UNIVERSIDAD
COMPLUTENSE
MADRID

NP2

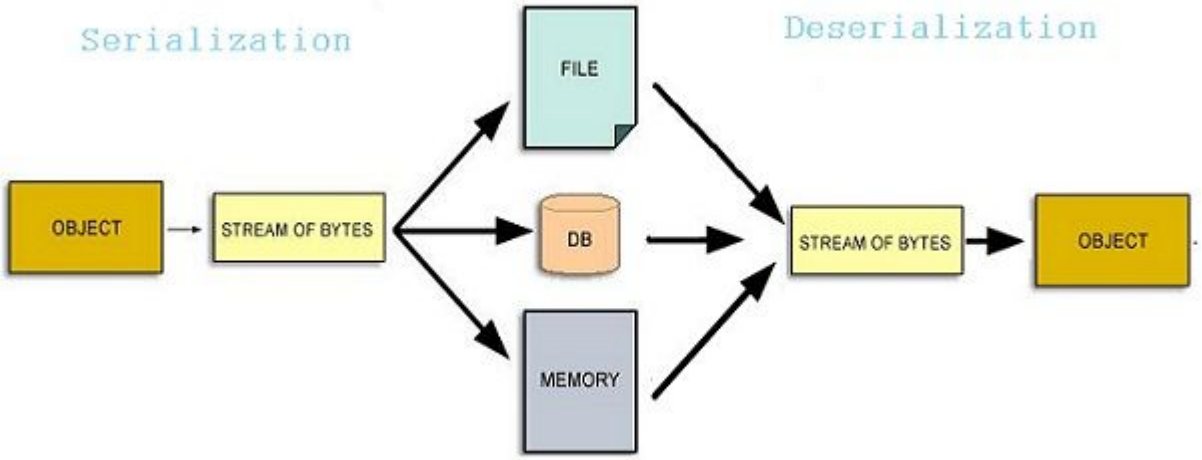
Information Representation

Facultad de Informática

Data serialization

Data Serialization

- ***Data Serialization***: converting organized objects in complex data structures for storage or transfer.
- This storage must be neutral w.r.t. the language and platform to attain interoperability.
- ***Data Deserialization***: Inverse process.

