UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

Departamento de Informática
Universidad Técnica Federico Santa María

INTERNACIONAL

Trabajo Final

MTI 476 - Sistemas y Aplicaciones Web

José Emilio Labra

Bastian Carvajal Yañez
bastian.carvajal@sansano.usm.cl (+56971420090)

Santiago, 21 de mayo de 2017

Enunciado

Se desea desarrollar la infraestructura de un portal Web de compra/venta de instrumentos musicales llamado QUENA. El portal dispone de una serie de establecimientos asociados repartidos a nivel mundial así como productos de dichas regiones que se desea vender a través de Internet. El portal Web gestionará la información de cada establecimiento así como los productos disponibles en cada una de ellos.

Las tareas a realizar serán:

1. Representación de la información.
   1.1. Representar la información de los establecimientos, los productos y sus características, precios, disponibilidad, etc. en ficheros XML y JSON. No es necesario representar toda la información en un único fichero XML/Json, si no que pueden utilizarse varios.
   1.2. Crear vocabularios mediante DTD, XML Schema para validar los ficheros XML de la sección anterior. Opcionalmente, crear ficheros JSON Schema para validar la representación Json.
   1.3. Realizar una comparación entre el formato XML y el formato JSON, indicando las ventajas e inconvenientes de cada uno de ellos. Incluir una comparación entre las capacidades expresivas de los lenguajes de esquema para XML y los lenguajes de esquema para JSON.
2. Aplicación Web.
   2.1. Crear un servicio Web REST para gestionar la información de los establecimientos, así como de los productos disponibles.
   2.2. El servicio Web deberá devolver la información de los establecimientos en los formatos XML, JSON y HTML. Puede realizarse mediante URIs diferentes o mediante negociación de contenido. Documentar brevemente el servicio Web creado, los métodos y la forma de invocarlos (se recomienda la utilización de Swagger para la documentación). Crear una aplicación Web cliente del servicio Web anterior.
   2.3. Realizar una valoración de la solución comparando el framework NodeJs con otros frameworks Web. Analizar posibles soluciones para desplegar el sistema desarrollado: servidores locales, servidores en la nube, microservicios, etc. teniendo en cuenta diversos aspectos: precio, disponibilidad, escalabilidad, etc.
3. Web Semántica
   3.1. Representar la información de establecimientos y productos musicales de la primer tarea en formato RDF. Opcionalmente, se podrá utilizar ShEx o SHACL para validar los ficheros creados (una posible herramienta sería para la validación sería RDFShape o Shaclex).

3.2. Existen 2 tecnologías que permiten incorporar metadatos en páginas Web: microdatos y RDFa. Enriquecer las páginas Web de establecimientos y productos musicales con microdatos ó con RDFa para que sus contenidos puedan ser indexados por buscadores. Utilizar para ello elementos del vocabulario Schema.org.

3.3. Analizar la solución presentada, así como las ventajas e inconvenientes de enriquecer portales Web con tecnologías semánticas. Este análisis puede llevarse a cabo utilizando uno o más casos de uso propuestos por los estudiantes. Se valorarán casos de uso reales o en los que los estudiantes tengan experiencia previa.

