

JavaScript Object Notation dan XML

Outline

Sekilas Data Serialization

- XML vs JSON vs AVRO (data serialization)
- Kelebihan dan kekurangan
- Komponen

Basic XML - JSON

- Data type
- Array
- Object
- tools Altova atau lainnya

XML - JSON Schema

- Use case SPT PPN

* **Induk**

* **Lampiran AB**

* **A1**

* **A2**

* **B1**

* **B2**

* **B3**

JSON Path

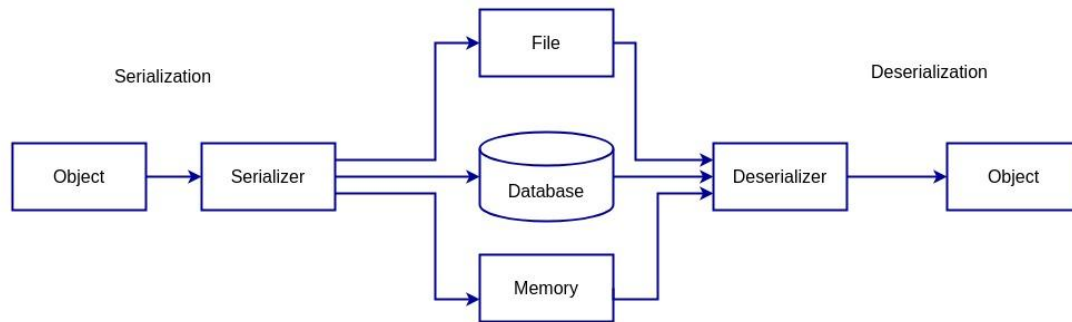


01 | Introduction

Data Serialization

Mengubah objek atau struktur data menjadi format yang nantinya dapat disimpan atau dikirim dan direkonstruksi kembali dengan mudah.

Proses sebaliknya disebut deserialisasi, yaitu merekonstruksi objek atau struktur data dari sebuah file/byte stream.



Data Serialization Format

1. Text based Format (teks yang dapat dibaca manusia) : XML, JSON, YAML, dll.

Kelebihan teks format di antaranya adalah : dapat dibaca oleh manusia, sehingga lebih mudah difahami dan didebug

2. Binary Format : ProtoBuf, Avro, MessagePack, dll.

Kelebihan binary format di antaranya adalah : lebih hemat space, dukungan definisi dan evolusi skema yang lebih kaya

JSON vs XML vs Avro vs ProtoBuf

	JSON	XML
Format	JSON menggunakan struktur map dengan pasangan key-value.	XML menyimpan data dalam struktur tree dengan <i>namespace</i> untuk kategori data yang berbeda.
Syntax	Sintaksis JSON lebih ringkas serta mudah dibaca maupun ditulis.	Sintaksis XML mengganti beberapa karakter untuk referensi entitas sehingga membuatnya menjadi lebih panjang.
Skema	sederhana dan lebih fleksibel.	kompleks dan kurang fleksibel.
Tipe data	Mendukung tipe data : null, angka, objek, <i>string</i> , <i>array</i> , dan Boolean.	XML mendukung semua tipe data JSON dan tipe-tipe tambahan, seperti binary, date, image, dan <i>namespace</i> .
Kemudahan penggunaan	Ukuran <i>file</i> yang lebih kecil dan transmisi data yang lebih cepat.	Struktur tag XML lebih kompleks untuk ditulis dan dibaca sehingga menghasilkan <i>file</i> yang lebih besar.
Keamanan	Lebih aman dibandingkan dengan XML.	Anda harus mematikan DTD ketika bekerja dengan XML untuk memitigasi potensi risiko keamanan.

JSON vs XML vs Avro vs ProtoBuf

	Avro	ProtoBuf
Format	Binary, dirancang untuk kecepatan dan efisiensi. Disupport secara terbuka oleh Hadoop	Secara fisik merupakan compact binary format yang dirancang untuk efisiensi dan fleksibel. Cocok untuk microservices
Syntax	Format binary yang memerlukan definisi skema terpisah.	Menggunakan bahasa tersendiri untuk mendefinisikan skema data.
Skema	Menggunakan JSON, IDL, atau SchemaBuilder untuk mendefinisikan skema data. Skema termasuk dalam data sbg header	Menggunakan IDL untuk mendefinisikan skema. Skema disimpan dalam file terpisah, shg cukup menyulitkan proses parsing/viewing
Tipe data	Mendukung tipe data primitif dan juga tipe data complex seperti record, enum, array, map dan union	Mendukung tipe data scalar dan juga composite seperti message dan enum. Tidak mendukung tipe kompleks seperti maps dan nested object.
Kemudahan penggunaan	Didukung oleh sebagian besar ekosistem hadoop, seperti spark, hive, nifi, flink dll	Didukung oleh javascript dan tools lain yang specific menggunakan protobuf.
Keamanan	tidak menyediakan mekanisme enkripsi bawaan.	tidak menyediakan mekanisme enkripsi bawaan.
Split/concat & kompresi	Data dapat displit, digabung (concat), maupun dikompresi	Tidak dapat

JSON vs XML vs Avro vs ProtoBuf

Contoh JSON

```
{  
  "nama": "John Doe",  
  "umur": 30,  
  "alamat": {  
    "jalan": "Jalan Raya",  
    "kota": "Jakarta"  
  },  
  "hobi": ["memancing", "membaca"]  
}
```

Contoh XML

```
<Pengguna>  
  <nama>John Doe</nama>  
  <umur>30</umur>  
  <alamat>  
    <jalan>Jalan Raya</jalan>  
    <kota>Jakarta</kota>  
  </alamat>  
  <hobi>memancing</hobi>  
  <hobi>membaca</hobi>  
</Pengguna>
```


JSON vs XML vs Avro vs ProtoBuf

Contoh AVRO

```
{
  "nama": "John Doe",
  "umur": 30,
  "alamat": {
    "jalan": "Jalan Raya",
    "kota": "Jakarta"
  },
  "hobi": ["membaca", "berkebun"]
}
```

Contoh ProtoBuf

```
{
  nama: "John Doe",
  umur: 30,
  alamat: {
    jalan: "Jalan Raya",
    kota: "Jakarta"
  },
  hobi: ["membaca", "berkebun"]
}
```

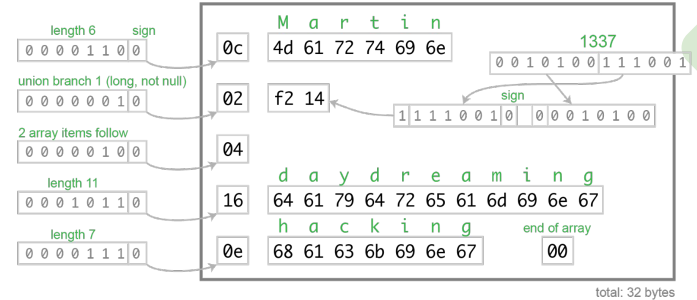
Avro vs Protobuf

```
{
  "userName": "Martin",
  "favouriteNumber": 1337,
  "interests": ["daydreaming", "hacking"]
}
```

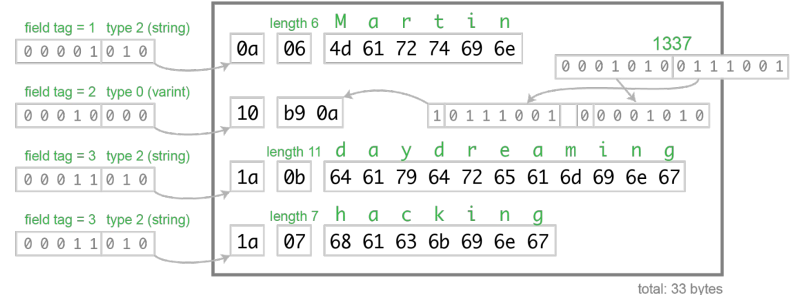
```
{
  "type": "record",
  "name": "Person",
  "fields": [
    {"name": "userName", "type": "string"},
    {"name": "favouriteNumber", "type": ["null", "long"]},
    {"name": "interests", "type": {"type": "array", "items": "string"}}
  ]
}
```

```
message Person {
  required string user_name      = 1;
  optional int64 favourite_number = 2;
  repeated string interests      = 3;
}
```

Avro



Protocol Buffers





02 | Sekilas XML

What is XML

- XML adalah singkatan dari eXtensible Markup Language.
- XML dirancang untuk menyimpan dan mengangkut data.
- XML dirancang agar dapat dibaca oleh manusia dan mesin.
- XML adalah bahasa markup yang mirip dengan HTML, tetapi tanpa tag yang telah ditentukan sebelumnya. Pengguna dapat menentukan tag sesuai kebutuhan.
- Ada banyak bahasa berdasarkan XML, termasuk XHTML, MathML, SVG, RSS, dan RDF.