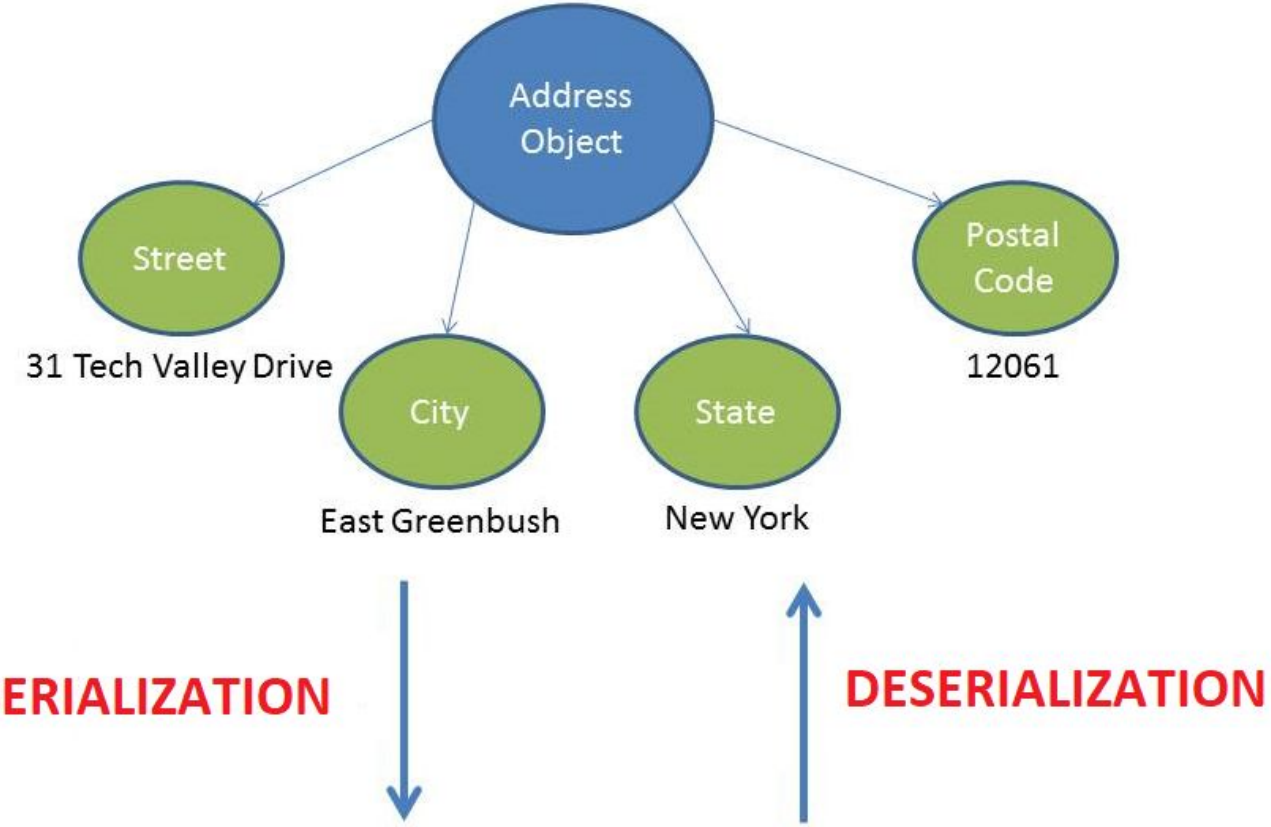


DS: Concerns

- For simple data, the process is easy.
- Problem: complex data, with references or multi-level nesting (e.g. trees).
- In case of references, double serialization for the same object must be avoided.
- The result of serialization is a byte flow that can be binary or text-based.
- For text, the encoding scheme of characters uses a standard format:
  - ISO-8859-1 (1-byte, latin alphabet characters, West Europe)
  - UTF-8 (Variable length, unicode characters, multilanguage)

DS: Example

- Serialization/Deserialization using XML



source: https://www.cisecurity.org/blog/data-deserialization/

DS: format selection

- The selection of a serialization format depends on factors such as:
  - Complexity
  - Usability (legibility)
  - Processing speed
  - Required storage space
  - Extensibility, compatibility
- Popular formats: XML, JSON, BSON, YAML, MessagePack, protobuf, CBOR, ...

DS: formats based on text

- **CSV (Comma-Separated Values)** - Table structure with delimiters. Easy to read. Can be opened on a spreadsheet.
- **XML (Extensible Markup Language)** - Nested format. Easy to read. Based on a validation schema. Used for metadata, webservices, …
- **JSON (JavaScript Object Notation)** - Compact format with nested data. Easy to read. Popular in webservices, usually via REST interfaces.
- **YAML (YAML Ain't Markup Language)** - Light format. Easy to read. Superset of JSON. Supports complex data. Usually employed in configuration files, document headers, …

DS: binary formats

- **BSON (Binary JSON)** - Created and used internally by MongoDB. Not easy to read. Similar to JSON. Includes more data types (dates, binaries, …). Used for video and multi-media documents, not usually in communications.
- **MessagePack** - Designed to be converted to/from JSON. Not easy to read. Provides better compatibility with JSON than BSON. Used for distributed applications.
- **protobuf (Protocol Buffers)** - Created by Google. Not easy to read. Allows using a schema for data. Provides data compression. Used in distributed applications.
- **CBOR** - Inspired by JSON. Not easy to read. Compact, easy to process and extensible. Supports multiple data types. Used in CoAP and COSE messages (CBOR Object Signing and Encryption, RFC-8152)

XML

https://www.ibm.com/developerworks/xml/tutorials/xmlintro/xmlintro.html

XML

- **What is XML?** XML = *eXtensible Markup Language*
  - Metalanguage that allows defining tag languages (that is, define own tags).
  - Origins: SGML (*Standard Generalized Markup Language*).
  - Developed by W3C (*World Wide Web Consortium*) to overcome limitations of HTML.
- **Pros**
  - Can be read.
  - Easy to use.
  - Versatile.
  - Portable.

XML: limitations of HTML

- XML aims at solving the problem of expressing information in a structured manner in an abstract and reusable fashion.
- HTML uses fixed marks that indicate how to visualize information (person oriented, not machine-oriented).