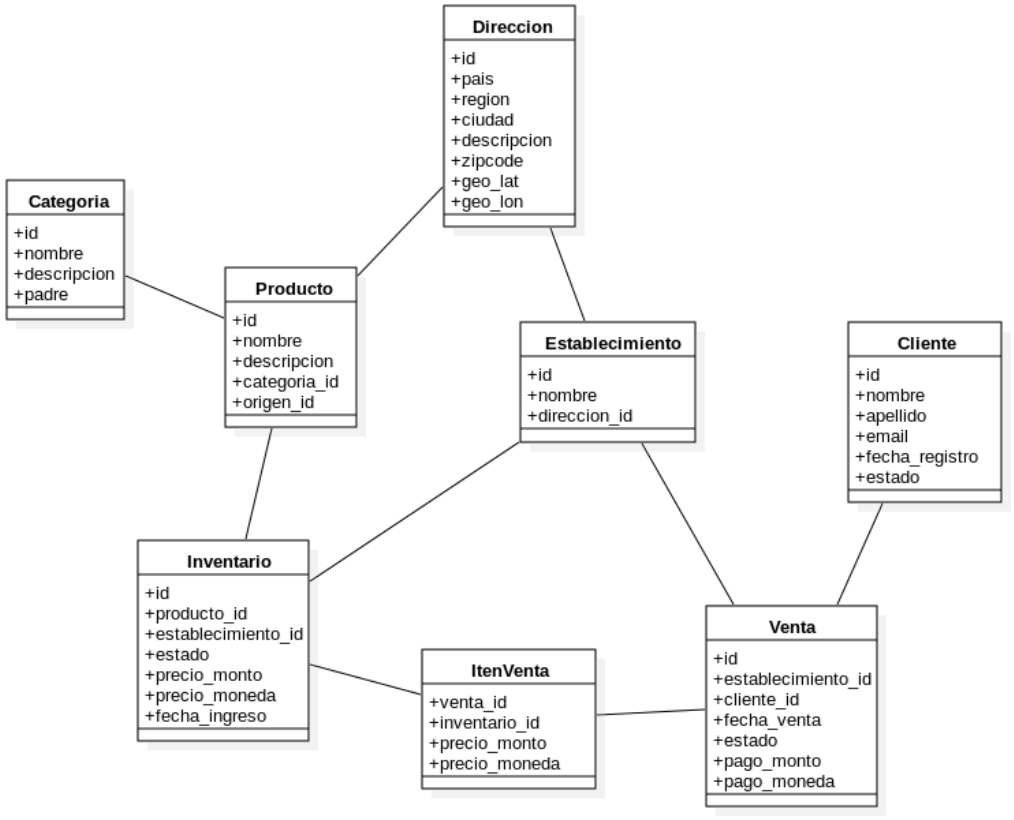
Desarrollo

A continuación se presenta el desarrollo de cada una de las tareas antes mencionadas. En algunos casos se hace referencia a archivos externos, estos se entregan adjuntos a este documento y también estarán disponibles en un repositorio de *github*.

Este documento y todo el código fuente generado para este proyecto puede encontrarse en la siguiente dirección: **https://github.com/bastiancy/quena-mti**

1.1 y 1.2

Para representar la información de establecimientos, productos, se realizó el siguiente diagrama de clases.



La representación en **xml** se divide en dos archivos, el primero incluye la definición del **dtd** para validar el documento, y el segundo que tiene separado el archivo **xsd** para validar a través de **xml schema**:

- dtd
  - parte1/xml/store1.xml
- xml schema
  - parte1/xml/store2.xml
  - parte1/xml/store2.xsd

Para validar estos archivos se utiliza xmllint en un entorno linux. Los siguientes comandos se pueden ejecutar en el directorio /parte1/xml/.

```
# Es válido si no retorna ningun mensaje.
xmllint --valid store1.xml --noout

# Es válido si retorna "store2.xml validates"
xmllint --schema store2.xsd --noout store2.xml
```

La representación en **json** tiene algunas variaciones respecto de la anterior. Igualmente se valida a través de **json schema**.

- parte1/json/store.json
- parte1/json/store.schema.json

Se creó un script en **Node.js** para validar este archivo. Los siguientes comandos se pueden ejecutar en el directorio /parte1/json/.

```
# instalar dependencias (probado con node v4.8.1 y npm v4.4.4)
npm install

# Es válido si no retorna ningun mensaje.
node validator.js store.json store.schema.json
```

1.3

Ambos formatos XML y JSON han sido utilizados extensivamente para el intercambio de información en Internet. Especialmente en web services, donde la tendencia en este último tiempo ha sido generar APIs más simples, donde algunos desarrolladores prefieren JSON para el intercambio de datos.

En XML (del inglés *eXtensible Markup Language*, o traducido como *Lenguaje de marcado extensible*) la intención es tomar *texto plano* y agregar "marcado" para crear un texto estructurado. Este *meta-lenguaje* (derivado del SGML) nació en IBM por la necesidad de estructurar documentos grandes, almacenando los datos en forma legible, para poder ser compartidos con otros SO.

En cambio, JSON fue diseñado para serializar datos en JavaScript, de aquí su nombre "*JavaScript Object Notation*". Es por esta razón que ha sido naturalmente aceptado por desarrolladores web.

