TEMA 7

Almacenamiento de información basada en ficheros

Estructura de la información (I)

La información almacenada en una base de datos relacional se considera como **datos estructurados**:

- Se representan en un formato estricto
- Todas las filas de una tabla tienen el mismo formato
- Se conoce a priori el número y formato de atributos de una tabla

Hay aplicaciones en las que estas condiciones son demasiado estrictas.

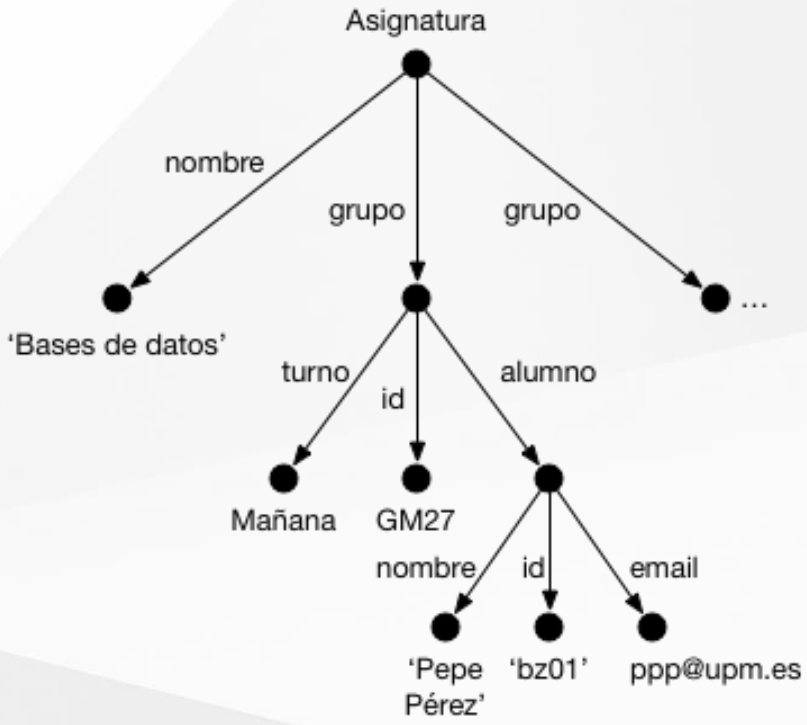
Estructura de la información (II)

Una estructura menos restrictiva serían los **datos semi-estructurados**:

- La información del esquema está mezclada con los valores de los atributos.
- Cada objeto de datos puede tener diferentes atributos que no se conocen a priori.
- Se les conoce como datos auto-descriptivos

Datos semi-estructurados

Una posible representación para los datos semi-estructurados sería un grafo dirigido:



CSV: Comma-Separated Values

Los ficheros *CSV* almacenan **información estructurada**:

- Cada fila del fichero se corresponde con un registro de información.
- Los elementos del registro se separan por un caracter delimitador, generalmente una coma.
- El orden de dichos elementos se mantiene a lo largo del fichero.

CSV: Un ejemplo

```
Login email,Identifier,First name,Last name
laura@example.com,2070,Laura,Grey
craig@example.com,4081,Craig,Johnson
mary@example.com,9346,Mary,Jenkins
jamie@example.com,5079,Jamie,Smith
```

Se observa que el nombre de los campos se encuentra en la primera fila del fichero (opcional).

También se comprueba que todos los campos de los registros siguen el mismo orden.

CSV: Ventajas

Las ventajas de usar ficheros CSV son:

1. Al ser información estructurada, su lectura y escritura es rápida y sencilla.
2. Salvo por el separador, no añade información adicional o supérflua a los datos.
3. Tiene un formato fácilmente entendible y editable.
4. Está considerado como un formato estándar (RFC4180).
5. Es fácil de generar.

CSV: Inconvenientes

1. Solo permite tipos de datos simples.
2. Hay que controlar que el separador no aparezca en los datos.
3. Hay que utilizar caracteres de escape para representar cadenas de texto que incluyan símbolos especiales.
4. Todos los registros tienen que tener el mismo número de elementos.