XML: limitations of HTML

- Example HTML

```
<p><b>Mrs. Mary McGoon</b>
<br>
1401 Main Street
<br>
Anytown, NC 34829</p>
```

- Example XML

```
<address>
  <name>
    <title>Mrs.</title>
    <first-name>
      Mary
    </first-name>
    <last-name>
      McGoon
    </last-name>
  </name>
  <street>
    1401 Main Street
  </street>
  <city>Anytown</city>
  <state>NC</state>
  <postal-code>
    34829
  </postal-code>
</address>
```

XML: marks, elements and attributes

- Mark (*tag*): text between symbols < and >
  - Marks appear in pairs (start-end)
- Element: content bound by start and end marks.
  - Elements cannot overlap.
  - Only one root element..
- Attribute: name-value pair inside the mark of an element (e.g. state="NC")
  - Example:

```
<address>
...
  <city state="NC">Anytown</city>
...
</address>
```

XML: declarations, comments

- A declaration provides basic information about the document.
  - Not required, only recommmended.
  - Example:

```
<?xml version="1.0" encoding="ISO-8859-1">
```

- A comment can include any text but --
  - Example:

```
<!-- Comment  -->
```

XML: document types

- Invalid:
  - Do not respect the syntax specifications of XML
  - Do not follow definition rules for contents (DTD or *schema*).
- Valids:
  - Respect specifications and rules
- *Well-formed*:
  - Respect specifications, rules and have a *schema*.

XML: content definitions

- Two ways to define contents:
  - **Document Type Definition (DTD)** - Defines which elements can appear in an XML document, the order in which they can appear, and other details of the document structure. Different syntax from XML.
  - **XML Schema** - Defines the same things than a DTD, but can also define datatypes and complex rules. ore expressive power.

XML: DTD

- Example:

```
<!-- address.dtd -->
<!ELEMENT address (name, street, city, state, postal-code)>
<!ELEMENT name (title? first-name, last-name)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT first-name (#PCDATA)>
<!ELEMENT last-name (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT postal-code (#PCDATA)>
```

- `#PCDATA` = *parsed character data* (no puede contener otros elementos)

XML: Schemas

- Example

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="address">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="name"/>
        <xsd:element ref="street"/>
        <xsd:element ref="city"/>
        <xsd:element ref="state"/>
        <xsd:element ref="postal-code"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
 ...