Una de las mayores diferencias es que XML soporta *mixed content*, es decir, permite mezclar etiquetas de nodos dentro de *strings* literales, por ejemplo:

```
<p>An <b>inline tag</b> in the text!</p>
```

Mientras que JSON no soporta esta característica; en JavScript un string literal no es interpretado por el compilador. De la misma forma, JSON tampoco soporta etiquetas para agregar comentarios como en XML, esto fue de hecho una decisión de diseño, como lo ha expresado Douglas Crockford:

> I removed comments from JSON because I saw people were using them to hold parsing directives, a practice which would have destroyed interoperability.
> -- Douglas CrockFord

Respecto de la validación de documentos con JSON Schema o XML Schema, en ambos casos es posible definir reglas acerca de los tipos de datos soportados, y la estructura a la que el documento debe apegarse. Comparten en esencia las mismas ventajas y desventajas desde el punto de vista de su sintaxis, ya que se definen en JSON y XML respectivamente.

Algunas de estas ventajas e inconvenientes son:

JSON

- Ventajas:
  - Sintaxis más reducida, ya que no necesita de etiquetas especiales para definir los nodos.
  - Facil integracion con JavaScript puesto que es la forma estándar de serializar los objetos.
  - Puede usar *JSON Schema* para la descripción y validación de los tipos de datos y la estructura del documento.

    - Puede usar *JsonPath* para la extracción de información en estructuras profundamente anidadas.
- Desventajas:
    - Estructura no es autodescriptiva y soporta pocos tipos de datos diferentes.
    - No permite agregar comentarios en el documento.
    - No soporta el uso de *namespaces*.

XML

- Ventajas:
    - Su sintaxis se basa en etiquetas para definir cada nodo, lo que permite representar estructuras diferentes en el mismo documento.
    - Los nodos pueden ser auto descriptivos al definir propiedades en la etiqueta.
    - Se pueden agregar comentarios al documento.
    - Puede usar *XML Schema* o *DTDs* para validación de datos y estructura.
    - Se puede usar *XSLT* para transformar a diferentes formatos de salida.
    - Puede usar *XPath/XQuery* para información en estructuras profundamente anidadas.
    - Soporte incluido de *namespaces*.
- Desventajas:
    - Sintaxis más compleja o más *verbose* (las etiquetas se repiten al cierre del nodo). Los documentos terminan usando más datos para la misma cantidad de información.
    - Puede ser menos legible para un humano en estructuras anidadas.
    - Puede ser más lento de interpretar por un compilador, y no tiene soporte nativo en JavaScript.

2.1 y 2.2

El servicio REST ha sido desarrollado en **nodejs**, y utiliza **MongoDB** (base de datos no relacional) para la persistencia de datos. El api ha sido documentada con **swagger**, lo cual permite interactuar con ella desde el mismo navegador (usando swagger-ui).

Esta api implementa *negociación de contenido* y puede servir los siguientes formatos: **json**, **xml**, **html**. Esta solo consume datos en **json**. Se puede acceder públicamente en las siguientes URLs:

- API: http://api.quenamti.bastianc.info/
- Documentacion: http://api.quenamti.bastianc.info/docs/

También se ha desarrollado un cliente para consumir el API antes mencionada, este cliente consiste en una *SPA* (single page application) construida sobre **AngularJs 2**. Y contiene una pequeña administración para editar parte de la información de productos y establecimientos. También se puede acceder públicamente y no requiere de autenticación.

- Acceso publico: http://quenamti.bastianc.info/
- Administracion: http://quenamti.bastianc.info/admin/categorias

Esta aplicación no tiene implementados todos los recursos expuestos por la api, por lo que es mejor probarla a través de la documentación online.

2.3

Node.js es un runtime para ejecutar JavaScript del lado del servidor, el cual está basado en el motor V8 de Gooogle, y en una arquitectura orientada a eventos. El runtime de Node.js es *single-threaded*, es decir que se ejecuta en un único hilo de proceso. Este hilo es compartido entre conexiones, permitiendo manejar un elevado número de conexiones concurrentes, ya que todas las llamadas de IO (por ejemplo acceso al disco o a la red) son asíncronas. Estas llamadas se consideran *non-blocking*, lo que significa que la ejecución de JavaScript no se detiene a esperar a que termine dicho proceso externo (leer un archivo del disco), sino que es procesado posteriormente por un *callback*.

Como se describe en su sitio oficial: "*As an asynchronous event driven JavaScript runtime, Node is designed to build scalable network applications*". Su utilidad en la construcción de aplicaciones escalables está dada por su capacidad de manejar muchas conexiones concurrentes de forma eficiente. Es similar a herramientas como *Twisted* en Python, y ha sido diseñado especialmente para el *streaming* sobre HTTP con bajas latencias. Esto lo hace especialmente apto para servir APIs, o soportar aplicaciones web *real-time*.