CSV: Casos de uso

El formato **CSV** se utiliza principalmente en las siguientes situaciones:

- Dada su estructura secuencial intrínseca, es muy común usar **CSV** para almacenar datos de procesos temporales (p.e. sensores), ya que es facil escribir al final de los ficheros.
- Análisis de datos, donde se realizan operaciones sobre el conjunto de datos completo.

XML

Extended
Markup
Language

Orígenes de XML

El origen de XML es SGML:

- Standard Generalized Markup Lenguage (SGML).
- Definido como estándar en 1986 (ISO 8879).
- Metalenguaje creado para mantener almacenes de documentación estructurada en formato electrónico.
- HTML y XML son aplicaciones de SGML.
- Muy potente y versátil pero complejo de utilizar.

Definición y antecedentes

- **XML** viene de *eXtensible Markup Language*.
- La versión 1.0 de **XML** es una recomendación del W3C (World Wide Web Consortium) desde Febrero de 1998.
- Estándar de facto para definir, crear, validar, compartir y publicar documentos con información, mediante marcas con significado.
- Puede representar datos estructurados, siendo útil como formato para el intercambio de datos entre aplicaciones.
- También se puede considerar como datos semi-estructurados.

El modelo de datos jerárquico de XML

El objeto básico de **XML** es el **documento**. Para construir un **documento XML** contamos con los siguientes conceptos:

- Elementos
- Atributos

¡OJO! El concepto de *atributo* en **XML** no se corresponde al de Bases de Datos que hemos visto hasta ahora. En **XML** los *atributos* añaden información a los *elementos*.

Elementos XML

Los **elementos** de un documento se identifican por su etiqueta de inicio y su etiqueta final.

```
<etiqueta>Elemento 1</etiqueta>
<cosa>Otro elemento</cosa>
```

El nombre de la etiqueta se incluye entre los caracteres `<` y `>`.

Además, se añade `/` delante del nombre de la **etiqueta** para identificarla como **etiqueta final**.

Elementos simples y complejos

Debido al modelo de datos de **XML** podemos distinguir dos tipos de **elementos**:

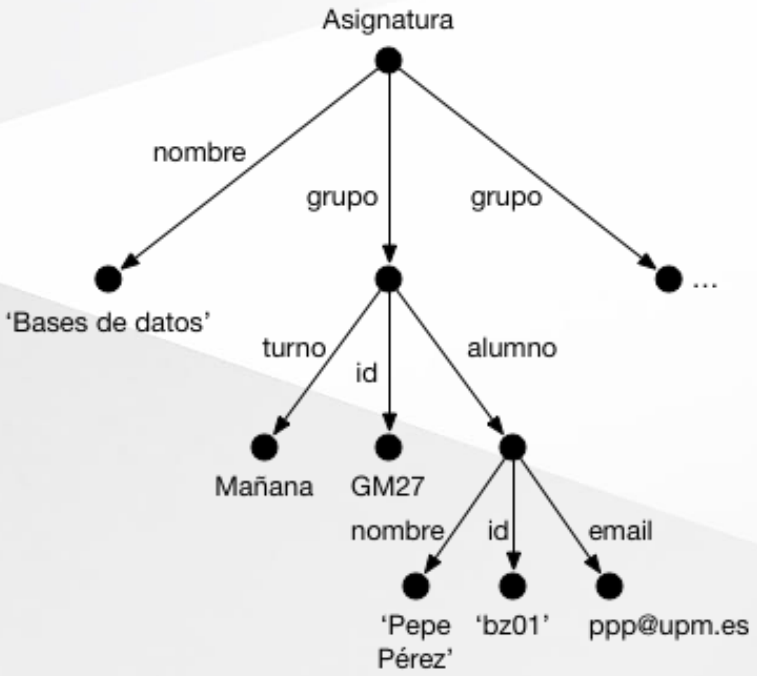
- **Elemento simple**: Contiene únicamente valores.

```
<simple>50.3</simple>
<simple>Hola Mundo</simple>
```