</xsd:schema>
```

EXI

Problems with XML

- Too verbose (designed for be readable)
- Example: 49 bytes of overhead to transmit a numerical field as text!!

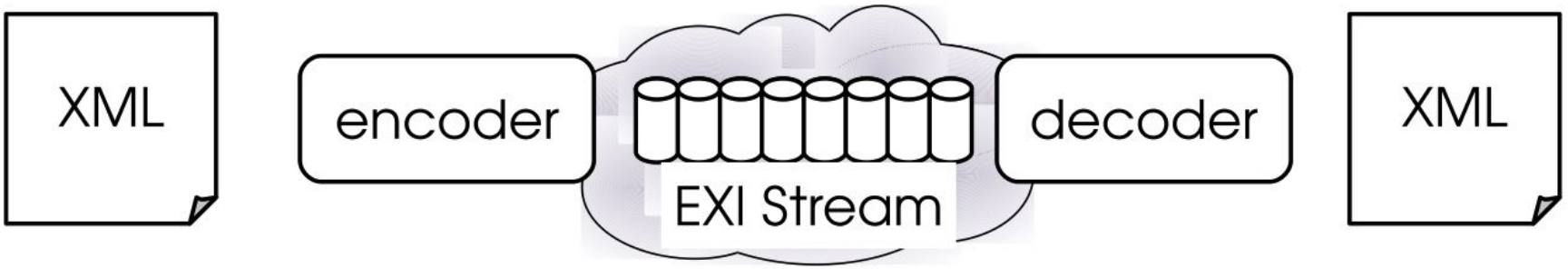
```
<SensorTemperatureValue>...</SensorTemperatureValue>
```

What is EXI?

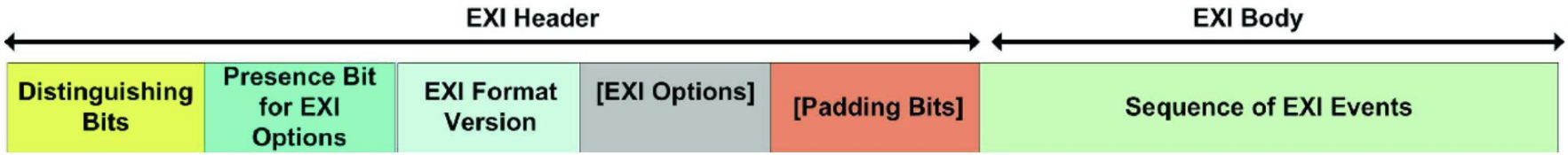
- Binary version of XML developed by W3C (2014)
- Reduces verbosity and parsing cost.
  - More compact
  - Less traffic
  - Less computational overhead
- Not a compression method, but an alternative encoding
- Suitable for small devices
  - (-) battery (-) memory (-) computation

EXI: Stream

- **EXI Stream -** Representation of an XML document via EXI



EXI: structutre of a *stream*



EXI: events

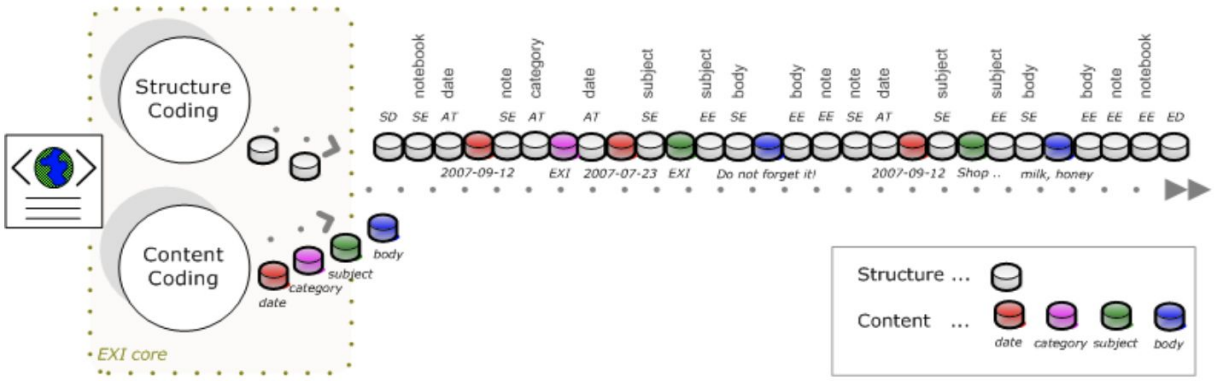
EXI Event Type	Grammar Notation	Structure	Content
Structure	Content		
[1]EXI Options such as preserve and selfContained can be used to prune events from the EXI stream to realize a more compact representation.			
Start Document	SD		
End Document	ED		
Start Element	SE (*qname*)	[*prefix*]	
SE (*uri*:*)	*local-name*, [*prefix*]		
SE (*)	*qname*, [*prefix*]		
End Element	EE		
Attribute	AT (*qname*)	[*prefix*]	*value*
AT (*uri*:*)	*local-name*, [*prefix*]		
AT (*)	*qname*, [*prefix*]		
Characters	CH		*value*

EXI: body stream

- Notebook (XML)

```
<?xml version="1.0" encoding="UTF-8"?>
<notebook date="2007-09-12">
 <note category="EXI" date="2007-07-23">
  <subject>EXI</subject>
  <body>Do not forget it!</body>
 </note>
 <note date="2007-09-12">
  <subject>Shopping List</subject>
  <body>milk, honey</body>
 </note>
</notebook>
```

- EXI Body Stream



UNIVERSIDAD COMPLUTENSE MADRID

EXI: example

- Notebook (XML): 285 bytes

```
<?xml version="1.0" encoding="UTF-8"?>
<notebook date="2007-09-12">
 <note category="EXI" date="2007-07-23">
  <subject>EXI</subject>
  <body>Do not forget it!</body>
 </note>
 <note date="2007-09-12">
  <subject>Shopping List</subject>
  <body>milk, honey</body>
 </note>
</notebook>
```

- Notebook (EXI sin *schema*): 123 bytes
- Notebook (EXI con *schema*): 60 bytes

(+detalles: https://www.w3.org/TR/exi-primer/)

JSON

What is JSON?

- JSON = JavaScript Object Notation
- Data serialization format based on text
- Uses object syntax of JavaScript
- Pros:
  - Readable
  - More compact and easier than XML
    - In Javascript, it would suffice to use eval()
  - Language independent
  - Widely expanded and used

JSON: description and types

- Essentially, a JSON document is composed by collections of key:value pairs, with the key being a string
- Datatypes:
  - **strings**: string between “
  - **numbers**: similar to majority of programming languages
  - **booleans**: true/false
  - **objects**: collection with no order of pairs key:value, separated by , and delimited by { }
  - **list**: ordered collection of values separated by , and delimited by [ ]
  - **null value**: null

JSON: example