Existen varios frameworks basados en Node.js para desarrollar aplicaciones web, como *Express*, *Hapi*, or *Sails*. Es aquí donde se pueden hacer comparaciones con otras plataformas, como *Rails* en el caso de *Ruby*, *Django* para Python, o *Laravel* en PHP. Todas estas plataformas tienen sus propios ecosistemas, y en muchos casos una gran comunidad que aporta a estos proyectos que son en su mayoría *open source*. Pero Node.js no se queda atrás, ya que aun cuando es de las plataformas menos maduras, se ha convertido en una de las más populares.

Respecto del ejercicio anterior cabe señalar que, gracias a la gran cantidad de paquetes disponibles en el ecosistema de Node.js, ha sido relativamente simple crear el API de la tienda virtual Quena. Usando **npm** es muy simple gestionar estas dependencias, ya que nos provee de una forma rápida de instalar estos paquetes, y nos ayuda a mantener las versiones correctas para cuidar la compatibilidad con la aplicación.

En la aplicación se utiliza el framework **Express**, el cual nos permite crear un servidor web completamente funcional en no más de 50 líneas de código. También, para persistir los datos en *MongoDB* se utiliza **Mongoose**, librería que nos abstrae de los detalles de la conexión a la base de datos y nos provee de una sencilla interfaz para crear el modelo. En términos generales esta solución parece mucho más sencilla y liviana, a diferencia de otros frameworks como Rails, donde se requiere de mucha configuración para lograr el mismo resultado.

Por otro lado, es importante analizar las alternativas para desplegar aplicaciones basadas en NodeJs. Al ser una herramienta multiplataforma se puede instalar en distintos sistemas operativos, en algunos casos existen paquetes de sistemas listos para instalar NodeJs, como en Debian o Centos. Además NodeJs puede gestionar directamente un puerto TCP para recibir conexiones, lo que elimina la necesidad de un web server adicional como por ejemplo Apache.

Además de desplegar la aplicación en servidores propios existen varias alternativas, como servicios cloud. A continuación algunos ejemplos y sus características.

Amazon Web Services

Amazon Web Services (AWS) permite hacer deploy de una aplicación web en Node.js de alta disponibilidad, usando el servicio *AWS Elastic Beanstalk*. Este es un sistema de contenedores el cual hace extremadamente sencillo subir aplicaciones empaquetadas a AWS como *Platform*

as a Service (PaaS). Se tiene completo control sobre el servidor en el cual corre la aplicación, y de hecho se pueden correr múltiples aplicaciones en el servidor sin incurrir en costos adicionales.

- Ventajas: Variadas ofertas de planes, además de amplio soporte en forma de foros y buena documentación.
- Desventajas: Una curva de aprendizaje muy pronunciada cuando se trata de AWS, especialmente para aquellos que nunca lo han utilizado antes, y aún más para aquellos que nunca han sido responsables de la administración de su propio servidor antes.

Heroku

Inicialmente sólo soportando a Ruby on Rails, Heroku ha añadido soporte para una amplia gama de lenguajes y entornos incluyendo Perl, Python, PHP, Java y por supuesto Node.js. Propiedad de Salesforce.com, Heroku también ofrece una integración perfecta con una gran cantidad de servicios de terceros (como SendGrid y Redis) y permite el despliegue en varias regiones.

- Ventajas: Heroku ofrece un nivel gratuito, y es muy fácil de empezar (incluso para los desarrolladores principiantes). Hay una gran documentación de soporte disponible de forma gratuita y una extensa lista de complementos y servicios que se pueden agregar de forma instantánea.
- Desventajas: Una vez que dejas el nivel libre, Heroku empieza a ser bastante caro muy rápidamente. Además, mientras que comenzar con Heroku es rápido, las aplicaciones más grandes tienden a conducir a despliegues más lentos.

AppFog

Con una interfaz de usuario super intuitiva, una potente línea de comandos y la API REST para una implementación robusta, AppFog tiene muchísimo que hacer. La oferta PaaS permite alojar fácilmente aplicaciones Node.js en la nube, con despliegue relativamente rápido y escalabilidad automática.