- **Elemento compuesto**: contiene una jerarquía de otros elementos.

```
<empleado>
    <nombre>Pepe</nombre>
    <edad>47</edad>
</empleado>
```

Los documentos XML son árboles




```
<asignatura>
    <nombre>Bases de datos</nombre>
    <grupo>
        <turno>Mañana</turno>
        <id>GM27</id>
        <alumno>
            <nombre>Pepe Pérez</nombre>
            <id>bz01</id>
            <email>ppp@upm.es</email>
        </alumno>
        ...
    </grupo>
    ...
</asignatura>
```

Atributos XML

Los **atributos** en **XML** se usan para describir propiedades y características de los elementos a los que se añaden.

```
<persona rol="jefe" departamento="ventas">
    <dni>12345678X</dni>
    <email>aaa@xyz.com</email>
    ...
</persona>
```

Los **atributos** se incluyen en la etiqueta inicial a continuación del nombre de la misma.

Es posible añadir tantos atributos como se deseen a un mismo elemento.

El formato es `nombre-atributo="valor"`.

XML vs. HTML

- Extensibilidad (etiquetas):
  - HTML: las etiquetas y atributos están prefijados.
  - XML: etiquetas y atributos **extensibles**.
- Estructura:
  - HTML se centra en presentación y es poco estructurado.
  - XML se centra en datos y es fuertemente estructurado.
- Validación:
  - HTML no comprueba tipo ni fin de las etiquetas.
  - XML requiere que el documento esté bien formado.

Puntos fuertes de XML (I)

- **Metalenguaje**: permite definir lenguajes para representar información.
- **Simplicidad**: facilidad de procesamiento por software y de entendimiento por personas.
  - Utilizable con cualquier lenguaje o alfabeto (representa el estándar unicode).
  - Sensitivo a mayúsculas y minúsculas.
  - Gramática de obligado cumplimiento.
- **Auto-descriptivo**: datos como texto, metadatos como etiquetas y atributos.

Puntos fuertes de XML (II)

- Separa:
  - Estructura (metadatos): DTD, Xml-Schema
  - Contenido (datos): documento xml
  - Apariencia (presentación): XSL, CSS
- Es un estándar para intercambio de datos en la Web y aplicaciones en general
- Poderosas técnicas para búsqueda de información: Xpath y XQuery.
- APIs en programación: DOM y SAX.

Documentos XML bien formados

Se considera que un documento **XML** está bien formado si:

1. Tiene un **único** elemento raíz.
2. Los elementos tienen una etiqueta final.
3. Las etiquetas son *case sensitive*.
4. Los elementos están anidados correctamente.
5. Los valores de los atributos están entre comillas dobles `""`.

Un documento XML bien formado

```
<?xml version="1.0" ?><!-- Nodo descriptivo -->
<w3resource>
    <design>
        <language>html</language>
        <language>xhtml</language>
        <language>css</language>
        <language>svg</language>
        <language>xml</language>
    </design>
    <programming>
        <language>php</language>
        <language>mysql</language>
    </programming>
</w3resource>
```

Documentos XML válidos

Además de estar bien formados, podemos comprobar la **validez** de un documento con respecto a un esquema determinado.

> Un documento XML se dice que es válido con respecto a un esquema si tanto su estructura como sus elementos cumplen con la especificación de dicho esquema

Para especificar equemas se usa:

- DTD (Document Type Definition)
- XML Schema

Document Type Definition (I)

Una **DTD** es un conjunto de reglas que deben cumplir tanto la estructura como los elementos de un documento **XML** para considerarse **válido**.

Una **DTD** puede incrustarse en un documento **XML** o almacenarse en un fichero distinto. En ese caso, habrá que referenciar a dicho fichero desde el documento **XML**.

