugar que se buscó, junto con una lista vacía de resultados de búsqueda.
- El servidor puede estar sobrecargado con solicitudes y ser incapaz de cumplir con esta solicitud en particular. El código de respuesta es 503 ("Service Unavailable"). Una alternativa consiste en negarse a atender la solicitud en absoluto.
- El servidor no funciona correctamente. Esto podría ser debido a datos faltantes o dañados, un error de software, un fallo de hardware, o cualquiera de las otras cosas que pueden salir mal con un programa informático. En este caso el código de respuesta es 500 ("Internal Server Error"). Este es un código de respuesta frustrante porque no hay nada que el cliente puede hacer al respecto. Muchos frameworks envían automáticamente este código de error cuando una excepción ocurre en el lado del servidor.

Conclusión

Tenemos ahora un diseño de un web service de mapas bastante simple para que sea usado por clientes sin requerir mucho trabajo, y suficientemente útil para ser utilizado por varios programas.

A la vez es de sólo lectura. Asume que los clientes no tienen nada que ofrecer, salvo su apetito por consumir datos. Muchos web services funcionan de esta manera, pero no son los únicos. Existen otros donde los clientes pueden utilizar interfaces uniformes para crear sus propios recursos.