- Ventajas: AppFog ofrece soporte profesional de 24 horas para todos los usuarios y tiene una sencilla utilidad de línea de comandos. Nivel gratuito para 2 GB de memoria y 100 MB de almacenamiento.
- Desventajas: Aunque el proceso de despliegue es generalmente sencillo con AppFog, el inconveniente es que no siempre es tan rápido como muchos quisieran. Si bien esto está bien si sólo está desplegando una vez al día o semana, para despliegues más regulares es posible que se desee un poco más de velocidad.

Microsoft Azure

Microsoft Azure es la plataforma basada en la nube de Microsoft que permite a los desarrolladores crear, desplegar y administrar rápidamente potentes sitios web y aplicaciones web. Se trata de una plataforma fiable con gran flexibilidad y alta disponibilidad en múltiples regiones. Con soporte para máquinas Windows y Linux, puede monitorear todas las aplicaciones Node.js alojadas en Azure en tiempo real, con soporte a escalabilidad automática.

- Ventajas: Un buen plan gratuito que le permite alojar hasta 10 aplicaciones en cada centro de datos. También es muy fácil escalar la aplicación Node.js hacia arriba o hacia abajo en Azure, o simplemente habilitar la escala automática para permitir que Azure pueda escalar de acuerdo con el tráfico.
- Desventajas: Los despliegues pueden ser lentos y no hay forma de desplegar a Azure desde un Mac.

3.1

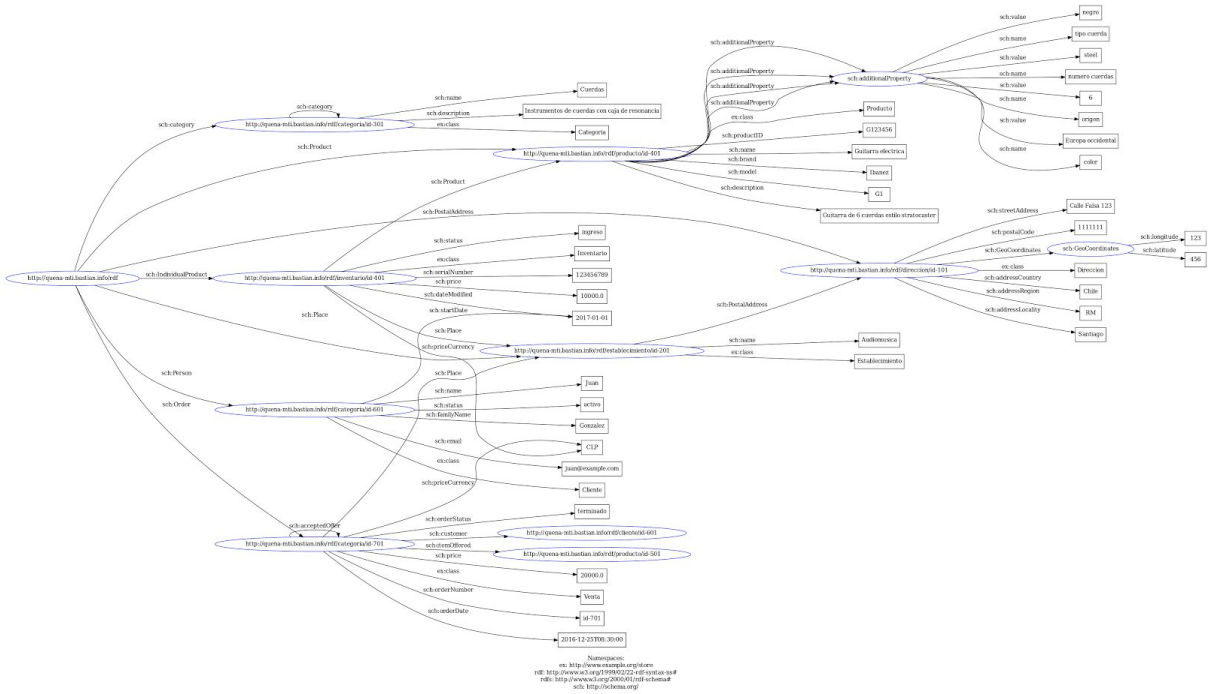
En este caso se han reutilizado los archivos XML generados en el primer ejercicio, pero siendo adaptados para cumplir con el estándar **XML/RDF v1**. El archivo se generó en base a una plantilla XLST, especialmente adaptada al archivo original.

- xml original: parte1/xml/store2.xml
- plantilla xsl: parte1/rdf/store.xls
- archivo resultante (xml/rdf): parte1/rdf/store.rdf.xml

También se ha generado una versión en formato **RDF/Turtle** del documento anterior.