Un ejemplo de regla:

```
<!ELEMENT grupo (turno id alumno+)>
```

Document Type Definition (II)

Podemos usar caracteres especiales en las reglas de una DTD:

- `+`: permite **uno o más** elementos de ese tipo dentro del elemento padre
- `*`: permite **cero o más** elementos de ese tipo dentro del elemento padre
- `?`: puede haber **entre 0 y 1** ocurrencias de elementos de ese tipo dentro del padre
- `|`: en conjunto con los paréntesis, permite establecer opcionalidad de los elementos permitidos. Equivale al operador booleano *OR*.

Document Type Definition (III)

DTD te permite el uso de palabras reservadas para definir los elementos:

- `#PCDATA` : indica que el elemento será un nodo hoja, pues requiere que tenga un valor
- `EMPTY` : indica que el elemento no tiene ningún contenido
- `ALL` : sin restricción sobre los sub-elementos de un elemento. Cualquier elemento incluso los no mencionados en la DTD pueden ser sub-elementos.

Un ejemplo de DTD

```
<!DOCTYPE banco [
    <!ELEMENT banco ((cuenta | cliente | impositor)+)>
    <!ELEMENT cuenta (número-cuenta nombre-sucursal saldo)>
    <!ELEMENT cliente (nombre-cliente calle-cliente ciudad-cliente)>
    <!ELEMENT impositor (nombre-cliente número-cuenta)>
    <!ELEMENT número-cuenta (#PCDATA)>
    <!ELEMENT nombre-sucursal (#PCDATA)>
    <!ELEMENT saldo (#PCDATA)>
    <!ELEMENT nombre-cliente (#PCDATA)>
    <!ELEMENT calle-cliente (#PCDATA)>
    <!ELEMENT ciudad-cliente (#PCDATA)>
]>
```

DTD: Atributos (I)

Para definir los atributos de cierto elemento:

```
<!ATTLIST element name type enum default mods>
```

- `element` : nombre del elemento cuyo atributo se quiere definir.
- `name` : nombre del atributo.
- `type` : tipo del atributo:
  - `CDATA` : caracteres.
  - `ID` : identificador único para el elemento (solo uno por elemento).
  - `IDREFS` : referencia al ID de otro elemento.

DTD: Atributos (II)

Para definir los atributos de cierto elemento:

```
<!ATTLIST element name type enum default mods>
```

- `enum` : (opcional) enumera los posibles valores que puede tomar el atributo (ej: `(a|b|c)`).
- `default` : (opcional) valor por defecto del atributo.
- `mods` : (opcional) modificadores que aplican al atributo:
  - `#REQUIRED` : es obligatorio definir el atributo para el elemento.
  - `#FIXED valor` : el atributo siempre será `valor` .

Un ejemplo de DTD para atributos

```
<!DOCTYPE banco-2 [
    <!ELEMENT cuenta (nombre-sucursal saldo)>
    <!ATTLIST cuenta
                número-cuenta ID #REQUIRED
                titulares IDREFS #REQUIRED>

    <!ELEMENT cliente (nombre-cliente ciudad)>
    <!ATTLIST cliente
                id-cliente ID #REQUIRED
                cuentas IDREFS #REQUIRED>
    ...
]>
```

```
<banco-2>
    <cuenta número-cuenta="C-401" titulares="C100 C102">
        <nombre-sucursal> Centro </nombre-sucursal>
        <saldo> 500 </saldo>
    </cuenta>
    <cuenta número_cuenta="C-402" titulares="C102 C101">
        <nombre-sucursal> Navacerrada </nombre-sucursal>
        <saldo> 900 </saldo>
    </cuenta>
    <cliente id-cliente="C100" cuentas="C-401">
        <nombre-cliente> Pedro </nombre-cliente>
        <calle-cliente> Arenal </calle-cliente>
        <ciudad-cliente> Toledo </ciudad-cliente>
    </cliente>
    <cliente id-cliente="C101" cuentas="C-402">
        <nombre-cliente> Ana </nombre-cliente>
        <calle-cliente> Mayor </calle-cliente>
        <ciudad-cliente> Málaga </ciudad-cliente>
    </cliente>
    ...
</banco-2>
```