XML Syntax

- Dokumen XML harus berisi satu elemen root yang merupakan induk dari semua elemen lainnya
- XML prolog bersifat opsional. Jika ada, maka harus muncul di awal dokumen.
- Semua Elemen XML Harus Memiliki Tag Penutup
- Tag XML peka terhadap huruf besar-kecil
- Elemen XML Harus bersarang (nested) dengan urutan yang tertib
- Nilai Atribut XML harus selalu menggunakan tanda kutip
- Ada 5 entity references pre-defined `< > & ' "`
- Dokumen XML yang sesuai aturan sintaksis disebut "Well Formed".

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

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

<!-- This is a comment -->
<food>
  <name>Berry-Berry Belgian Waffles</name>
  <price>8.95</price>
  <description>
    Waffles with fresh berries & cream
  </description>
  <calories>900</calories>
</food>
```

XML Element vs. Attribute

- Elemen XML adalah semua yang muncul mulai dari (termasuk) tag awal sampai (termasuk) tag akhir elemen.
- Sebuah elemen dapat berisi: teks, atribut, elemen lain, atau gabungan
- Atribut digunakan untuk memuat data yang terkait dengan elemen tertentu.
- Beberapa hal yang perlu diperhatikan ketika menggunakan atribut adalah:
 - atribut tidak dapat berisi banyak nilai
 - atribut tidak dapat berisi struktur pohon
 - atribut tidak mudah diperluas

```
<person>Bob</person>
```

```
<person gender='male'>Bob</person>
```

```
<person gender="female">  
  <firstname>Anna</firstname>  
  <lastname>Smith</lastname>  
</person>
```

```
<person>  
  <gender>female</gender>  
  <firstname>Anna</firstname>  
  <lastname>Smith</lastname>  
</person>
```

Document Type Definition

- A DTD defines the structure and the legal elements and attributes of an XML document.
- With a DTD, independent groups of people can agree on a standard DTD for interchanging data.
- An application can use a DTD to verify that XML data is valid.
- DTD can be defined inside or outside the XML file
- If the DTD is declared inside the XML file, it must be wrapped inside the `<!DOCTYPE>` definition

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE food [
  <!ELEMENT food (name,price,description,calories)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT price (#PCDATA)>
  <!ELEMENT description (#PCDATA)>
  <!ELEMENT calories (#PCDATA)>
]>

<food>
  <name>Berry-Berry Belgian Waffles</name>
  <price>8.95</price>
  <description>
    Waffles with fresh berries &amp; cream
  </description>
  <calories>900</calories>
</food>
```

DTD Building Block

- **Elements**
- **Attributes** : provide extra information about elements, always come in name/value pairs.
- **Entities** : define shortcuts to special characters.
- **PCDATA** : text that WILL be parsed by a parser. The text will be examined by the parser for entities and markup.
- **CDATA** : text that will NOT be parsed by a parser. Tags inside the text will NOT be treated as markup and entities will not be expanded.

```
<!ELEMENT food  
(name,price,description,calories)>  
<!ELEMENT name (#PCDATA)>  
<!ELEMENT price (#PCDATA)>  
<!ELEMENT description (#PCDATA)>  
<!ELEMENT calories (#PCDATA)>
```

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE food SYSTEM "food.dtd">  
<food>  
  <name>Berry-Berry Belgian Waffles</name>  
  <price>8.95</price>  
  <description>  
    Waffles with fresh berries & cream  
  </description>  
  <calories>900</calories>  
</food>
```


XML Schema Definition

- Skema XML menjelaskan struktur dokumen XML.
- Tujuan dari Skema XML adalah untuk mendefinisikan building block dari dokumen XML:
 - elemen dan atribut yang dapat digunakan dalam dokumen
 - jumlah dan urutan child elements
 - tipe data untuk elemen dan atribut
 - nilai default dan konstan untuk elemen dan atribut
- Skema XML menggunakan Sintaks XML
- Skema XML Komunikasi Data Aman
- Meskipun dokumen bersifat well-formed, error masih mungkin terjadi. Dengan Skema XML, sebagian besar kesalahan ini dapat diatasi oleh validator software.

XML Schema Definition

- `<schema>` adalah root XML Schema.
- Built-in data type : `xs:string`, `xs:decimal`, `xs:integer`, `xs:boolean`, `xs:date`, `xs:time`
- Jenis elemen :
 - Simple element : hanya mengandung teks
 - Complex type : dapat berupa empty, atau mengandung teks maupun elemen lain.
- Untuk mengatur value dari elemen atau atribut, digunakan restriction
- Untuk mengatur bagaimana elemen digunakan dalam dokumen, digunakan indicator

```
<food>
  <name>Berry-Berry Belgian Waffles</name>
  <price>8.95</price>
  <description>
    Waffles with fresh berries & cream
  </description>
</food>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="food">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="price" type="xs:decimal"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

DTD vs. XSD

DTD	XSD
Derived from SGML syntax.	Written in XML.
Doesn't support datatypes.	Supports data types for elements and attributes.
Doesn't support namespace.	Supports namespace.
Doesn't define order for child elements.	Defines order for child elements.
DTD is not extensible.	XSD is extensible.
DTD is not simple to learn.	Don't need to learn new language.
Provides less control on XML structure.	Provides more control on XML structure.



03 | Sekilas AVRO

Apache Avro

- Sangat efisien karena berformat biner. Key tidak perlu ditulis berkali-kali seperti pada JSON maupun XML
- Berformat baris, tidak optimal untuk agregasi, tetapi sesuai untuk pertukaran data
- Mendukung schema evolution
- Avro didukung oleh ekosistem Hadoop dan merupakan format terbuka

Avro Schema

- Avro Schema dibuat menggunakan format JSON
- Sebuah Avro Schema dapat berupa
 - JSON string mengandung nama type
 - JSON Object
 - JSON Array yang berisi nama type

```
"string"
```

```
{  
  "type": "record",  
  "namespace": "com.example",  
  "name": "FullName",  
  "fields": [  
    { "name": "first", "type": "string" },  
    { "name": "last", "type": "string" }  
  ]  
}
```

```
["string", "int", "boolean"]
```

Avro Schema : Primitive Data Type

- **null** : No value.
- **boolean** : A binary value.
- **int** : A 32-bit signed integer.
- **long** : A 64-bit signed integer.
- **float** : A single precision (32 bit) IEEE 754 floating-point number.
- **double** : A double precision (64-bit) IEEE 754 floating-point number.
- **bytes** : A sequence of 8-bit unsigned bytes.
- **string** : A Unicode character sequence


Avro Schema : Complex Data Type

- **record**: a type that contains named fields, each having a type, required, default, etc.
- **enum**: an enumeration of given values; other values aren't allowed
- **array**: an array of items
- **map**: a map/dictionary of key-value pairs, where the keys must be strings
- **fixed**: a fixed number of bytes
- **union**: an array containing type names; the resulting type can be any one of these types

Contoh Avro Schema

```
{
  "type": "record",
  "name": "Pengguna",
  "fields": [
    {"name": "nama", "type": "string"},
    {"name": "umur", "type": "int"},
    {
      "name": "alamat",
      "type": {
        "type": "record",
        "name": "Alamat",
        "fields": [
          {"name": "jalan", "type": "string"},
          {"name": "kota", "type": "string"}
        ]
      }
    },
    {"name": "hobi", "type": {"type": "array",
    "items": "string"}}
  ]
}
```

```
{
  "nama": "John Doe",
  "umur": 30,
  "alamat": {
    "jalan": "Jalan Raya",
    "kota": "Jakarta"
  },
  "hobi": ["membaca", "berkebun"]
}
```