- rdf/turtle: parte1/rdf/store.rdf.tt

El siguiente diagrama muestra el grafo que se desprende de la especificación en el archivo rdf/turtle.



(Se puede ver esta imagen a tamaño completo en https://github.com/bastiancy/quena-mti/blob/master/parte1/rdf/store.rdf.svg)

Los archivos antes mencionados se validaron utilizado algunos script en **Node.js**, también especialmente diseñados para este ejercicio. Por esta razón se deben instalar algunas dependencias (npm install) en la carpeta parte1/rdf/.

```
# Es válido si no retorna ningun mensaje.
node validate.js store.rdf.tt text/turtle

# Es válido si no retorna ningun mensaje.
node validate.js store.rdf.xml application/rdf+xml
```

3.2

En la aplicación Quena se ha implementado **RDFa 1.1 Lite** para agregar metadatos. Se ha utilizado *schema.org* principalmente para definir las propiedades.

Se validaron estas páginas con la herramienta online “RDFa 1.1 Distiller and Parser” de la *W3C*. En este caso se validó la página de productos del API, en el siguiente link se muestran los resultados:

https://www.w3.org/2012/pyRdfa/extract?uri=http%3A%2F%2Fapi.quenamti.bastianc.info%2Fproductos&rdfa_lite=true&vocab_expansion=false&embedded_rdf=true&validate=yes&space_preserve=true&vocab_cache_report=false&vocab_cache_bypass=false

A continuación se presenta un extracto del HTML generado y el documento RDF que se generó en formato Turtle.

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h3>Productos</h3><br>
    <ul typeof="schema:Thing">
        <li>
            <a href="/productos/590f9d24e7fe09009866f3ad">id: <span
property="schema:identifier">590f9d24e7fe09009866f3ad</span></a>
        </li>
        <li>nombre: <span property="schema:name">Quena</span></li>
        <li>descripcion: <span property="schema:description">La quena (del quechua qina) es
un instrumento de viento...</span></li>
    </ul><br>
    <ul typeof="schema:Thing">
        <li>
            <a href="/productos/590f9dafe7fe09009866f3b3">id: <span
property="schema:identifier">590f9dafe7fe09009866f3b3</span></a>
        </li>
        <li>nombre: <span property="schema:name">ZampoÃ±a</span></li>
        <li>descripcion: <span property="schema:description">La zampoÃ±a es un instrumento
de viento...</span></li>
    </ul><br>
    <ul typeof="schema:Thing">
        <li>
            <a href="/productos/590f9dc7e7fe09009866f3b9">id: <span
property="schema:identifier">590f9dc7e7fe09009866f3b9</span></a>
        </li>
        <li>nombre: <span property="schema:name">Charango</span></li>
        <li>descripcion: <span property="schema:description">El charango es un instrumento
de cuerda...</span></li>
    </ul><br>
    <ul typeof="schema:Thing">
        <li>
            <a href="/productos/590f9dd4e7fe09009866f3bf">id: <span
property="schema:identifier">590f9dd4e7fe09009866f3bf</span></a>
        </li>
        <li>nombre: <span property="schema:name">Gaita</span></li>
```

```
        <li>descripcion: <span property="schema:description">La gaita o cornamusa1 es un
instrumento de viento...</span></li>
    </ul>
</body>
</html>
```

```
@prefix schema: <http://schema.org/> .

<http://api.quenamti.bastianc.info/productos/590f9d24e7fe09009866f3ad> schema:identifier
"590f9d24e7fe09009866f3ad" .

<http://api.quenamti.bastianc.info/productos/590f9dafe7fe09009866f3b3> schema:identifier
"590f9dafe7fe09009866f3b3" .

<http://api.quenamti.bastianc.info/productos/590f9dc7e7fe09009866f3b9> schema:identifier
"590f9dc7e7fe09009866f3b9" .

<http://api.quenamti.bastianc.info/productos/590f9dd4e7fe09009866f3bf> schema:identifier
"590f9dd4e7fe09009866f3bf" .

[] a schema:Thing;
 schema:description "La zampoña es un instrumento de viento..." .

[] a schema:Thing;
 schema:description "El charango es un instrumento de cuerda....";
 schema:name "Charango" .

[] a schema:Thing;
 schema:description "La quena (del quechua qina) es un instrumento de viento....";
 schema:name "Quena" .

