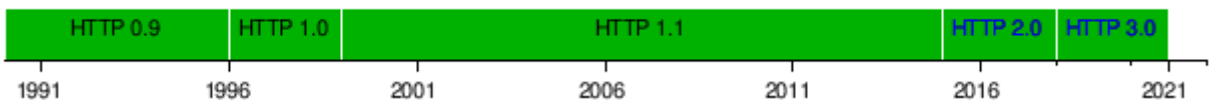
Tema 1. Desarrollo de APIs REST con Spring Boot.

1. Protocolo HTTP

El Protocolo de transferencia de hipertexto (en inglés, Hypertext Transfer Protocol, abreviado HTTP) es el protocolo de comunicación que permite las transferencias de información a través de archivos (XHML, HTML . . .) en la World Wide Web.

HTTP define la sintaxis y la semántica que utilizan los elementos de software de la arquitectura web (clientes, servidores, proxies) para comunicarse.

HTTP es un protocolo sin estado, es decir, no guarda ninguna información sobre conexiones anteriores. El desarrollo de aplicaciones web necesita frecuentemente mantener estado. Para esto se usan las cookies, que es información que un servidor puede almacenar en el sistema cliente. Esto le permite a las aplicaciones web instituir la noción de sesión, y también permite rastrear usuarios ya que las cookies pueden guardarse en el cliente por tiempo indeterminado.



Actualmente, podemos llegar a encontrar servidores trabajando aun en HTTP/1.1 o HTTP/1.2 (Feb 2000), aunque principalmente deberían estar haciéndolo en HTTP/2 (May 2015). También es posible que encontremos algún servidor trabajando ya en HTTP/3 (Oct 2018).

1.1 Descripción

Es un protocolo orientado a transacciones y sigue el esquema petición-respuesta entre un cliente y un servidor. El cliente (se le suele llamar "agente de usuario", en inglés *user agent*) realiza una petición enviando un mensaje, con cierto formato al servidor. El servidor (se le suele llamar un servidor web) le envía un mensaje de respuesta. El ejemplo más claro de cliente son los navegadores web, aunque también lo podría ser un televisor o un dispositivo IoT.

1.2 Mensajes

Los mensajes tienen la siguiente estructura:

- Línea inicial (termina con un retorno de carro y salto de línea) con:
  - Peticiones: acción requerida al servidor, URL del recurso y versión HTTP soportada.
  - Respuestas: versión HTTP usada, código de respuesta y la URL del recurso.
- Las cabeceras, que terminan con una línea en blanco. Son metadatos que aportan gran flexibilidad al protocolo.
- Cuerpo del mensaje. Es opcional. Su presencia depende de la línea anterior del mensaje y del tipo de recurso al que hace referencia la URL. Típicamente tiene los datos que se intercambian cliente y servidor. Por ejemplo para una petición podría contener ciertos datos que se quieren enviar al servidor para que los procese. Para una respuesta podría incluir los datos que el cliente ha solicitado.

1.3 Métodos (*a.k.a.* verbos)

HTTP define una serie predefinida de métodos de petición (algunas veces referido como "verbos") que pueden utilizarse. El protocolo tiene flexibilidad para ir añadiendo nuevos métodos y para así añadir nuevas funcionalidades. El número de métodos de petición se ha ido aumentando según se avanzaba en las versiones.

Los más conocidos son:

- `GET`: solicita un recurso al servidor
- `POST`: envía información para crear un nuevo recurso
- `PUT`: actualiza un recurso de forma completa
- `DELETE`: borra un recurso

> En este enlace puedes consultar la lista de verbos HTTP y la versión o RFC en la que fueron incluidos.

1.4 Códigos de respuesta

El código de respuesta o retorno es un número que indica que ha pasado con la petición. El resto del contenido de la respuesta dependerá del valor de este código. El sistema es flexible y de hecho la lista de códigos ha ido aumentando para así adaptarse a los cambios e identificar nuevas situaciones. Cada código

tiene un significado concreto. Sin embargo el número de los códigos están elegidos de tal forma que según si pertenece a una centena u otra se pueda identificar el tipo de respuesta que ha dado el servidor:

- Códigos con formato 1xx: Respuestas informativas. Indica que la petición ha sido recibida y se está procesando.
- Códigos con formato 2xx: Respuestas correctas. Indica que la petición ha sido procesada correctamente.
- Códigos con formato 3xx: Respuestas de redirección. Indica que el cliente necesita realizar más acciones para finalizar la petición.
- Códigos con formato 4xx: Errores causados por el cliente. Indica que ha habido un error en el procesado de la petición a causa de que el cliente ha hecho algo mal.
- Códigos con formato 5xx: Errores causados por el servidor. Indica que ha habido un error en el procesado de la petición a causa de un fallo en el servidor.