```
{
   "first name":"John",
   "last name":"Appleseed",
   "age":30,
   "house":{
      "address":{
         "house no":"D12",
         "street":"College Ave East",
         "city":"Singapore"
      },
      "owner":"John Appleseed",
      "market price ($)":50000.00
   },
   "friends":[
      "Charles",
      "Mark",
      "Darren"
   ],
   "married":false
}
```

Herramienta comprobación formato: https://jsonformatter.curiousconcept.com/

Limitations of JSON

- No support for binary data
- Decimal numbers, necessary analysis
- Format requires:
  - Escape strings
  - Use base64 for binary data
- Not extensible (e.g. dates?)
- Interoperability problems

BSON

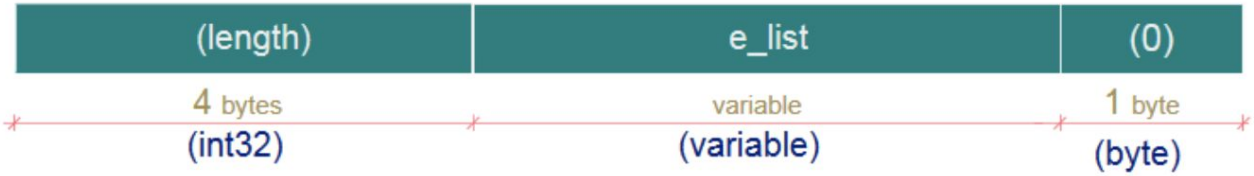
What is BSON?

- BSON: Binary JSON
- Binary representation format with origins in MongoDB
- Pros:
  - Light
  - Compact
  - Additional data types over JSON (e.g. datetime, binary data)

BSON: format and types

Format:

- length: document length
  - e_list: binary elements representation (pairs key-value) for each one, fomed by:
  - 1 byte code (value type)
  - key: character string ended by \0
  - binary value
- Basic datatypes: byte, int32, int64, uint64, double, decimal128 (little-endian)
- zero teminator:



- Tipos de datos básicos: **byte**, **int32**, **int64**, **uint64**, **double**, **decimal128** (little-endian)

BSON: codes and types

Code	Type of value	Details
`0x01`	64-bit binary floating point	-
`0x02`	UTF-8 string	the string value is comprised of: • a 32-bit integer storing the actual length of the string (say n) + 1 (for a null-terminator) • n - bytes of string • 1 byte of null-terminator
`0x03`	Embedded document	The embedded document is a BSON by itself and can be obtained via recursive approaches.
`0x04`	Array	The same as code `0x03` above except: 1. the code is `0x04`. 2. The object equivalent ({ index: element }) of the array is put as a value. For example, the object equivalent of `["A", "B", "C"]` is `{"0": "A", "1": "B", "2": "C"}`
`0x08`	Boolean	`false` boolean value is written as `0x00` and `true` is written as `0x01`.
`0x0A`	`NULL`	The value is a zero-length byte
`0x10`	32-bit integer	-

http://bsonspec.org/spec.html

UNIVERSIDAD COMPLUTENSE MADRID

BSON: example

- JSON:

```
{"abc":5}
```

- BSON:

```
\x0e\x00\x00\x00        // lentgh
\x10                    // 0x10 = integer
abc\x00                 // key
\x05\x00\x00\x00        // field value (size of value, value, null terminator)
\x00                    // 0x00 = type EOO ('end of object')
```

CBOR

What is CBOR?

- CBOR = Concise Binary Object Representation
- Binary format for data serialization inspired by JSON
- Pros:
  - Compact
  - Easy to process (small code footprint)
  - Extensible

CBOR: format and types

CBOR data	Data item 1			
Byte count	1 byte (CBOR data item header)		Variable	Variable
Structure	Major type	Additional information	Payload length (optional)	Data payload (optional)
Bit count	3 Bits	5 Bits	8 Bits × variable	8 Bits × variable

- Main type (major type, MT):
  - integers with (0) or without (1) sign
  - char strings encoded as bytes (2) or UTF-8 (3)
  - arrays (4), maps (5) (objects)
  - tagging (6)
  - simple types (7) - floating point, booleans
- Additional information (AI): immediate value or length information

CBOR: AI

- 5 bits
  - 0…23: immediate value
  - 24…27: followed by a value with 1, 2, 4 or 8 bytes
  - 28…30: reserved
  - 31: undefined length
    - Ended by 0xFF instead of a data item

- Can generate:
  - value for MT = 0, 1, (integers), 7 (simple data)
  - Length (in bytes) for MT = 2, 3
  - Count (in items) for MT = 4, 5 (arrays, maps)
  - Tag for MT=6

CBOR: MT 6 y 7

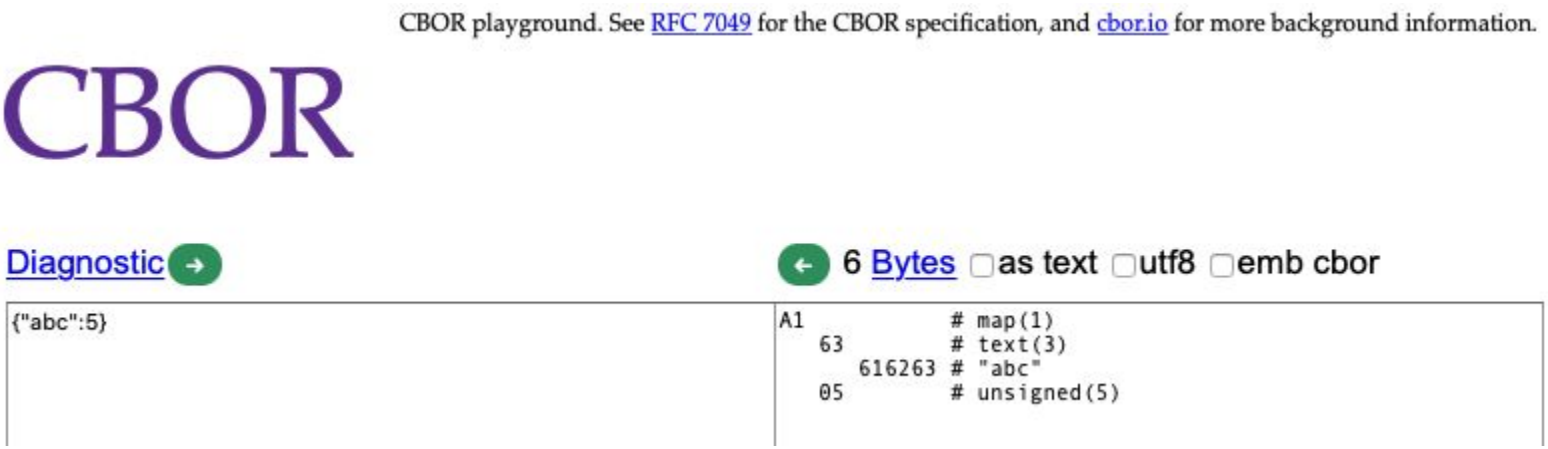
- MT=7
  - AI = 0…24
    - False, true, null, undef
    - See IANA register for more information
  - AI=25,26,27: IEEE floats, half, single, double precision
- MT=6
  - Semantic tagging (Tags)

CBOR: Tags

- A Tag contains an item
  - 0: text string that represents date/time
  - 1: UNIX time
  - 2 / 3: bignum
  - 24: CBOR item nested (byte stream)
  - 32…: URI

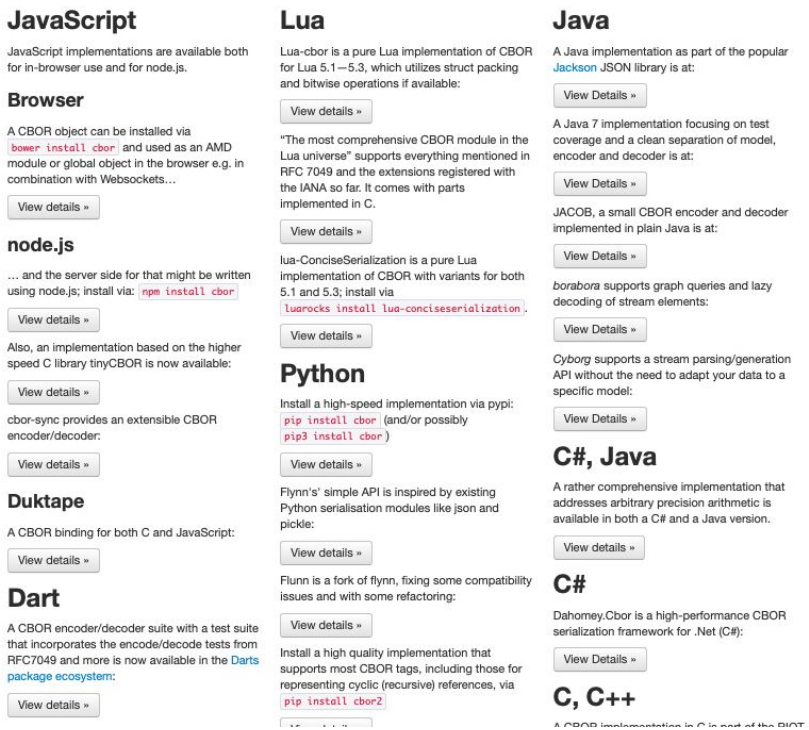
CBOR: example

- http://cbor.me
  - Converts to (~JSON) to CBOR and viceversa



CBOR: implementations

- Generating/parsing CBOR is easy
  - Smallest implementation: 822 bytes!
- Many implementations (>25)
  - https://cbor.io/impls.html



JavaScript

JavaScript implementations are available both for in-browser use and for node.js.

Browser

A CBOR object can be installed via `bower install cbor` and used as an AMD module or global object in the browser e.g. in combination with Websockets…

View details »

node.js

… and the server side for that might be written using node.js; install via: `npm install cbor`

View details »

Also, an implementation based on the higher speed C library tinyCBOR is now available:

View details »

cbor-sync provides an extensible CBOR encoder/decoder:

View details »

Duktape