[] a schema:Thing;
 schema:description "La gaita o cornamusa1 es un instrumento de viento.. .
```

Puesto que la aplicación cliente fue desarrollada como una *SPA* (single page application), todo el html es generado de forma dinámica con JavaScript. Esto supone un problema para el validador el cual no puede procesar JavaScript, sino que extrae directamente el documento HTML que entrega el servidor. Por esta razón se validó contra el API, donde se devuelve por defecto documentos en HTML.

3.3

Muchos sitios se generan a partir de datos estructurados, que a menudo se almacenan en bases de datos. Cuando estos datos se formatean en HTML, resulta muy difícil recuperar los datos estructurados originales. Muchas aplicaciones, especialmente los motores de búsqueda, pueden beneficiarse enormemente del acceso directo a estos datos estructurados. El marcado en página permite a los motores de búsqueda comprender la información de las páginas web y proporcionar resultados de búsqueda más ricos para facilitar a los usuarios la búsqueda de información relevante en la Web. Para esto se usan modelos como *Microdatos* y *RDFa*.

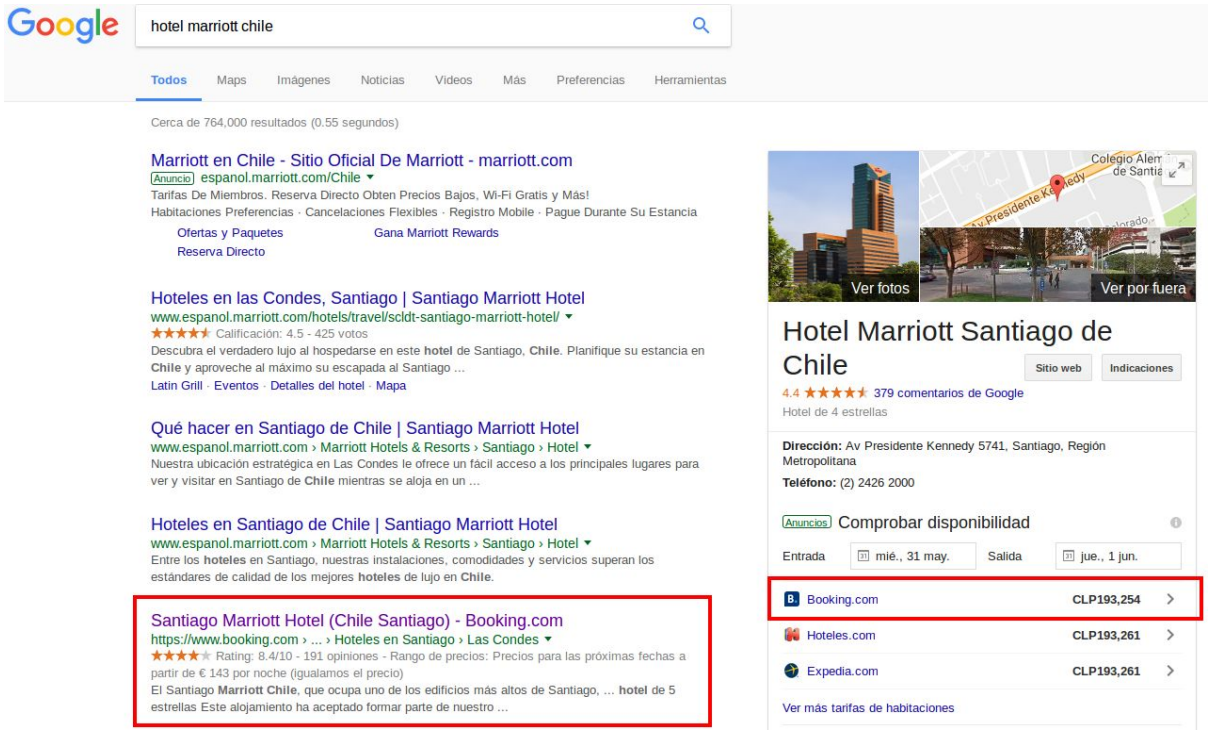
En el ejercicio anterior se utilizó **RDFa**, específicamente la recomendación *RDFa 1.1 Lite* de la *W3C*, a modo de enriquecer los documentos HTML. La ventaja de *RDFa Lite* es que se hace muy simple su implementación, puesto que es un subconjunto de *RDFa 1.1* (RDFa-core) que contiene sólo cinco atributos: vocab, typeof, property, resource y prefix. RDFa Lite es también completamente compatible con RDFa-core, donde solo es necesario agregar el resto de atributos para generar estructuras de datos más complejas.

Como se ha dicho antes, la ventaja de agregar datos estructurados a los documentos HTML es que otras herramientas (que podrían ser robots de búsqueda, clientes, complementos de navegador, CMS, etc.) puedan entender el contexto de estos datos. Un caso práctico es el de Google Search, que provee de *"Rich Snippets"* como una forma de mejorar los resultados de búsqueda. *Rich snippets* consiste en un cuadro que representa visualmente la información de los resultados de búsqueda en el sitio de Google, y que generalmente aparece más destacados que los otros resultados.

Una de las posibles desventajas es que aún cuando se pueden crear vocabularios propios, es mejor seguir convenciones como los de **schema.org**. Además el implementar estas propiedades a las etiquetas requiere de modificar la aplicación original, lo que puede significar mayores costos y esfuerzo por parte del equipo de desarrollo.

booking.com

Un caso de uso que se analizó es el sitio **http://www.booking.com**, donde se ha dado soporte a RDFa y OGP (facebook open graph). Este portal permite buscar y reservar hoteles en todo el mundo, incluso permite pagar la reserva online usando tarjeta de crédito.
Al buscar en google la frase "hotel marriott chile" aparece entre los primeros resultados el sitio de booking.com, donde se indica la valoración, cantidad de comentario, e incluso el rango de precios del hotel.



Como ejercicio se validó esta pagina particular de booking.com y se genero el siguiente documento RDF en formato Turtle (se muestra solo un extracto):