04 | JSON Format

JSON (JavaScript Object Notation)

- Format pertukaran dan penyimpanan data yang ringan
- Sintaksnya sederhana, mudah dibaca dan ditulis, serta mudah diterjemahkan dan di-generate oleh komputer
- Format ini dibuat berdasarkan format object JavaScript
- Meski demikian, JSON tidak bergantung pada bahasa pemrograman apapun

Dokumen JSON

Dokumen JSON dapat berupa :

- Basic value : **string**, **number**, **boolean**, **null**
- Array, yaitu list of value yang diawali dan diakhiri dengan kurung siku []
- Object, yaitu kumpulan pasangan **key:value** yang diawali dan diakhiri kurung kurawal {}

`null`

`"To be or not to be that is the question"`

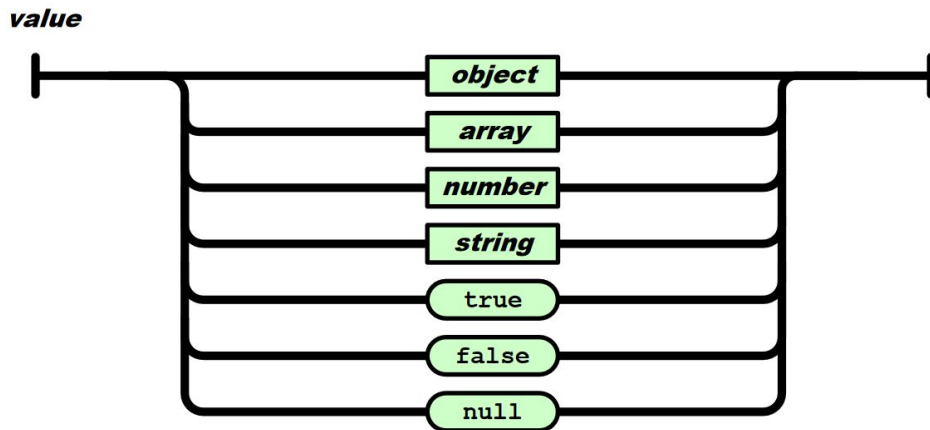
`["pepaya","mangga","pisang","jambu", true false]`

```
[  
  { "nama" : "Bob", "umur" : 35},  
  { "nama" : "Jim", "umur" : 32},  
  { "nama" : "Dana", "umur" : 29},  
]
```

```
{  
  "nama" : "Bob",  
  "umur" : 35,  
  "nilai" : [100, 200, 150]  
}
```

JSON Value

- Value dapat berupa string dalam tanda kutip ganda, atau number, true, false, null, objek, ataupun array.
- Struktur ini dapat bersarang (nested).

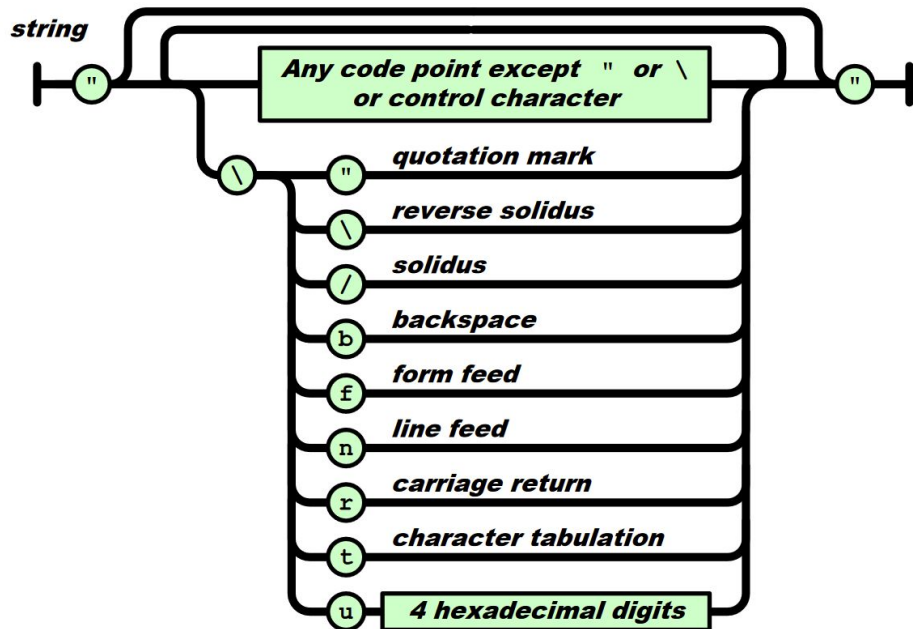


string	"Bob" "Apel" "a"
number	1 1.3 -5 1.2e20
boolean	true, false
null	null
array	[1,2,3] ["a","b","c"]
object	{"key" : "value"} {"nama" : "Bob"}

JSON String

- String adalah urutan dari 0 atau lebih karakter unicode
- String diawali dan diakhiri dengan tanda kutip ganda "

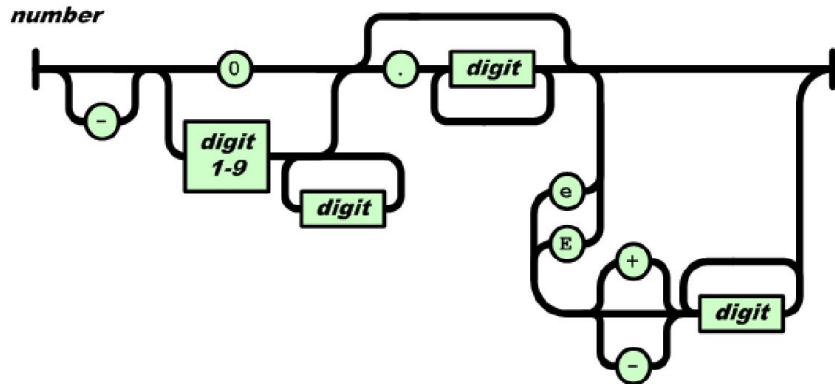
"Seruan \"merdeka!\""	Seruan "merdeka!"
"D:\\User\\Download"	D:\User\Download
"1\2 sendok makan"	1/2 sendok makan
"Satu \fDua"	Satu Dua
"Satu \nDua"	Satu Dua
"Satu \rDua"	Satu Dua
"Satu \tDua"	Satu Dua



JSON Number

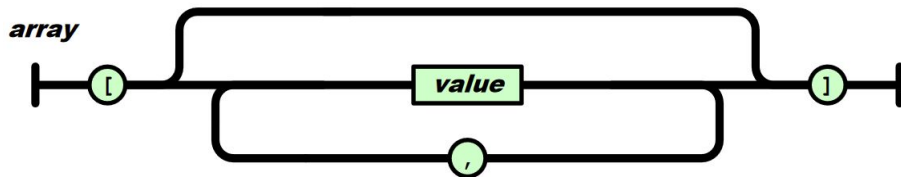
- number adalah barisan angka desimal tanpa tambahan nol di depannya. Bisa diawali tanda minus.
- Tipe data number sangat mirip dengan tipe number di C atau Java, hanya saja format oktal dan heksadesimal tidak digunakan/disupport.