1.5 Cabeceras

Son los metadatos que se envían en las peticiones o respuesta HTTP para proporcionar información esencial sobre la transacción en curso. Cada cabecera es especificada por un nombre de cabecera seguido por dos puntos, un espacio en blanco y el valor de dicha cabecera seguida por un retorno de carro seguido por un salto de línea. Se usa una línea en blanco para indicar el final de las cabeceras. Si no hay cabeceras la línea en blanco debe permanecer.

Las cabeceras le dan gran flexibilidad al protocolo permitiendo añadir nuevas funcionalidades sin tener que cambiar la base. Por eso según han ido sucediendo las versiones de HTTP se han ido añadiendo más y más cabeceras permitidas.

Las cabeceras pueden tener metadatos que tienen que ser procesados por el cliente (ej. en respuesta a petición se puede indicar el tipo del contenido que contiene), por el servidor (ej. tipos de representaciones aceptables por el cliente del contenido que pide) o por los intermediarios (ej. como gestionar el cacheo por parte de los proxys)

Dependiendo del tipo de mensaje en el que puede ir una cabecera las podemos clasificar en cabeceras de petición, cabeceras de respuesta y cabeceras que pueden ir tanto en una petición como en una respuesta.

Podemos clasificar las cabeceras según su función. Por ejemplo:

- Cabeceras que indican las capacidades aceptadas por quién envía el mensaje: *Accept*, *Accept-Charset*, *Accept-Encoding*, ...
- Cabeceras que describen el contenido: *Content-Type*, *Content-Length*, ...
- Cabeceras que hacen referencias a URIs: *Location*, *Refer*.
- Cabeceras que permiten ahorrar transmisiones: *Date*, *Cache-Control*, *Age*, *Last-Modified*, ...
- Cabeceras para el control de cookies: *Set-Cookie*, *Cookie*
- Cabeceras para la autenticación: *Authorization*, *WWW-Authenticate*,
- ...

1.5 Ejemplo de diálogo HTTP

Para obtener un recurso con el URL http://www.example.com/index.html

Se abre una conexión en el puerto 80 del host www.example.com. El puerto 80 es el puerto predefinido para HTTP. Si se quisiera utilizar el puerto XXXX habría que codificarlo en la URL de la forma http://www.example.com:XXXX/index.html Se envía un mensaje en el estilo siguiente:

```
 GET /index.html HTTP/1.1
 Host: www.example.com
 Referer: www.google.com
 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:45.0) Gecko/20100101
Firefox/45.0
 Connection: keep-alive
 [Línea en blanco]
```

La respuesta del servidor está formada por encabezados seguidos del recurso solicitado, en el caso de una página web:

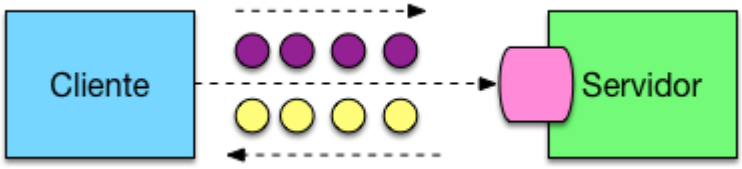
```
HTTP/1.1 200 OK
Date: Fri, 31 Dec 2003 23:59:59 GMT
Content-Type: text/html
Content-Length: 1221

<html lang="eo">
<head>
<meta charset="utf-8">
<title>Título del sitio</title>
</head>
<body>
<h1>Página principal de tuHost</h1>
(Contenido)
  .
  .
  .
</body>
</html>
```

2. REST

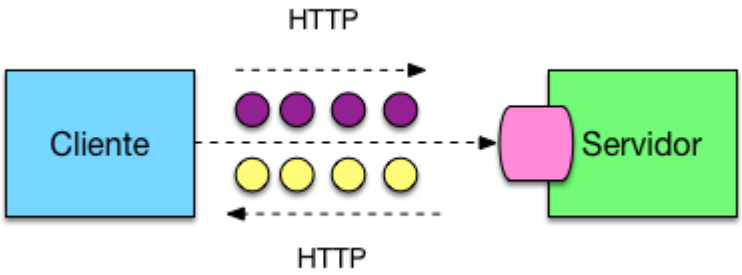
2.1 Introducción

¿Qué es REST? Esta pregunta es una de las más habituales en nuestros días. Para algunas personas REST es una arquitectura , para otras es un patrón de diseño , para otras un API. **¿Qué es REST exactamente?**. REST o *Representational State Transfer* es un **ESTILO** de Arquitectura a la hora de realizar *una comunicación entre cliente y servidor*.