A CBOR binding for both C and JavaScript:

View details »

Dart

A CBOR encoder/decoder suite with a test suite that incorporates the encode/decode tests from RFC7049 and more is now available in the Darts package ecosystem:

View details »

Lua

Lua-cbor is a pure Lua implementation of CBOR for Lua 5.1—5.3, which utilizes struct packing and bitwise operations if available:

View details »

"The most comprehensive CBOR module in the Lua universe" supports everything mentioned in RFC 7049 and the extensions registered with the IANA so far. It comes with parts implemented in C.

View details »

lua-ConciseSerialization is a pure Lua implementation of CBOR with variants for both 5.1 and 5.3; install via `luarocks install lua-conciseserialization`.

View details »

Python

Install a high-speed implementation via pypi: `pip install cbor` (and/or possibly `pip3 install cbor`)

View details »

Flynn's' simple API is inspired by existing Python serialisation modules like json and pickle:

View details »

Flunn is a fork of flynn, fixing some compatibility issues and with some refactoring:

View details »

Install a high quality implementation that supports most CBOR tags, including those for representing cyclic (recursive) references, via `pip install cbor2`

Java

A Java implementation as part of the popular Jackson JSON library is at:

View Details »

A Java 7 implementation focusing on test coverage and a clean separation of model, encoder and decoder is at:

View Details »

JACOB, a small CBOR encoder and decoder implemented in plain Java is at:

View Details »

borabora supports graph queries and lazy decoding of stream elements:

View Details »

Cyborg supports a stream parsing/generation API without the need to adapt your data to a specific model:

View Details »

C#, Java

A rather comprehensive implementation that addresses arbitrary precision arithmetic is available in both a C# and a Java version.

View details »

C#

Dahomey.Cbor is a high-performance CBOR serialization framework for .Net (C#):

View Details »

C, C++

References

- **Data serialization formats**
  - https://en.wikipedia.org/wiki/Comparison_of_data-serialization_formats
  - https://docs.python-guide.org/scenarios/serialization/
- **XML**
  - https://tools.ietf.org/html/rfc3470
  - https://www.ibm.com/developerworks/xml/tutorials/xmlintro/xmlintro.html
- **EXI**
  - https://www.w3.org/TR/exi-primer/
  - https://exificient.github.io
- **JSON**
  - https://json.org/
  - https://tools.ietf.org/html/rfc8259

References

- **BSON**
  - http://bsonspec.org/
- **CBOR**
  - https://cbor.io
  - https://en.wikipedia.org/wiki/CBOR
  - https://tools.ietf.org/html/rfc7049