1
-12345.6
12.30
1.0123e-45



JSON Array

- JSON Array adalah kumpulan value yang memiliki urutan
- Array diawali dan diakhiri dengan kurung siku
- Antara value dipisahkan dengan tanda koma



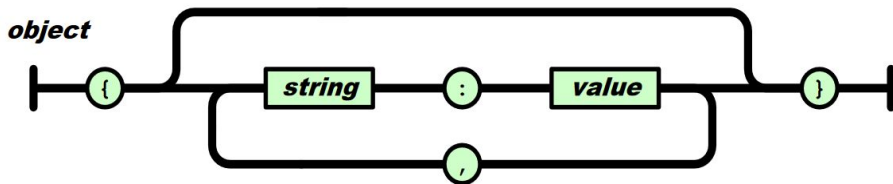
```
["baca", "jalan", "makan"]
```

```
[1, null, 3, false]
```

```
[  
  { "nama" : "Bob", "umur" : 35},  
  { "nama" : "Jim", "umur" : 32},  
  { "nama" : "Dana", "umur" : 29},  
]
```


JSON Object

- JSON Object adalah sekumpulan pasangan name/value tanpa urutan
- Object diawali dan diakhiri dengan kurung kurawal
- name diikuti tanda colon : dan masing-masing pasangan name/value dipisahkan oleh tanda koma



```
{"nama" : "Bob"}
```

```
{  
  "nama" : "Bob",  
  "umur" : 35,  
  "pekerjaan" : "karyawan"  
}
```

```
{  
  "member" : [{"nama" : "Bob"},  
               {"nama" : "Jim"}]  
}
```



Hands-On

Membuat XML & JSON Document



Deskripsi

Dalam latihan ini kita akan membuat struktur JSON :

- Basic
- Nested



05 | JSON Schema

Apa itu JSON Schema

- JSON Schema adalah kumpulan keyword dan semantik untuk melakukan anotasi dan validasi terhadap data JSON
- Aturan validasi sebuah JSON Schema dilakukan dengan mendefinisikan constraint/batasan terhadap data JSON
- Data JSON yang divalidasi menggunakan JSON Schema disebut JSON instance
- JSON Schema juga digunakan sebagai nama project <https://json-schema.org>

Mengapa Menggunakan JSON Schema

- Menjelaskan struktur, batasan, dan tipe data sebuah dokumen JSON
- Menentukan aturan yang konsisten untuk memudahkan pertukaran data antar aplikasi
- Dokumentasi yang jelas dan mudah dibaca
- JSON Schema dapat diperluas dan disesuaikan dengan kebutuhan
- Validasi data membantu untuk:
 - *Otomatisasi pengujian*: JSON Schema Validation memungkinkan pengujian otomatis, memastikan bahwa data secara konsisten mematuhi aturan dan batasan yang ditentukan.
 - *Meningkatkan kualitas data*: Dengan menerapkan aturan validasi, Skema JSON membantu memastikan kualitas data yang diterima, mencegah inkonsistensi, kesalahan, dan input berbahaya.

Contoh JSON Schema

```
{
  "nama" : "Bob",
  "umur" : 35,
  "pekerjaan" : "karyawan"
  "hobi" : ["membaca","kuliner"]
}
```

- nama : nama, tipe string, mandatory
- umur : tipe integer, mandatory
- pekerjaan : tipe data string, opsional
- hobi : tipe data array of string, opsional

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://contoh.com/anggota.schema.json",
  "title": "Anggota",
  "description": "Data anggota",
  "type": "object",
  "properties": {
    "nama": {"type": "string"},
    "umur": {"type": "integer", "minimum": 30},
    "pekerjaan": {"type": "string"},
    "hobi": {
      "type": "array",
      "items": {"type": "string"},
    }
  },
  "required": ["nama","umur"]
}
```

Konsep Dasar

- JSON Schema adalah sebuah dokumen bertipe JSON dengan keyword-keyword tertentu
- JSON Schema bisa berupa **boolean** atau **object**
- Bentuk paling sederhana :
 - **true** → selalu valid
 - **false** → selalu invalid
 - **{}** → selalu valid
- JSON Schema merupakan **constraint system** : tidak didefinisikan → dianggap valid

JSON Schema dialect

- Versi JSON Schema disebut juga *dialect*
- Dialect mewakili sekumpulan keyword dan semantik untuk mengevaluasi sebuah skema
- Keyword "**\$schema**" digunakan untuk menyatakan versi yang dipakai untuk membuat skema
- Versi JSON Schema disebut "draft" karena mengikuti aturan rilis IETF. Meskipun disebut draft, versi-versi tersebut adalah working version yang digunakan untuk production
- Beberapa versi yang banyak digunakan :
 - <http://json-schema.org/draft-04/schema#>
 - <http://json-schema.org/draft-06/schema#>
 - <http://json-schema.org/draft-07/schema#>
 - <https://json-schema.org/draft/2019-09/schema>
 - <https://json-schema.org/draft/2020-12/schema>

Keyword : "const"

- Nilai konstan
- Valid jika nilainya sama dengan value dari keyword ini
- Value dapat berupa apa saja

Jika berupa string, huruf besar/kecil harus sama

```
{  
  "const": "hello"  
}
```

"hello"	valid
"Hello"	invalid

Jika berupa array, jumlah, urutan dan tipe item harus sama

```
{  
  "const": [0,1,2]  
}
```

[0,1,2]	valid
[1,2,0]	invalid
null	invalid

Keyword : "enum"

- **enum** digunakan untuk membatasi value ke dalam sekumpulan pilihan nilai tertentu
- Value dari **enum** harus berupa array, dengan isi apa saja, tapi array tidak boleh kosong
- Sebuah instance valid terhadap keyword ini jika nilainya dapat ditemukan dalam array.

```
{  
  "enum": ["hello", "world", 1, 2, 3]  
}
```

"hello"	valid
"Hello"	invalid

```
{  
  "type": "object",  
  "properties": {  
    "name": { "type": "string" },  
    "hobby": {  
      "type": "string",  
      "enum": ["reading", "running"]  
    }  
  }  
}
```

{ "name" : "Bob", "hobby" : "running" }	valid, hobby ada di list enum
{ "name" : "Bob", "hobby" : "writing" }	Invalid, hobby tidak ada di list enum
{ "name" : "Bob" }	valid, hobby tidak mandatory

Keyword : "type"

- Menentukan tipe data
- Value-nya berupa :
 - Salah satu string berikut ini :
"null", "boolean", "object",
"array", "number", "integer",
"string"
 - Atau array berisi kombinasi
string-string di atas
- Jika value berupa array, maka instance
valid jika type-nya ditemukan dalam
array

type dengan satu nilai

```
{  
  "type": "number"  
}
```

5	valid
null	invalid

type dengan beberapa pilihan value

```
{  
  "type": ["number", "object", "null"]  
}
```

6	valid, number
"hello"	invalid, string tidak ada di list
null	valid, null
{"key1":10}	valid, object
[1, {"key1":10}, null]	invalid, array tidak ada di list

Number

- number : value bertipe integer dan float
- Beberapa validator yang dapat digunakan : `minimum`, `maximum`, `exclusiveMinimum`, `exclusiveMaximum`, `multipleOf`

multipleOf valid jika nilai instance merupakan kelipatan dari value

```
{  
  "type": "number",  
  "multipleOf": 0.2  
}
```

10	valid, kelipatan 0.2
3.7	invalid, bukan kelipatan 0.2

```
{  
  "type": "number",  
  "multipleOf": 1  
}
```

6	valid, kelipatan 1
6.2	invalid, bukan kelipatan 1

Number - minimum & exclusiveMinimum

- **minimum** dan **exclusiveMinimum** digunakan untuk membatasi nilai terkecil dari value bertipe number
- **minimum** → lebih dari atau sama dengan. Value-nya bertipe number
- **exclusiveMinimum** → lebih dari. Value-nya bertipe number atau boolean
- Jika **exclusiveMinimum** bertipe boolean, harus disertai keyword **minimum** untuk menentukan nilainya

```
{  
  "type": "number",  
  "minimum": 5  
}
```

5	valid, >= 5
4	invalid

```
{  
  "type": "number",  
  "exclusiveMinimum": 5  
}
```

5.5	valid, >5
5	invalid

```
{  
  "type": "number",  
  "exclusiveMinimum": true,  
  "minimum": 50  
}
```

55	valid, >50
50	invalid

Number - maximum & exclusiveMaximum

- **maximum** dan **exclusiveMaximum** digunakan untuk membatasi nilai terkecil dari value bertipe number
- **maximum** → kurang dari atau sama dengan. Value-nya bertipe number
- **exclusiveMaximum** → kurang dari. Value-nya bertipe number atau boolean
- Jika **exclusiveMaximum** bertipe boolean, harus disertai keyword **minimum** untuk menentukan nilainya

```
{  
  "type": "number",  
  "maximum": 5  
}
```

5	valid, <= 5
6	invalid

```
{  
  "type": "number",  
  "exclusiveMaximum": 5  
}
```

4.9	valid, <5
5	invalid

```
{  
  "type": "number",  
  "exclusiveMaximum": true,  
  "maximum": 50  
}
```

45	valid, <50
50	invalid

String - minLength, maxLength

- string/teks yang berisi karakter Unicode
- minLength dan maxLength mengatur jumlah karakter/panjang sebuah string
- pattern menggunakan regular expression untuk memvalidasi nilai string

pattern menggunakan regular expression untuk validasi pola string