Limitaciones de la DTD

DTD como mecanismo de definición de esquema tiene las siguientes limitaciones:

- No se puede declarar el **tipo de cada elemento** y de cada atributo de texto.
  - El elemento `saldo` no se puede restringir para que sea un número positivo.
- No hay forma de especificar el **tipo de elemento** al que se debería **referir** un atributo `IDREF`.
  - No se evita, por ejemplo, que el atributo `titulares` de un elemento `cuenta` se refiera a otros números de cuentas (aunque no tenga sentido).

XML Schema

- Surge como un intento para mejorar las deficiencias de las DTDs.
- Define varios tipos predefinidos: string, integer, decimal, date y boolean.
- Permite tipos definidos por el usuario.
- Se especifica en **XML**.
- El esquema se encierra en un elemento global:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    ...
</xsd:schema>
```

Un ejemplo de XML Schema

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="Empleado">
    <xsd:complexType>
        <xsd:all>
            <xsd:element name="Nombre" type="xsd:string"/>
            <xsd:element name="Sueldo" type="xsd:integer"/>
            <xsd:element name="Categoria" type="xsd:string"/>
        </xsd:all>
        <xsd:attribute name="CodE" type="xsd:string"/>
    </xsd:complexType>
</xsd:element>
</xsd:schema>
```

XPath

Lenguaje de consulta sobre documentos XML

El lenguaje XPath

Es un lenguaje de consulta sobre documentos **XML**:

- Basa su funcionamiento en expresiones de ruta.
- Estas expresiones representan una navegación por los nodos del árbol del documento XML.
- Visto de otra forma, representan la ruta a un determinado punto del documento.
- Una expresión **XPath** devuelve una colección de elementos que cumplen el patrón de la consulta (expresión).

```
/elem1/elem2/elemento
```

XPath: consultando rutas a elementos

Podemos acceder a los elementos que hay en la ruta del documento:

```
/banco/cliente/nombre-cliente
```

La consulta devolvería

```
<nombre-cliente>Pedro</nombre-cliente>
<nombre-cliente>Ana</nombre-cliente>
```

Podemos aplicar la función `text()` para quitar etiquetas y quedarnos solo con los valores:

```
/banco/cliente/nombre-cliente/text()
```

XPath: más opciones de consulta (I)

Es posible consultar los atributos de un elemento utilizando el prefijo `@`:

```
/banco/cuenta/@numero-cuenta
```

Incluso podemos indicar en la expresión algún predicado de selección:

```
/banco/cuenta[saldo > 600]/@numero-cuenta
```

que devolvería los números de cuenta con un saldo superior a 600.

XPath: más opciones de consulta (II)

También podemos usar funciones proporcionadas por XPath:

```
/banco/cuenta[count(./cliente)>2]
```

devuelve las cuentas con más de dos clientes.

Podemos buscar también por nodos enlazados por ID:

```
/banco/cuenta/id(@titulares)
```

devuelve todos los clientes referenciados desde el atributo titulares de los elementos "cuenta".

XPath: más opciones de consulta (III)

El operador `|` permite unir resultados de expresiones:

```
/banco/cuenta/id(@titulares) | /banco/préstamo/id(@prestatario)
```

Otra opción interesante es usar `//` que realiza la búsqueda a cualquier nivel del documento:

```
//curso
```

devolvería **todos** los elementos `curso` con independencia de su ubicación en el documento.

XPath: más opciones de consulta (IV)

Otros operadores de búsqueda interesantes:

Operador	Descripción
`.` / `..`	Nodo actual / Padre del nodo actual
`/centro/curso[1]`	Primer elemento `curso` hijo de `centro`
`/centro/curso[last()]`	Último elemento `curso` hijo de `centro`

XQuery

Lenguaje de consultas sobre documentos XML

XQuery: introducción

- Lenguaje de consulta para documentos XML.
- Es una recomendación del W3C.
- Integrado con XPath.
- Mantiene cierta analogía con SQL.
- La entrada y la salida de una consulta XQuery corresponde a un documento o fragmento de documento XML.