Vamos a intentar explicarlo esto paso a paso . Habitualmente cuando nosotros realizamos una comunicación cliente servidor accedemos al servidor en un punto de acceso , le enviamos una información y recibimos un resultado.



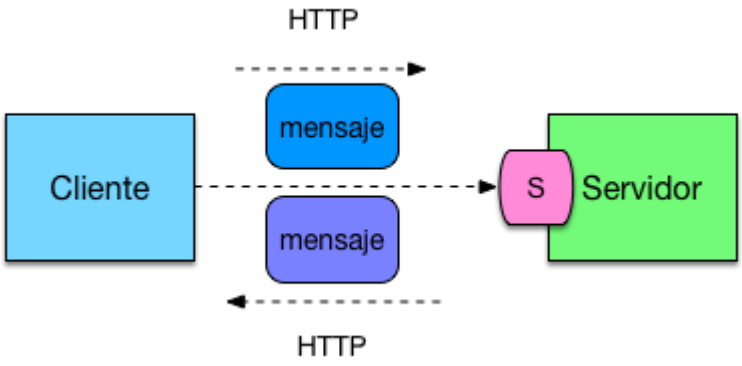
Ahora bien hay muchas formas de realizar esta operación. **¿Cuál es la más correcta?**. Esa es una buena pregunta. Hoy por hoy una de las necesidades más claras es que esa comunicación sea abierta y **podemos acceder desde cualquier sitio**. Así pues estamos hablando de una comunicación **HTTP** (*Hyper Text Transfer Protocol*) . Usamos el puerto 80 de nuestro servidor para permitir el acceso a la información desde cualquier lugar.



> **¡OJO!** HTTP no es el único protocolo de comunicación, ni el primero, ni podríamos que decir que el mejor. Tiene una serie de características que han hecho que se haya impuesto a otros, pero éstos se pueden seguir utilizando. **Investiga otros protocolos como RMI o Corba**.

Una vez tenemos claro el protocolo de comunicación el siguiente paso es decidir **qué tipología de mensajes enviamos**. Como punto de partida podemos mandar a un servicio un mensaje **en formato XML o JSON**. El servicio lo recepcionará y nos devolverá una respuesta.

> Hablaremos de JSON y XML más adelante.

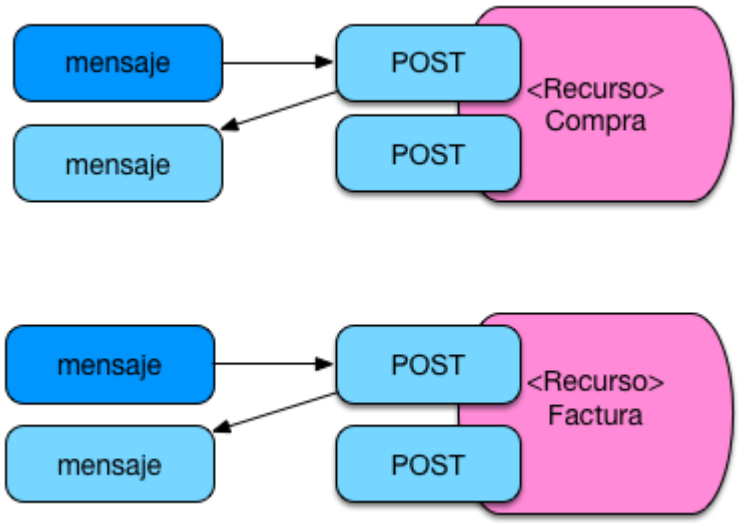


Esto es lo que habitualmente en **Arquitecturas REST se denomina el nivel 0**. No tenemos ningún tipo de organización. Es el caos pero accedemos a la información en formato de dato puro.

2.2 Niveles

2.2.1 Nivel 1: Recursos

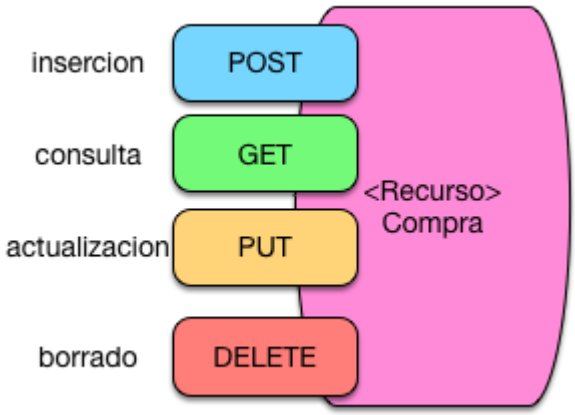
El siguiente paso es lo que se denomina nivel 1 , en vez de tener servicios con métodos diversos declaramos Recursos. ¿Qué es un Recurso? . Un recurso hace referencia a un concepto importante de nuestro negocio (Facturas ,Cursos , Compras etc). Es lo que habitualmente se denomina un objeto de negocio. Este estilo permite un primer nivel de organización permitiendo acceder a cada uno de los recursos de forma independiente, favoreciendo la reutilización , aumentando la flexibilidad y abordando operaciones de inserción ,borrado , busqueda etc.