```
{  
  "type": "string",  
  "pattern": "^[a-z- ]+$"  
}
```

"hello world"	valid
"Hello World"	invalid, pola tidak memperbolehkan huruf besar

minLength valid jika jumlah karakter \geq value, di mana value berupa positive integer

```
{  
  "type": "string",  
  "minLength": 10  
}
```

"Hello world!"	valid, length 12 \geq 10
"Hello!"	invalid

maxLength valid jika jumlah karakter \leq value

```
{  
  "type": "string",  
  "maxLength": 10  
}
```

"Hello!"	valid, length 6 \leq 10
"Hello World!"	invalid, length melebihi

String - `contentEncoding`, `contentType`

- Untuk non-JSON data yang disimpan dalam JSON dokumen, JSON schema menyediakan beberapa keyword untuk anotasi
- `contentEncoding` dan `contentType` diperkenalkan sejak *draft-07*
- Pilihan encoding: *7bit*, *8bit*, *binary*, *quoted-printable*, *base16*, *base32*, dan *base64*

contentEncoding menyatakan bahwa string harus diinterpretasikan sebagai data dengan encoding tertentu

```
{  
  "type": "string",  
  "contentEncoding": "base64"  
}
```

"SGVsbG8gV29ybGQh"	valid, base64 dari "Hello World!"
"Hello World!"	invalid, bukan base64

contentType menyatakan MIME type dari konten string, seperti dijelaskan dalam RFC 2046

```
{  
  "type": "string",  
  "contentType": "application/json"  
}
```

String Format - date, time, date-time

- Keyword **format** digunakan untuk melakukan validasi semantik pada data
- Format date, time, dan date-time mengikuti aturan RFC 3339 section 5.6
- Format time menggunakan UTC, dengan format +/-HH:mm. Jika tidak disebutkan, dianggap sebagai local time

Format **date** yaitu **"YYYY-MM-DD"**

```
{  
  "type": "string",  
  "format": "date"  
}
```

"2023-01-12"	valid
"12-01-2023"	invalid

Format **time** yaitu **"hh:mm:ss.sZ"**

```
{  
  "type": "string",  
  "format": "time"  
}
```

"09:05:08"	valid
"09:05:08-05:45"	valid
"09:05"	invalid

Format **date-time** **"YYYY-MM-DDThh:mm:ss.sZ"**

```
{  
  "type": "string",  
  "format": "^[a-z-]+$"  
}
```

"2023-11-11T09:05:08+01:00"	valid
"12-01-2023"	invalid
"2023-11-11T09:05:08.50"	valid

String Format - email, hostname

- Format email mengikuti *RFC 5321, section 4.1.2 - Mailbox* yaitu local-part@(domain / address-literal)
Address literal dapat berupa ipv4 atau ipv6 address
- Format hostname yaitu rangkaian karakter ASCII dari a-z, digit 0-9, dan - , dihubungkan dengan tanda titik

Format **email** yaitu **Local-part@(Domain / address-literal)**

```
{  
  "type": "string",  
  "format": "email"  
}
```

"bob@gmail.com"	valid
"bob.gmail.com"	invalid, tanpa @
"user@[IPv6:2001:db8:1ff: :a0b:dbd0]"	valid, ipv6 address

Format **hostname**

```
{  
  "type": "string",  
  "format": "hostname"  
}
```

"member.domain.com"	valid
"member@domain_com"	invalid, @ dan _

String Format - duration

- Format **duration** mengikuti ISO 8601, yaitu
`duration = "P"(dur-date / dur-time / dur-week)`
di mana
`dur-second = 1*DIGIT "S"`
`dur-minute = 1*DIGIT "M" [dur-second]`
`dur-hour = 1*DIGIT "H" [dur-minute]`
`dur-time = "T" (dur-hour / dur-minute / dur-second)`
`dur-day = 1*DIGIT "D"`
`dur-week = 1*DIGIT "W"`
`dur-month= 1*DIGIT "M" [dur-day]`
`dur-year = 1*DIGIT "Y" [dur-month]`
`dur-date = (dur-day / dur-month / dur-year)[dur-time]`

Format **duration**

```
{  
  "type": "string",  
  "format": "duration"  
}
```

"P10M5DT10H50S"	valid, 10 months, 5 days, 10 hours 50 secs
"P10M5D10H50S"	invalid, tanpa T
"P10WT10H50S"	valid, 10 weeks, 10 hours, 50 secs

Array

- Beberapa validator yang dapat digunakan : `minItems`, `maxItems`, `uniqueItems`, `contains`, `minContains`, `maxContains`, `items`, `prefixItems`, `additionalItems`, `unevaluatedItems`
- Keyword `item` dan `additionalItem` diperkenalkan sejak *draft-06*, sedangkan `minContains` dan `maxContains` sejak *2019-09*

Array - items

- Valid jika semua array item memenuhi subschema pada keyword "items"

Array dengan satu type data

```
{
  "type": "array",
  "items": {"type": "integer"}
}
```

[1,2,3,4,5]	valid, semua item bertipe integer
[1,"2",3,null]	invalid

Array dengan beberapa tipe data, jumlah & urutan bebas

```
{
  "type": "array",
  "items": {"type": ["integer","string"]}
}
```

[1,"hello",3,4,"world"]	valid
[1,"hello",3,null]	invalid

Array dengan beberapa elemen awal ditentukan tipe datanya

```
{
  "type": "array",
  "items": [{"type": "integer"},
            {"type": "string"},
            {"type": "null"}
          ]
}
```

[1,"dua",null]	valid, 3 elemen pertama sesuai
["satu",2,null]	invalid, urutan tidak sesuai
[1,"dua"]	valid, 2 elemen yang muncul sesuai
[1,"dua",null,"tiga",4,5,6]	valid, 3 elemen pertama sesuai. Selebihnya bebas

Array - items

- Valid jika semua array item memenuhi subschema pada keyword "items"

Array dengan enum

```
{
  "type": "array",
  "items": {
    "enum": ["pepaya", "mangga", "pisang", "jambu"]
  }
}
```

<code>["mangga", "mangga", "pisang", "pisang"]</code>	valid, semua elemen ada di enum
<code>["pepaya", "apel"]</code>	invalid , "apel" tidak ada di enum
<code>["jambu"]</code>	valid,
<code>[]</code>	valid

Array dengan nilai minimum

```
{
  "type": "array",
  "items": {
    "type": "integer",
    "maximum": 5,
  }
}
```

<code>[0, 1, 2, 3, 1, 2]</code>	valid, semua elemen <=
<code>["satu", 2, null]</code>	invalid , tipe bukan integer
<code>[1, -2]</code>	validi
<code>[1, 2, 3, 4, 4, 5, 6]</code>	invalid , ada elemen > 5

Array - minItem, maxItem

- minItem, maxItem digunakan untuk membatasi jumlah item

Array dengan minimum jumlah item 3

```
{  
  "type": "array",  
  "minItems": 3  
}
```

[1, "two", "three", 4, 5, 6, 7, 9, 10]	valid
[1, "two", 3]	valid
[1, "two"]	invalid

Array dengan jumlah item maksimum 3

```
{  
  "type": "array",  
  "maxItems": 3  
}
```

[]	valid
[1, "two", 3]	valid
[1, "two", 3, null]	invalid

Array dengan jumlah item minimum 3 dan maksimum 6

```
{  
  "type": "array",  
  "minItems": 3,  
  "maxItems": 6  
}
```

[1, "two", "three", 4, 5]	valid
[1, "two", 3]	valid
[1, "two"]	invalid

Array - contains, minContains, maxContains

- **"contains"** : valid jika salah satu item memenuhi subschema keyword ini
- **"minContains"** & **"maxContains"** : mengatur jumlah item yg memenuhi aturan keyword **"contains"**

Array mengandung **null** setidaknya 1

```
{  
  "type": "array",  
  "contains": { "type": "null"  
}
```

[1,2,null,4,5]	valid
[1,"hello","world",4,5]	invalid

Array mengandung setidaknya 2 elemen bertipe string