```
@prefix fb: <http://ogp.me/ns/fb#> .
@prefix ns1: <http://www.w3.org/1999/xhtml/vocab#> .
@prefix ns2: <http://ogp.me/ns#image:> .
@prefix ns3: <al:android:> .
@prefix ns4: <al:ios:> .
@prefix ns5: <http://www.w3.org/ns/rdfa#> .
@prefix ns6: <http://ogp.me/ns/fb/booking_com#location:> .
@prefix ns7: <wb:> .
@prefix ns8: <http://schema.org/> .
@prefix og2: <http://ogp.me/ns#> .

<https://www.booking.com/hotel/cl/marriott-stgo.es.html> ns3:app_name "Booking.com Hotel
Reservation"@es;
 ns3:package "com.booking"@es;
 ns3:url "booking://hotel/92228?affiliate_id=375119"@es;
 ns4:app_name "Booking.com Hotel Reservations"@es;
 ns4:app_store_id "367003839"@es;

<https://www.booking.com/hotel/cl/marriott-stgo.es.html#footertopnav> ns1:role
ns1:navigation .
```

```
<https://www.booking.com/hotel/cl/marriott-stgo.es.html#frm> ns1:role ns1:search .

<https://www.booking.com/hotel/cl/marriott-stgo.es.html#hotelpage_availform> ns1:role
ns1:form .

[] ns1:role ns1:navigation .

[] ns1:role ns1:alert .

[] a ns8:BreadcrumbList;
 ns8:itemListElement [ a ns8:ListItem;
         ns8:item <https://www.booking.com/region/cl/metropolitana.es.html?4mbcrumb=1>;
         ns8:name "Región Metropolitana"@es;
         ns8:position "3"@es ],
     [ a ns8:ListItem;
         ns8:item
<https://www.booking.com/district/cl/santiago/lascondes.es.html?4mbcrumb=1>;
         ns8:name "Las Condes"@es;
         ns8:position "5"@es ],
     [ a ns8:ListItem;
         ns8:item <https://www.booking.com/country/cl.es.html?4mbcrumb=1>;
         ns8:name "Chile"@es;
         ns8:position "2"@es ],
```

En este caso se puede apreciar que se utilizaron varios vocabularios, como *schema.org*, y *facebook open graph*, para describir distintos aspectos del sitio. Como por ejemplo, la navegación por el sitio y los detalles de este hotel, o la valoración en base a los comentarios de los clientes.

Se puede concluir entonces que agregar datos estructurados a los sitios favorece el descubrimiento de esta información en internet. Permite que otras herramientas puedan analizar estos datos, y de cara al usuario hacer más accesible dicha información. Sin embargo, también significa un esfuerzo adicional para el desarrollo de estas aplicaciones web.