2.2.2 Nivel 2: Verbos HTTP

Hasta este momento para realizar las peticiones y enviar datos se usan GET o POST indistintamente . En el nivel 2 las operaciones **pasan a ser categorizadas de forma más estricta**. Dependiendo de cada tipo de operación se utilizará **un método diferente de envío**.

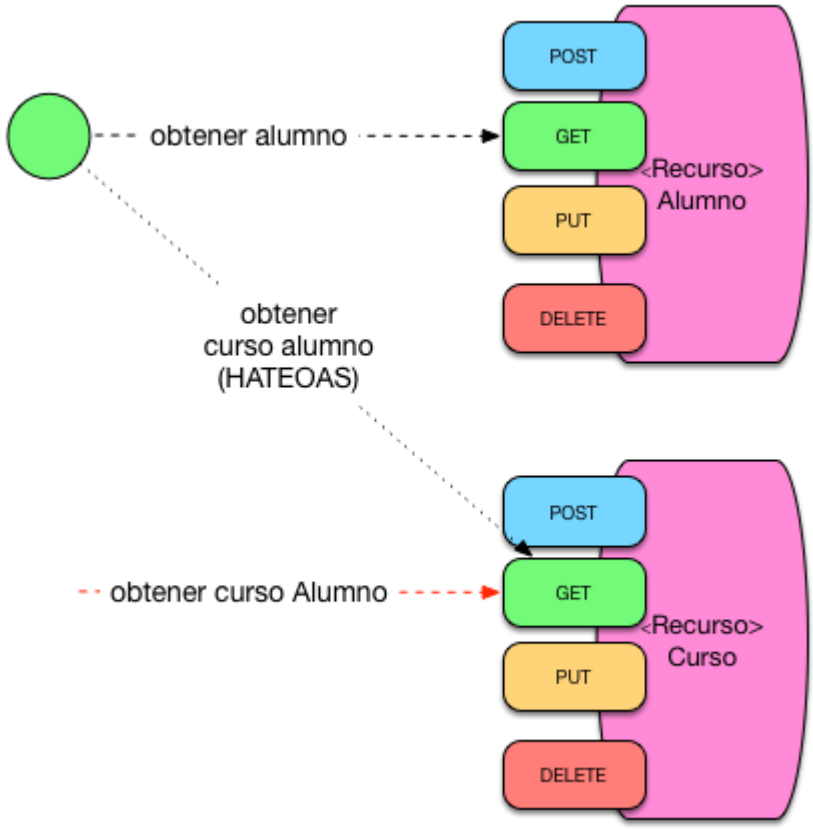
- **GET**: Se usara para solicitar consultar a los recursos
- **POST**: Se usará para insertar nuevos recursos
- **PUT** : Se usará para actualizar recursos
- **DELETE** : Se usará para borrar recursos



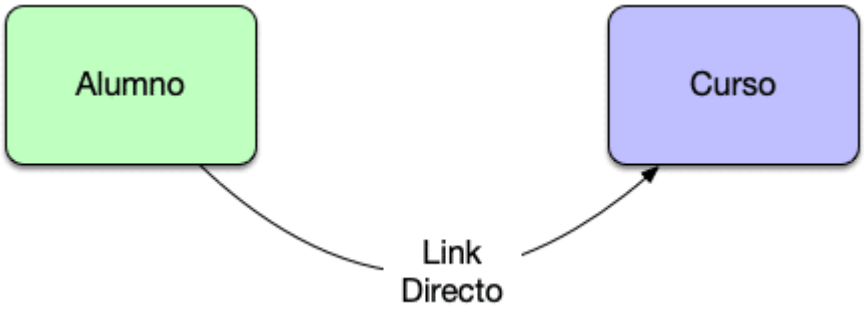
Este es el nivel en el que hoy en día se encuentran muchas de las Arquitecturas REST.

2.2.3 Nivel 3: HATEOAS, el cielo del REST

Todavía nos queda un nivel más que es el que se denomina **HATEOAS (Hypertext As The Engine Of Application State)**. ¿Para qué sirve este nivel?. Supongamos que queremos acceder a un recurso Alumno via REST . Si tenemos una *Arquitectura a nivel 2* primero accederemos a ese recurso **utilizando GET**. En segundo lugar deberemos acceder al recurso de Cursos para añadir al alumno **al curso (linea roja)** .