```
{  
  "type": "array",  
  "contains": { "type": "string",  
    "minContains": 2  
}
```

[1,"hello",3,4,"world"]	valid
[1,"hello",3,null]	invalid

Array tidak boleh mengandung lebih dari 2 elemen bertipe integer

```
{  
  "type": "array",  
  "contains": { "type": "integer",  
    "maxContains": 2  
}
```

[1,"dua",null]	valid
[1,2,3,4,5]	invalid

Array - additionalItems

- **additionalItems** , valid jika semua item yang tidak terkena aturan (unchecked items) memenuhi subskema pada keyword ini
- Keyword ini tidak lagi disertakan pada definisi draft 2020-12

Secara default, behavior JSON schema adalah `additionalItems : true`

```
{  
  "type": "array",  
  "items": [{  
    "type": "integer",  
    "type": "string",  
    "type": "null"  
  },  
  ],  
  "additionalItems": {  
    "type": "integer"  
  }  
}
```

[1, "dua", null]	valid
["satu", 2, null]	invalid
[1, "dua"]	valid
[1, "dua", null, 4, 5]	valid

Object

- Beberapa keyword yang paling umum digunakan untuk skema Object adalah : **properties** dan **required**
- Keyword lain yang dapat digunakan adalah : **dependencies**, **dependentSchemas**, **dependentRequired**, **minProperties**, **maxProperties**, **propertyNames**, **patternProperties**, **additionalProperties**, **unevaluatedProperties**

Skema object paling sederhana. Tidak ada constrain mengenai property-nya.

```
{  
  "type": "object"  
}
```

<code>{}</code>	valid
<code>{"nama" : "Bob"}</code>	valid
<code>{"nama" : "Bob", "alamat" : { "jalan": "jl.melati", "kota": "jakarta", "kodepos": "12120" } }</code>	valid
<code>["Bob", "jakarta", "12120"]</code>	invalid

Object - properties & required

- **properties** digunakan untuk mengatur property dari tiap pasangan key:value dalam object. Value-nya berupa object, yang disebut juga dengan subschema.
- **required** digunakan untuk menentukan key atau kolom yang mandatory. Value-nya berupa array of string.

nama dan **umur** mandatory. **alamat** opsional, tapi jika muncul harus sesuai skema yang ditentukan

```
{
  "type": "object",
  "properties": {
    "nama" : {"type": "string"},
    "umur" : {"type": "integer"},
    "alamat" : {
      "jalan" : {"type": "null"},
      "kota" : {"type": "null"},
      "kodepos" : {"type": "null"}
    }
  },
  "required": ["nama", "umur"]
}
```

<pre>{"nama" : "Bob", "umur" : 35}</pre>	valid
<pre>{ "nama" : "Bob", "alamat" : { "jalan": "jl.melati", "kota": "jakarta", "kodepos": "12120" } }</pre>	invalid

Object - minProperties & maxProperties

- Menentukan jumlah property minimum dan maksimum dari sebuah object
- Type value : integer

Jumlah property minimum 2. Type dan skemanya tidak dibatasi

```
{  
  "type": "object",  
  "minProperties": 2  
}
```

<code>{"a":1, "b":2, "c":3}</code>	valid
<code>{}</code>	invalid

Jumlah property maksimum 2. Jika keyword yang muncul terdefinisi, skemanya harus sesuai definisi. Jika keyword tidak didefinisikan, skemanya bebas

```
{  
  "type": "object",  
  "maxProperties": 2,  
  "properties": {  
    "a": {"type": "string"},  
    "b": {"type": "number"},  
    "c": {"type": "number"}  
  }  
}
```

<code>{}</code>	valid
<code>{"d":1, "e":null}</code>	valid
<code>{"a":"hello", "b":2, "c":3}</code>	invalid, jumlah property > 2
<code>{"a":1, "b":2}</code>	invalid, "a" seharusnya string

Object - propertyNamees & patternProperties

- Mengatur/memvalidasi nama-nama property dari sebuah object
- Valid jika nama properti sesuai dengan subschema ini.
- patternProperties menggunakan regular expression untuk memvalidasi pola nama properties dari sebuah object beserta aturannya

Nama property tidak boleh kurang dari 3 karakter

```
{  
  "type": "object",  
  "propertyNamees": {"minLength": 3}  
}
```

<code>{"kolom1": 1, "kolom2": 2}</code>	valid
<code>{"k1": 1, "k2": 2}</code>	invalid

Nama property yang diawali "str-" harus bertipe string, diawali "num-" harus bertipe number. Selain itu bebas

```
{  
  "type": "object",  
  "patternProperties": {  
    "^str-": {"type": "string"},  
    "^num-": {"type": "number"}  
  }  
}
```

<code>{ "str-nama": "Bob", "num-age": 35 }</code>	valid
<code>{ "str-nama", "Bob", "num-age": "35" }</code>	invalid
<code>{ "str-nama" : "Bob", "num-age": 35, "other" : null }</code>	valid

Object - additionalProperties

- Mengatur property diluar yang ditentukan dalam keyword **properties**
- Value keyword ini berupa JSON Skema, yaitu object atau boolean (true/false)
- Jika value "additionalProperties" :
 - true : selalu valid
 - false : selalu invalid (tidak boleh ada property tambahan)
- Jika value berupa object/subskema, maka additional properties harus sesuai aturan subskema tersebut
- Secara default, behavior JSON Schema adalah "additionalProperties" : true

Jika ada property selain name dan age, harus bertipe integer.

```
{  
  "type": "object",  
  "properties": {  
    "name": { "type": "string" },  
    "age": { "type": "integer" }  
  },  
  "additionalProperties": {  
    "type": "integer"  
  }  
}
```

<pre>{ "name": "Bob", "score" : 100}</pre>	valid, property selain name dan age bertipe integer
<pre>{ "rating": 1}</pre>	valid, property selain name dan age bertipe integer
<pre>{ "name": "Bob", "age": 35, "ismember" : true}</pre>	invalid , property selain name dan age bukan integer

Object : dependentRequired

- Valid jika, untuk setiap kemunculan nama/key yang menjadi value dari keyword ini, harus disertai kemunculan nama/key dari array yang bersesuaian dengannya.
- Type : object < string key : array of string >
- Keyword ini diperkenalkan sejak *draft 2019-09*

Jika terdapat property **card**, maka harus ada property **billing_address** bertipe string.

```
{
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "card": { "type": "number" },
    "billing_address": { "type": "string" }
  },
  "dependentRequired": {
    "card": ["billing_address"]
  }
}
```

<code>{}</code>	valid
<code>{"name" : "Bob"}</code>	valid
<code>{"name" : "Bob", "card" : 12345}</code>	invalid
<code>{"name" : "Bob", "card" : 12345, "billing_address": "jakarta"}</code>	valid

Combined Restrictions : *anyOf*, *allOf*, *oneOf*

Jika suatu dokumen perlu divalidasi berdasarkan beberapa subskema, kita dapat menggunakan kata kunci berikut:

- **anyOf**: valid terhadap setidaknya satu skema
- **allOf**: valid terhadap semua skema
- **oneOf**: valid terhadap salah satu skema saja

```
{
  "anyOf": [
    { "type": "string", "minLength": 5 },
    { "type": "number", "minimum": 0 }
  ]
}
```

"Hello World!"	valid
100	valid
"Hi!"	invalid

Hanya boleh memenuhi constraint salah satu subschema saja

```
{
  "oneOf": [
    { "type": "number", "minimum": 5 },
    { "type": "number", "maximum": 10 }
  ]
}
```

13	valid, memenuhi 1 kondisi
0	valid, memenuhi 1 kondisi
8	invalid, memenuhi 2 kondisi

Conditional Validation : if-then-else

- Terdiri dari 3 subskema dengan keyword if, then, dan else
- Sebuah instance valid terhadap aturan ini jika :
 - Valid terhadap skema if dan skema then
 - Tidak valid terhadap skema if, dan valid terhadap skema else

String dengan panjang minimal 6 karakter, atau nilai konstan 0

```
{  
  "if": { "type": "string" },  
  "then": { "minLength": 6 },  
  "else": { "const": 0 }  
}
```

"Hello"	invalid
0	valid

String dengan panjang minima 6 karakter, atau bertipe selain string