FLWOR: for, let, where, order by, return

- **FOR**: similar al `FROM` de SQL. Asigna resultados de consultas *XPath* a variables. Si pones varias variables, se realiza el producto cartesiano.
- **LET**: asigna resultados parciales a variables temporales.
- **WHERE**: aplica filtrados a las tuplas resultantes del `FOR`.
- **ORDER BY**: permite la ordenación de las salidas.
- **RETURN**: establece la forma en la que se devuelven los resultados.

XQuery: documento de ejemplo

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE libros SYSTEM "libros.dtd">
<libros>
     <libro id="1">
          <titulo>El Secreto</titulo>
          <autor>Rhonda Byrne</autor>
          <año>2007</año>
          <precio>22.50</precio>
     </libro>
     <libro id="2">
          <titulo>Indignaos</titulo>
          <autor>Stephane Hessel</autor>
          <autor>Jose Luis Sampedro</autor>
          <año>2011</año>
          <precio>15</precio>
     </libro>
</libros>
```

XQuery: consulta de ejemplo

Un ejemplo de consulta con **XQuery**:

> Obtener el titulo de los libros con valor 2 en el identificador

```
for $b in doc("libros.xml")//libro
where $b/@id = 2
return $b/titulo
```

- Indicamos el documento XML mediante: `doc("libros.xml")`.
- La doble barra `//` indica la parte del árbol xml a considerar.
- Como id es un atributo y no un elemento se antepone `@`.

Otro ejemplo de consulta con **XQuery**:

> Titulo de los libros con precio superior a 20€ ordenados por autor

```
for $x in /libros/libro
let $tit :=$x/titulo/text()
where $x/precio>20
order by $x/autor
return <titulo-libro>{$tit}</titulo-libro>
```

- El uso de llaves `{}` permite ser tratado como expresiones a evaluar. Si no aparecieran, se trataría como una cadena `$tit`.

XQuery: Uniones naturales

En **XQuery** podemos realizar uniones naturales al igual que con *SQL*:

```
for   $a in /banco/cuenta,
      $c in /banco/cliente,
      $i in /banco/impositor
let   $ccc :=$a/número-cuenta/text()
where $a/número-cuenta=$i/número-cuenta and
      $c/nombre-cliente=$i/nombre-cliente
return <cuenta-cliente>{$ccc}</cuenta-cliente>
```

que devolvería los códigos de cuenta de la unión natural entre cuentas, clientes e impositores.

XQuery: Consultas anidadas

También nos permite anidar sub-consultas entre llaves `{}`:

```
for $c in /banco/cliente
return <cliente>
        {$c/*}
        {for $i in /banco/impositor[nombre-cliente = $c/nombre-cliente],
             $a in /banco/cuenta[número-cuenta=$i/número-cuenta]
         return $a}
       </cliente>
```

ya que, como hemos dicho antes, cualquier cosa que pongamos entre llaves se va a evaluar.

XQuery: Ordenación de resultados

Para ordenar los resultados según el valor de un elemento hay que especificarlo en la parte `order by` de la consulta **XQuery**:

```
for $c in /banco/cliente
order by $c/nombre-cliente descending
return <cliente>{$c/*}</cliente>
```

Esta consulta nos devolvería los sub-elementos incluidos en cada `cliente`, pero ordenados de manera descendente según el nombre de los mismos.

XQuery: Funciones de ayuda

Tipo	Funciones
Numéricas	`floor(), ceiling(), round()`
De cadena	`concat(), string(),upper-case(), ...`
Genéricas	`distinct-values(), empty(), exists()`
De conjunto	`union (|), intersect, except`
Agregadas	`count(), sum(), avg(), min(), max()`
De contexto	`position(), text(), last()`

XQuery: Sentencias condicionales

Podemos usar sentencias condicionales en **XQuery**, muy similares a las de otros lenguajes de programación:

```
for $lib in doc("libros.xml")//libro
return
    <libro>
        {$lib/titulo}
        {$lib/autor}
        { if (count($lib/autor) > 1
          then <autor>et al.</autor>
          else ()}
    </libro>
```

¡OJO! La cláusula `else` es **obligatoria** en **XQuery**.

XQuery: Cuantificadores

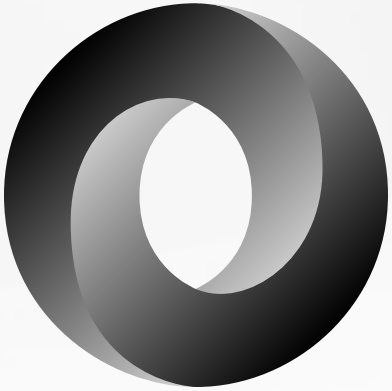
En principio las consultas devuelven aquellos nodos que cumplen las condiciones. Podemos usar cuantificadores para restringir qué nodos se devuelven:

- `some` : recupera aquellas tuplas en las que algún nodo cumpla la condición
- `every` : tuplas para las que todos sus nodos cumplen la condición

```
for $lib in //libro
where some $a in  $lib/autor satisfies ($a/first = “Jose")
return $lib/titulo
```

JSON

JavaScript Object Notation



JSON: definición

- Formato de datos semi-estructurados.
- Es una representación textual de objetos de datos.
- Permite el intercambio sencillo de información entre servicios.
- Representa objetos usando pares atributo-valor.
- Formato para SGBD no relacionales (NoSQL) como *MongoDB*.
- Su sintaxis es un subconjunto de JavaScript.

JSON: ventajas (I)

Una de las principales ventajas de **JSON** es que es auto-descriptivo y fácil de entender:

```
{
    "employees":[
        { "firstName":"John", "lastName":"Doe" },
        { "firstName":"Anna", "lastName":"Smith" },
        { "firstName":"Peter", "lastName":"Jones" }
    ]
}
```

Se pueden observar los pares de clave-valor en el documento.

JSON: ventajas (II)

Además, es más compacto que **XML** ya que elimina las etiquetas. Por ejemplo, el JSON anterior en XML sería:

```
<employees>
  <employee>
    <firstName>John</firstName> <lastName>Doe</lastName>
  </employee>
  <employee>
    <firstName>Anna</firstName> <lastName>Smith</lastName>
  </employee>
  <employee>
    <firstName>Peter</firstName> <lastName>Jones</lastName>
  </employee>
</employees>
```

JSON: estructuras básicas (I)

- Objeto
  - Colección de datos expresados como pares nombre-valor
  - Van encerrados entre llaves
  - El par nombre/valor se separa por `:`
  - Los datos o pares están separados por comas
- Lista de valores
  - Llamada array en los lenguajes de programación
  - Se encierra con corchetes `[]` y los valores se separan por comas `,`

JSON: estructuras básicas (II)

```
{
    "nombre" : "Pepe" ,
    "apellidos" : "Pérez Pérez" ,
    "estudios" : [ "Grado", "Máster" ] ,
    "edad" : 25 ,
    "teléfonos" : [
        {
            "tipo" : "casa" ,
            "numero" : "222111111"
        } ,
        {
            "tipo" : "movil" ,
            "numero" : "111111111"
        }
    ]
}
```

Documentos embebidos/integrados

- Los datos relacionados se almacenan en una sola estructura de documento.
- Recuperación y manipulación datos relacionados con una sola operación (un solo documento).