Esto implica que el cliente que accede a los servicios REST *asume un acoplamiento muy alto*, **debe conocer la url del Alumno y la del Curso**. Sin embargo **si el recurso del Alumno contiene un link al recurso de Curso esto no hará falta** y desde la url de Alumno directamente podremos acceder a la url del curso que pertenece a ese alumno.



Podríamos tener una estructura JSON como la siguiente:

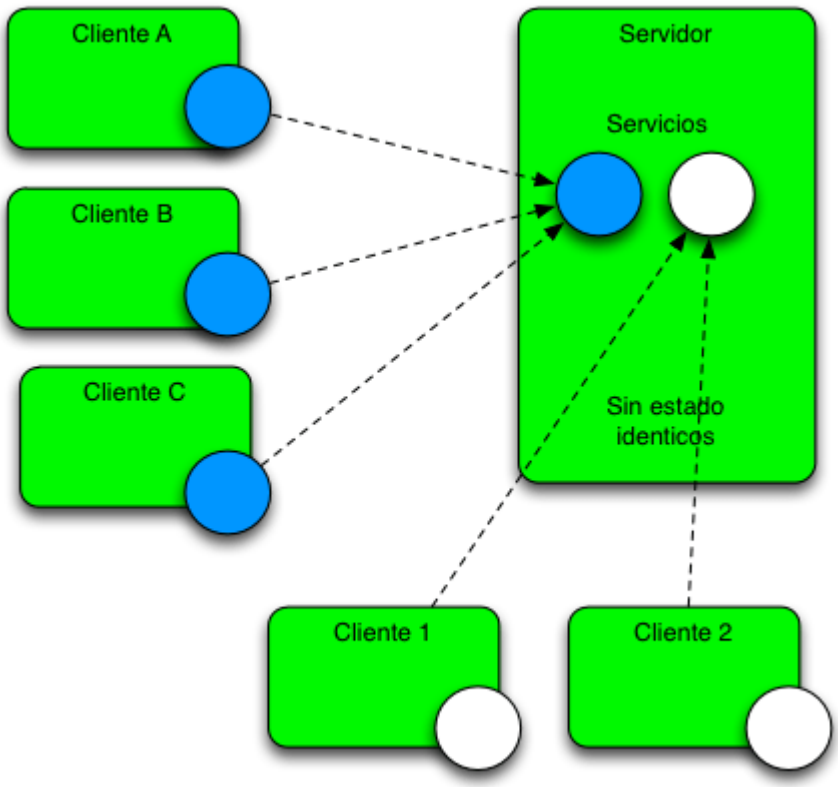
```
{
    nombre:”pedro”,
    apellidos:”gomez”,
    cursos:”http://miurl/cursos”
}
```

Podremos acceder directamente al recurso de Curso utilizando las propiedades del Alumno esto es **HATEOAS**. De esta forma se aumenta la flexibilidad **y se reduce el acoplamiento**. Construir arquitecturas sobre estilo REST no es sencillo y hay que **ir paso a paso**.

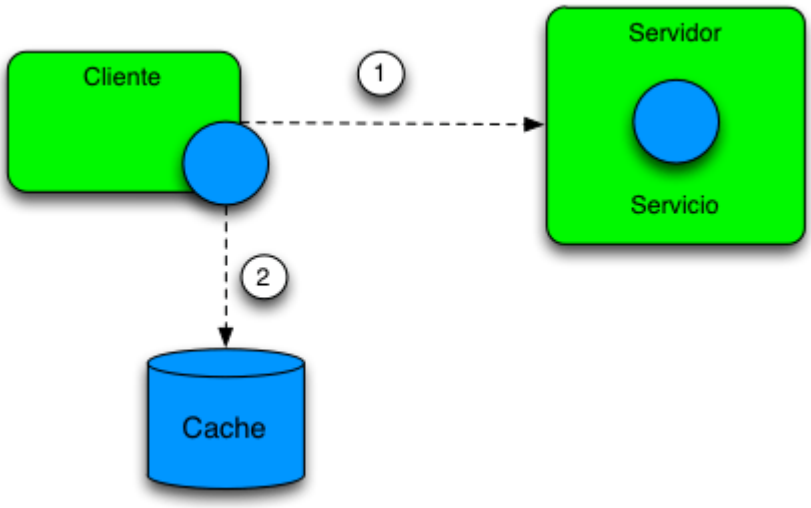
2.3 Premisas a cumplir

Los servicios REST que siguen este estilo deben cumplir algunas premisas:

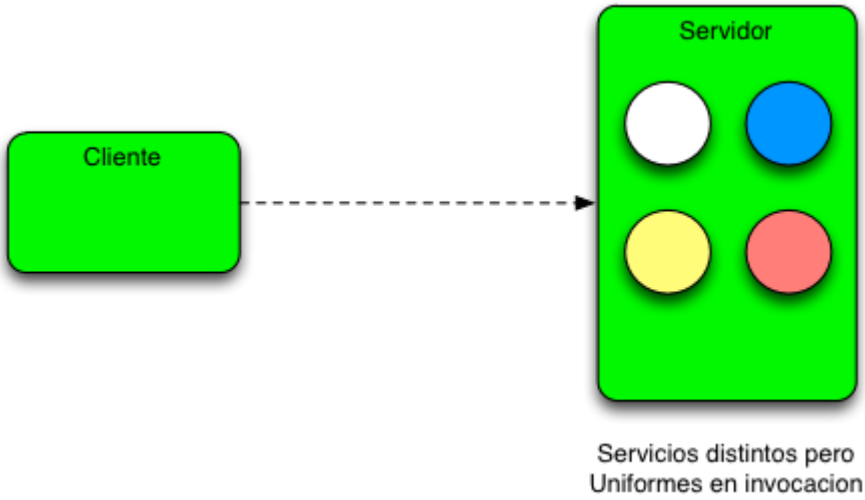
- **Cliente/Servidor**: Como servicios web son cliente servidor y definen un interface de comunicación entre ambos separando completamente las responsabilidades entre ambas partes.
- **Sin estado**: Son servicios web que no mantienen estado asociado al cliente .Cada petición que se realiza a ellos es completamente independiente de la siguiente . Todas las llamadas al mismo servicio serán idénticas.



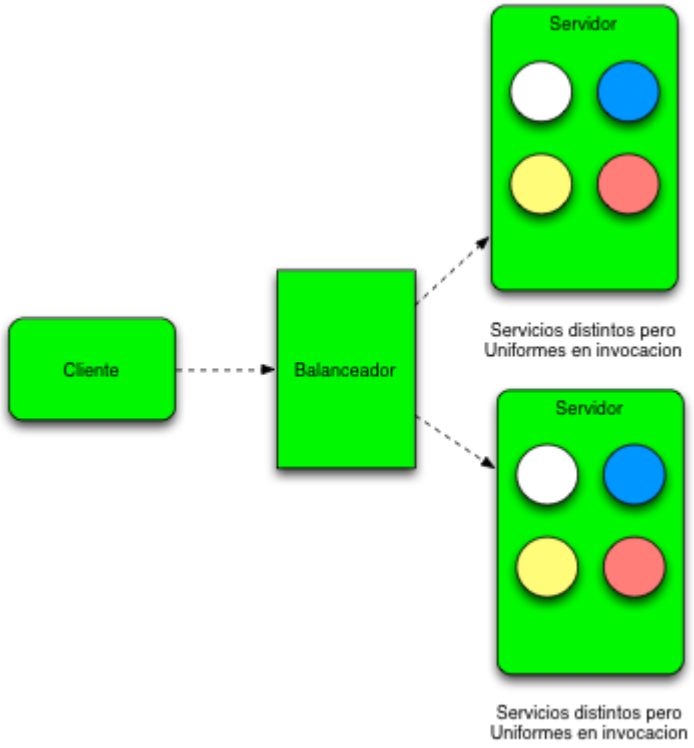
- **Caché**: El contenido de los servicios web REST ha se puede cachear de tal forma que una vez realizada la primera petición al servicio el resto puedan apoyarse en la cache si fuera necesario.



- **Servicios Uniformes**:Todos lo servicios REST compartirán una forma de invocación y métodos uniforme utilizando los metodos `GET`, `POST`, `PUT`, `DELETE`.



- **Arquitectura en Capas**: Todos los servicios REST estan orientados hacia la escalabilidad y un cliente REST no será capaz de distinguir entre si esta realizando una petición directamente al servidor, o se lo esta devolviendo un sistema de caches intermedio o por ejemplo existe un balanceador que se encarga de redirigirlo a otro servidor.



3. JSON

JavaScript Object Notation (JSON) es un formato basado en texto estándar para representar datos estructurados en la sintaxis de objetos de JavaScript. Es comúnmente utilizado para transmitir datos en aplicaciones web (por ejemplo: enviar algunos datos desde el servidor al cliente, así estos datos pueden ser mostrados en páginas web, o viceversa). Te enfrentarás a menudo con él, así que vamos a aprender un poco más sobre este formato.

3.1 ¿Qué es realmente JSON?

JSON es un formato de datos basado en texto que sigue la sintaxis de objeto de JavaScript. Aunque es muy parecido a la sintaxis de objeto literal de JavaScript, puede ser utilizado independientemente de JavaScript, y muchos entornos de programación poseen la capacidad de leer (convertir; parsear) y generar JSON.