```
{  
  "if": { "type": "string" },  
  "then": { "minLength": 6 }  
}
```

"Hello World"	valid, string 11 char
"12"	invalid
{"length":10,"width":12 }	valid, type object

JSON Schema Core

Keywords yang diperlukan untuk memproses skema atau meta-skema, termasuk yang terbagi dalam beberapa dokumen, atau berfungsi me-reserve keyword untuk tujuan interoperabilitas.

- `$schema` : menunjukkan dialect/versi skema yang digunakan
- `$id` : identifikasi skema, mirip dengan tag `<base>` dalam HTML
- `$comment` : komentar dari penulis skema, tidak mempengaruhi perilaku skema
- `$ref` : mereferensi identifikasi skema statis (`$def`, `$anchor`, `$id`)
- `$defs` : identifikasi subschema dalam dokumen untuk diguna-ulang
- `$anchor` : string yang mengidentifikasi lokasi dalam dokumen skema. Digunakan bersama dengan `$ref`

References : \$defs & \$ref

- Untuk skema JSON yang relatif rumit, pada umumnya skema dibagi menjadi dua bagian, yaitu : skema utama dan definisi subskema.
- Definisi subskema bertipe object, dengan keyword "**\$defs**"
- Untuk mengakses definisi sebuah subskema, digunakan keyword "**\$ref**". Value-nya berformat "**#!/\$defs/nama_subskema**"

```
{
  "oneOf": [
    {"$ref": "#/$defs/books"},
    {"$ref": "#/$defs/magazines"}
  ],
  "$defs": {
    "books": {
      "type": "object",
      "properties": {
        "title": {"type": "string"},
        "pages": {"type": "number"}
      },
      "required": ["title", "pages"]
    },
    "magazines": {
      "type": "object",
      "properties": {
        "name": {"type": "string"},
        "edition": {"type": "number"}
      },
      "required": ["name", "edition"]
    }
  }
}
```

References : \$anchor

- Definisi menggunakan **\$def** hanya dapat diakses secara internal (dalam satu file)
- Ketika kita membuat skema yang perlu diakses dari file lain, kita gunakan **\$anchor** dalam definisinya
- anchor harus dimulai dengan huruf diikuti dengan sejumlah huruf, angka, - _ : atau .

Skema utama yang merujuk ke file *definitions.json*

```
{
  "anyOf": [
    { "$ref": "definitions.json#books" },
    { "$ref": "definitions.json#magazines" }
  ]
}
```

Definisi skema yang akan dirujuk, disimpan dalam file bernama *definitions.json*

```
{
  "$defs": {
    "books": {
      "$anchor": "books",
      "type": "object",
      "properties": {
        "title": { "type": "string" },
        "pages": { "type": "number" }
      },
      "required": [ "title", "pages" ]
    },
    "magazines": {
      "$anchor": "magazines",
      "type": "object",
      "properties": {
        "name": { "type": "string" },
        "edition": { "type": "number" }
      },
      "required": [ "name", "edition" ]
    }
  }
}
```



06 | JSON Path



JSON Path

- JSON Path adalah bahasa pencarian dan navigasi yang digunakan untuk menemukan elemen tertentu dalam struktur data JSON.
- JSON Path mirip dengan XPath untuk XML dan memungkinkan Anda melakukan pencarian data yang spesifik di dalam JSON.
- Notasi umum JSON Path:
 - `$` : Representasi root object.
 - `.` : Mengakses properti objek.
 - `[]` : Mengakses elemen dalam array atau melakukan filter.
 - `*` : Mengakses semua elemen atau properti.

Operasi Pada JSON Path

- Operasi JSON Path:
 - **Dot Notation (.)**: Mengakses properti dalam objek.
Contoh: `$.alamat.kota`
Hasil: Nilai dari properti *"kota"* dalam objek *"alamat"*.
 - **Bracket Notation ([])**: Mengakses elemen dalam array atau melakukan filter.
Contoh: `$.hobi[0]`
Hasil: Elemen pertama dari array *"hobi"*.
 - **Wildcard (*)**: Mengakses semua elemen atau properti.
Contoh: `$.hobi[*]`
Hasil: Semua elemen dari array *"hobi"*.
- JSON Path dapat menggunakan notasi dot atau notasi bracket untuk mengakses property
Misalnya : `$.alamat.kota` dapat ditulis `$['alamat']['kota']`

```
{  
  "nama": "John Doe",  
  "umur": 30,  
  "alamat": {  
    "jalan": "Jalan Raya",  
    "kota": "Jakarta"  
  },  
  "hobi": ["membaca", "berkebun"]  
}
```

```
$.alamat.kota  
[ "Jakarta" ]
```

```
$.hobi[0]  
[ "membaca" ]
```

```
$.hobi[*]  
[ "membaca", "berkebun" ]
```

```
$['alamat']['kota']  
[ "Jakarta" ]
```


JSONPath Syntax

Ekspresi	Keterangan	Contoh	Output
\$	Akses root object atau array	\$	Mengembalikan seluruh dokumen
.property	Memilih property tertentu dari parent object	\$.drink	[[{ "name": "Hot Tea", "price": 1}, { "name": "Ice Tea", "price": 1}]]
['property']	Memilih property tertentu parent object	\$['drink']	sda
[n]	Memilih elemen ke n dari array, indeks dimulai dari 0	\$.food[1].price	[6.95]
[index1,index2,...]	Memilih elemen-elemen dengan indeks tertentu	\$.food[0,1].price	[5.95, 6.95]
..property	Memilih property dengan nama tertentu secara rekursif	\$\$name	[5.95, 6.95, 1, 1]

```
{  
  "food": [  
    {  
      "name": "Belgian Waffles",  
      "price": "$5.95",  
      "ingredients": ["waffle", "syrup"],  
      "calories": 650  
    },  
    {  
      "name": "Homestyle Breakfast",  
      "price": "$6.95",  
      "ingredients": ["eggs", "toast"],  
      "calories": 950  
    }  
  ],  
  "drink": [  
    { "name": "Hot Tea", "price": "$1" },  
    { "name": "Ice Tea", "price": "$1" }  
  ]  
}
```

JSONPath Syntax

Ekspresi	Keterangan	Contoh	Output
*	Memilih semua elemen array atau object	\$.drink[*].*	["Hot Tea", 1, "Ice Tea", 1]
[a:b]	Memilih elemen array dari indeks a sampai sebelum b	\$.drink[0:1].*	["Hot Tea", 1]
[a:]	Memilih elemen array dari indeks a sampai akhir	\$.drink[1:].*	["Ice Tea", 1]
[:n]	Memilih n elemen pertama sebuah array	\$.drink[:1].*	["Hot Tea", 1]
[-n:]	Memilih n elemen terakhir dari array	\$.drink[-1:].*	["Ice Tea", 1]
[?(expression)]	Memilih semua elemen dalam objek atau array yang sesuai dengan ekspresi filter	\$.[?(@.price < 3)].name	["Hot Tea", "Ice Tea"]
		\$.food[?(@.price < 6)].name	["Belgian Waffles"]

```
{
  "food": [
    {
      "name": "Belgian Waffles",
      "price": "$5.95",
      "ingredients": ["waffle", "syrup"],
      "calories": 650
    },
    {
      "name": "Homestyle Breakfast",
      "price": "$6.95",
      "ingredients": ["eggs", "toast"],
      "calories": 950
    }
  ],
  "drink": [
    { "name": "Hot Tea", "price": "$1" },
    { "name": "Ice Tea", "price": "$1" }
  ]
}
```

Contoh Operasi Pada JSON Path

- Misalkan kita memiliki data JSON sebagai berikut:

```
{
  "isbn": "978-602-8519-93-9",
  "buku": {
    "judul": "Belajar JSON",
    "penulis": "John Doe",
    "bab": [
      {"bab": 1, "nama": "Pendahuluan"},
      {"bab": 3, "nama": "Pengenalan JSON"},
      {"bab": 4, "nama": "Manipulasi Data"}
    ]
  },
  "tahun": "2019",
  "penerbit": "PT Bangun Cetak Sejahtera"
}
```

- **`$.buku.judul`**: Mengakses nilai properti "judul" dalam objek "buku".
`["Belajar JSON"]`
- **`$.buku.bab[1].nama`**: Mengakses nilai properti "nama" dari setiap elemen kedua bab
`["Pengenalan JSON"]`
- **`$.buku.bab[*].nama`**: Mengakses nilai properti "nama" dari setiap elemen dalam array "bab"
`["Pendahuluan", "Pengenalan JSON", "Manipulasi Data"]`

Contoh Operasi Pada JSON Path