```
{
    "_id": "emple2",
    "nombre": "Pepe Pérez",
    "contacto": { "telefono": "999999999",
                  "email": "pepe.perez@json.kon"} ,
    "coche": { "matricula": "EEE 0010",
               "marca": "Toyota"}
}
```

Documentos referenciados

- Se les llama modelos normalizados.
- Los datos se almacenan con más de un documento y se referencian entre sí.
- Requieren más accesos al servidor pero permite evitar duplicación de datos.

Documento **empleado**:

```
{
    "_id": "E001",
    "nombre": "Pepe Pérez",
    "categoria": "Programador"
}
```

Documento **contacto** referencia a **empleado**:

```
{
    "_id": "CT004" ,
    "empleado_id": "E001",
    "telefono": "999999999",
    "email": "pepe.perez@json.kon"
}
```

Documento **coche** referencia a **empleado**:

```
{
    "_id": "CC407",
    "empleado_id": "E001",
    "matricula": "XYZ 0010",
    "marca": "Toyota"
}
```

Relaciones 1:1 (embebido)

```
{
    "_id": "E005",
    "nombre": "Boni Ficado",
    "coche": {
        "matricula": "JXR 5367",
        "marca": "Toyota",
        "modelo": "MA"
    }
}
```

Añadimos uno de los extremos de la relación (coche) como documento del otro extremo (empleado).

Relaciones 1:1 (normalizado)

```
{
    "_id": "E005",
    "nombre": "Boni Ficado",
}

{
    "emple_id": "E005",
    "matricula": "JXR 5367",
    "marca": "Toyota",
    "modelo": "MA"
}
```

Vinculamos un extremo (coche) con el otro (empleado).

Relaciones 1:N (embebido)

```
{
    "_id": "D001",
    "nombreDepartamento": "Ventas",
    "empleados": [
        {"nombre": "Pepe", "apellidos": "Pérez"}
        {"nombre": "Luis", "apellidos": "López"}
    ]
}
```

Añadimos una lista de objetos a la parte 1 de la relación (departamento), y añadimos los documentos de la parte N (empleados).

Relaciones 1:N (normalizado)

```
{
    "_id": "D001",
    "nombreDepartamento": "Ventas"
}
{
    "idDepartamento": "D001",
    "nombre": "Pepe",
    "apellidos": "Pérez"
}
{
    "idDepartamento": "D001",
    "nombre": "Luis",
    "apellidos": "López"
}
```

Se disocian departamento y empleados en varios documentos, y se vinculan los últimos con los primeros.

Relaciones N:M (I)

Dos documentos, cada uno de ellos incluyendo un array de referencias al otro:

```
"empleados": [
{
    "codE": "E001",
    "nombre": "Santiago",
    "departamentos": ["D001", "D002"]
}, ...
```

```
"departamentos": [
{
    "dodD": "D001",
    "descripcion": "Servicios Centrales",
    "empleados": ["E001", "E002", "E003"]
}, ...
```

Relaciones N:M (II)

Tres documentos, uno para cada Entidad relacionada y otro para reflejar referencias entre los dos anteriores:

```
"empleados": [
{
    "codE": "E001",
    "nombre": "Santiago",
}, ... ]

"departamentos": [
{
    "codD": "D001",
    "descripcion": "Servicios Centrales",
}, ... ]

"trabaja": [{"codE": "E001", "codD": "D001"},
            {"codE": "E001", "codD": "D002"}, ... ]
```

Recomendaciones de diseño (I)

Recomendable **diseño normalizado (referenciado)**:

- Relaciones complejas entre documentos de diferentes colecciones.
- Si se realizan actualizaciones frecuentemente sobre los documentos.
- Cuando la duplicación de datos no aporta ventajas suficientes que compensen el aumento de espacio en disco utilizado para ello.
- El modelo de datos se rige por una jerarquía compleja.
- Realizar varias consultas para obtener los datos no tiene un coste importante.

Recomendaciones de diseño (II)

Recomendable diseño embebido:

- Sin jerarquía compleja ni relaciones con otras colecciones de documentos.
- Se quieren obtener los datos con las mínimas peticiones al servidor
- En el modelo de datos se tienen relaciones 1:N, donde el lado N siempre será consumidos en el contexto del elemento principal.
- Optimizar la lectura de los datos, por encima de la escritura o actualización.
- Actualizaciones atómicas a nivel de documento.