Los JSON son cadenas de caracteres, y son muy útiles para transmitir datos por la red.

Un objeto JSON puede ser almacenado en su propio archivo, que es básicamente sólo un archivo de texto con una extension .json, y un *MIME type* de `application/json`.

> **¿Qué tal si investigas qué es eso de los tipos MIME?**

3.2 Estructura de JSON

JSON es una cadena cuyo formato recuerda al de los objetos literales JavaScript. Es posible incluir los mismos tipos de datos básicos dentro de un JSON que en un objeto estándar de JavaScript: cadenas, números, arrays, booleanos, y otros literales de objeto. Esto permite construir una jerarquía de datos, como ésta:

```
{
  "squadName": "Super hero squad",
  "homeTown": "Metro City",
  "formed": 2016,
  "secretBase": "Super tower",
  "active": true,
  "members": [
    {
      "name": "Molecule Man",
      "age": 29,
      "secretIdentity": "Dan Jukes",
      "powers": [
        "Radiation resistance",
        "Turning tiny",
        "Radiation blast"
      ]
    },
    {
      "name": "Madame Uppercut",
      "age": 39,
      "secretIdentity": "Jane Wilson",
      "powers": [
        "Million tonne punch",
        "Damage resistance",
        "Superhuman reflexes"
      ]
    },
    {
      "name": "Eternal Flame",
      "age": 1000000,
      "secretIdentity": "Unknown",
      "powers": [
        "Immortality",
        "Heat Immunity",
        "Inferno",
        "Teleportation",
        "Interdimensional travel"
      ]
    }
  ]
}
```

3.2.1 Arrays en JSON

Anteriormente hemos mencionado que el texto JSON básicamente se parece a un objeto JavaScript, y esto es en gran parte cierto. La razón de esto es que un array de Javascript es también un JSON válido, por ejemplo:

```
[
  {
```

```
    "name": "Molecule Man",
    "age": 29,
    "secretIdentity": "Dan Jukes",
    "powers": [
      "Radiation resistance",
      "Turning tiny",
      "Radiation blast"
    ]
  },
  {
    "name": "Madame Uppercut",
    "age": 39,
    "secretIdentity": "Jane Wilson",
    "powers": [
      "Million tonne punch",
      "Damage resistance",
      "Superhuman reflexes"
    ]
  }
]
```

3.3 Sintaxis

Los tipos de datos disponibles con JSON son:

- Números: Se permiten números negativos y opcionalmente pueden contener parte fraccional separada por puntos. Ejemplo: `123.456`
- Cadenas: Representan secuencias de cero o más caracteres. Se ponen entre doble comilla y se permiten cadenas de escape. Ejemplo: `"Hola"`
- Booleanos: Representan valores booleanos y pueden tener dos valores: `true` y `false`
- `null`: Representan el valor nulo.
- Array: Representa una lista ordenada de cero o más valores los cuales pueden ser de cualquier tipo. Los valores se separan por comas y el vector se mete entre corchetes. Ejemplo `["juan","pedro","jacinto"]`
- Objetos: Son colecciones no ordenadas de pares de la forma : separados por comas y puestas entre llaves. El nombre tiene que ser una cadena entre comillas dobles. El valor puede ser de cualquier tipo. Ejemplo:

```
{
  "departamento":8,
  "nombredepto":"Ventas",
  "director": "Juan Rodríguez",
  "empleados":[
    {
      "nombre":"Pedro",
      "apellido":"Fernández"
    },{
      "nombre":"Jacinto",
      "apellido":"Benavente"
    }
```

```
    ]
}
```

> En esta web son JSON (https://www.json.org/json-es.html) puedes encontrar algunos gráficos que explican muy bien la sintaxis de JSON.

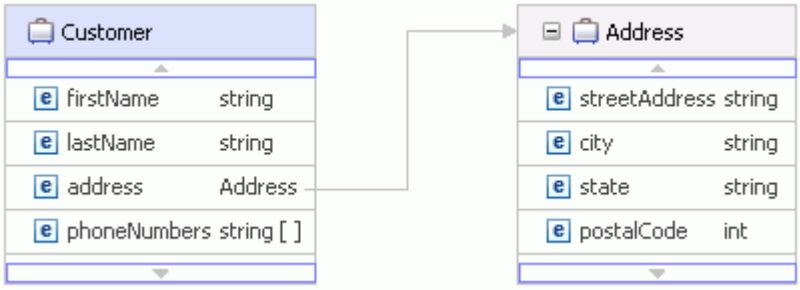
3.4 JSON en Java

Hemos podido leer anteriormente que JSON es una notación de objetos que viene de Javascript, y por tanto, en dicho lenguaje se puede trabajar con este formato de forma nativa. Pero, ¿qué pasa con otros lenguajes, como por ejemplo Java?