```
{
  "nama": "John Doe",
  "umur": 30,
  "alamat": {
    "jalan": "Jalan Raya",
    "kota": "Jakarta"
  },
  "hobi": ["membaca", "berkebun", "makan"]
}
```

- **\$.nama**: Mengakses nilai properti "nama" dari root.
["John Doe"]
- **\$.alamat.kota**: Mengakses nilai properti "kota" di dalam objek "alamat".
["Jakarta"]
- **\$.hobi[0]**: Mengakses elemen pertama dari array "hobi".
["membaca"]

- **\$.hobi[*]**: Mengakses semua elemen dari array "hobi".
["membaca", "berkebun", "makan"]
- **\$.hobi[?(@ == 'membaca')]**: Mengakses semua elemen dari array "hobi" yang nilainya sama dengan "membaca".
["membaca"]
- **\$.***: Mengakses semua properti dari root objek. Tidak rekursif. Object dikembalikan sebagai object.
[
 "John Doe",
 30,
 {
 "jalan": "Jalan Raya",
 "kota": "Jakarta"
 },
 ["membaca", "berkebun", "makan"]
]
- **\$.kota**: Mengakses nilai properti "kota" dari seluruh struktur JSON.
["Jakarta"]

Operasi Filtering Pada JSON

- Notasi bracket, selain untuk mengakses array beruka index, dapat juga digunakan untuk operasi filtering

```
$.array[?(expression)]
```

JSONPath Filter Operators

Ekspresi	Contoh	Output
==	<code>\$.?(@.name=='Hot Tea').price</code>	[1]
!=	<code>\$.?(@.name!='Hot Tea').price</code>	[5.95, 6.95, 1]
<, >, <=, >=	<code>\$.?(@.price<=6).name</code>	["Belgian Waffles", "Hot Tea", "Ice Tea"]
&&	<code>\$.?(@.price<6 && @.calories < 900).name</code>	["Belgian Waffles"]
	<code>\$.?(@.price<6 @.calories < 900).name</code>	["Belgian Waffles", "Hot Tea", "Ice Tea"]

```
{  
  "food": [  
    {  
      "name": "Belgian Waffles",  
      "price": "$5.95",  
      "ingredients": ["waffle", "syrup"],  
      "calories": 650  
    },  
    {  
      "name": "Homestyle Breakfast",  
      "price": "$6.95",  
      "ingredients": ["eggs", "toast"],  
      "calories": 950  
    }  
  ],  
  "drink": [  
    { "name": "Hot Tea", "price": "$1" },  
    { "name": "Ice Tea", "price": "$1" }  
  ]  
}
```

Contoh Operasi Filtering

```
{
  "pengguna": [
    {
      "nama": "John Doe",
      "umur": 30,
      "alamat": {
        "jalan": "Jalan Raya",
        "kota": "Jakarta"
      },
      "hobi": ["membaca", "berkebun"]
    },
    {
      "nama": "Jane Smith",
      "umur": 25,
      "alamat": {
        "jalan": "Jalan Besar",
        "kota": "Surabaya"
      },
      "hobi": ["menggambar", "menulis",
        "mendaki"]
    }
  ]
}
```

Contoh Penggunaan Operasi Filtering JSON Path:

1. Filtering Objek berdasarkan Properti:
 - **JSON Path: \$.pengguna[?(@.umur > 25)]**
 - Hasil: Menampilkan pengguna yang memiliki usia di atas 25 tahun.
2. Filtering List dalam Objek:
 - **JSON Path: \$.pengguna[?(@.hobi[0] == 'membaca')]**
 - Hasil: Menampilkan pengguna yang memiliki hobi pertama "membaca".
3. Filtering Objek berdasarkan Properti dalam List:
 - **JSON Path: \$.pengguna[?(@.alamat.kota == 'Surabaya')]**
 - Hasil: Menampilkan pengguna yang alamatnya di "Surabaya".
4. Filtering dengan Penggunaan Operasi Regex pada List:
 - **JSON Path: \$.pengguna[?(@.alamat.kota =~ /J.*i)].nama**
 - Hasil: Menampilkan nama pengguna yang memiliki alamat dengan kota didahului dengan "J" (case insensitive).
 - Tidak semua JSON Evaluator tools mensupport regex

Filtering Dengan Logical Operator

- Misalkan kita memiliki data JSON sebagai berikut:

```
{
  "buku": [
    {"judul": "Belajar JSON", "kategori": ["teknologi", "pemrograman"], "harga": 50},
    {"judul": "Pengantar JavaScript", "kategori": ["teknologi", "javascript"], "harga": 40},
    {"judul": "Seni Membaca", "kategori": ["non-fiksi", "hobi"], "harga": 30}
  ]
}
```

1. **Filtering dengan Elemen Tertentu dalam Array:**
 - **JSON Path:** \$.buku[?(@.kategori[0] == 'teknologi')]
 - **Hasil:** Menampilkan buku-buku dengan kategori pertama 'teknologi'.
2. **Filtering dengan Elemen yang Mengandung Nilai Tertentu dalam Array:**
 - **JSON Path:** \$.buku[?(@.kategori.includes('javascript'))]
 - **Hasil:** Menampilkan buku-buku yang memiliki kategori 'javascript'.
3. **Filtering dengan Operator Numerik dalam Array:**
 - **JSON Path:** \$.buku[?(@.harga > 40)]
 - **Hasil:** Menampilkan buku-buku dengan harga di atas 40.
4. **Filtering dengan Kombinasi Operasi dan Operator Logika:**
 - **JSON Path:** \$.buku[?(@.kategori.includes('teknologi') && @.harga < 50)]
 - **Hasil:** Menampilkan buku-buku dengan kategori 'teknologi' dan harga di bawah 50.



Hands-On

Query JSON



Deskripsi

Dalam latihan ini kita akan melakukan query dengan menggunakan JSON Path :

- Menampilkan semua data
- Menampilkan berdasarkan index
- Menampilkan menggunakan filter



Data JSON

```
{ "store": {
  "name": "TB Bahagia Sehat Sejahtera",
  "books": [
    { "category": "reference",
      "author": "Nigel Rees",
      "title": "Sayings of the Century",
      "price": 8.95,
      "sections": ["s1", "s2", "s3"]
    },
    { "category": "fiction",
      "author": "Evelyn Waugh",
      "title": "Sword of Honour",
      "price": 12.99,
      "sections": ["s4", "s1", "s3"]
    },
    { "category": "fiction",
      "author": "Herman Melville",
      "title": "Moby Dick",
      "isbn": "0-553-21311-3",
      "price": 8.99,
      "sections": ["s1"]
    },
  ],
}
```

```
{ "category": "fiction",
  "author": "J. R. R. Tolkien",
  "title": "The Lord of the Rings",
  "isbn": "0-395-19395-8",
  "price": 22.99,
  "sections": ["s8", "s9"]
},
{ "time": {
  "starttime": "8:00",
  "endtime": "18:00"
}
}
```



Query JSON Path

1. Tampilkan semua buku

```
$.store.books[*]
```

2. Tampilkan semua sections buku

```
$.store.books[*]
```

3. Tampilkan semua author buku

```
$.store.books[*].author
```

4. Tampilkan buku ke 3 (index ke 2)

```
$.store.books[2]
```



Query JSON Path

5. Tampilkan buku index ke 1 hingga posisi ke 3

```
$.store.books[1:3]
```

6. Tampilkan 2 buku dari belakang

```
$.store.books[-2:]
```

7. Tampilkan semua buku yang harganya kurang dari 10

```
$.store.books[?(@.price < 10)]
```

8. Tampilkan semua buku yang pengarangnya Nigel Rees

```
$.store.books[?(@.author == "Nigel Rees")]
```



Query JSON Path

9. Tampilkan semua buku yang kategorinya fiction dan harganya > 10

```
$.store.books[?(@.category=='fiction' && @.price>10 )]
```

10. Tampilkan semua buku yang memiliki isbn

```
$.store.books[?(@.isbn)]
```