Java ofrece múltiples formas de trabajar con JSON. La idea principal que subyace en todas estas formas de trabajar son dos procesos:

- Parsear de objeto a JSON: significa que queremos transformar un objeto (Java) en una cadena JSON. Esto será útil, por ejemplo, si queremos enviar a través de HTTP los datos que hemos recogido de un formulario web.
- Parsea de JSON a objeto: significa que queremos transformar un JSON en un objeto (Java). Esto por ejemplo, será útil para recibir información a través de HTTP, si queremos almacenar la misma en nuestra base de datos como una instancia de una entidad.

El *mapeo* entre objetos (de Java o cualquier otro lenguaje de POO) no es compleja. Podemos ver un par de ejemplos con las siguientes clases:



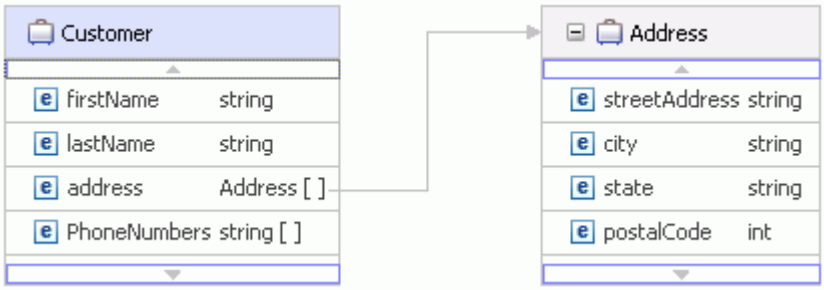
Aquí podemos ver que tenemos un objeto que representa a un cliente, con algunos datos básicos, un objeto anidado, y un array de cadenas de caracteres.

Su representación en JSON sería la siguiente:

```
{
    "firstName": "John",
    "lastName": "Smith",
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": 10021
    },
    "phoneNumbers": [
        "212-732-1234",
```

```
            "646-123-4567"
        ]
    }
```

Si por contra, tenemos un objeto con un array de objetos anidados:



Su representación en JSON sería la siguiente:

```
    {
        "firstName": "John",
        "lastName": "Smith",
        "address": [{
            "streetAddress": "21 2nd Street",
            "city": "New York",
            "state": "NY",
            "postalCode": 10021
        },{
            "streetAddress": "577 Airport Blvd",
            "city": "Burlingame",
            "state": "CA",
            "postalCode": 94010
        }],
        "phoneNumbers": [
            "212-732-1234",
            "646-123-4567"
        ]
    }
```

Bien, ahora viene la pregunta clave: ¿cómo hacemos este *parseo* tanto de JSON a Java como de Java a JSON? Java no incluye clases *propias* para el procesamiento de JSON, así que tendremos que acudir a librerías de terceros.

- La librería `org.json` es una que nos permite trabajar a bastante bajo nivel. Para usarla, podríamos añadir la siguiente dependencia Maven:

```
<dependency>
    <groupId>org.json</groupId>
    <artifactId>json</artifactId>
    <version>20180130</version>
</dependency>
```

Ofrece algunas clases, como `JSONObject`, `JSONArray`, ... que permiten realizar todas las operaciones que necesitemos.

Como podrás suponer, este enfoque es trabajoso, tedioso, y no asumible para un modelo de datos complejo.

- La librería `Jackson` (que será la que utilicemos) nos permitirá hacer ese proceso de parseo o transformación de forma *automágica*. Así, nosotros solamente tendremos que seguir estos pasos:

    1. Definir los objetos Java necesarios para trabajar con las cadenas JSON que queramos utilizar.
    2. Si queremos transformar de Java a JSON, tan solo necesitamos instanciar un objeto de nuestro modelo, darle los valores oportunos, y pedirle a Jackson que lo transforme.
    3. Si lo que queremos es transformar de JSON a Java, lo que le indicaremos que el tipo de objeto de destino, y Jackson se encargará de todo.

Jackson es la librería escogida, no solamente porque sea buena, sino porque viene integrada directamente con Spring Boot.

> Existen otras librerías en el mercado, como GSON, de Google.

Bibliografía

- HTTP. Wikipedia (https://es.wikipedia.org/wiki/Protocolo_de_transferencia_de_hipertexto)
- ¿Qué es REST? (https://www.arquitecturajava.com/que-es-rest/)
- Trabajando con JSON (https://developer.mozilla.org/es/docs/Learn/JavaScript/Objects/JSON)
- JSON. Wikipedia (https://es.wikipedia.org/wiki/JSON)