

# JSON

JSON

JavaScript Object Notation

# Introduction

---

- ▶ **This Lecture Covers**

- ▶ What is JSON?
  - ▶ Basic types
  - ▶ Arrays
  - ▶ Objects
  - ▶ Nesting
  - ▶ Indentation
  - ▶ Conversion between JSON and other structures
-

# JSON

---



- ▶ **JavaScript Object Notation**
    - ▶ JavaScript is a programming language
    - ▶ JSON was originally created to hold structured data to be used in JavaScript
  - ▶ **JSON became so popular...**
    - ▶ It is used for data for all kinds of applications
    - ▶ It is the most popular way of sending data for Web APIs
-

# JSON and JavaScript

javascript	json
<pre>1 v const person = { 2   name: "John", 3   age: 30, 4   isEmployed: true, 5   skills: ["JavaScript", "Python", "SQL"] 6 };</pre>	<pre>1 v { 2   "name": "John", 3   "age": 30, 4   "isEmployed": true, 5   "skills": ["JavaScript", "Python", "SQL"] 6 }</pre>

## Key differences:

- In JavaScript, keys do not need quotes unless they contain special characters or spaces.
- In JSON, all keys must be enclosed in double quotes(“)

javascript	
<pre>1 // JSON string 2 const jsonString = '{"name": "Alice", "age": 25, "isStudent": true}'; 3 4 // Parse JSON into a JavaScript object 5 const person = JSON.parse(jsonString); 6 7 console.log(person.name); // Output: Alice 8 console.log(person.age); // Output: 25 9 console.log(person.isStudent); // Output: true</pre>	<pre>1 // JavaScript object 2 v const car = { 3   make: "Toyota", 4   model: "Corolla", 5   year: 2022, 6   features: ["GPS", "Bluetooth", "Backup Camera"] 7 }; 8 9 // Convert JavaScript object to JSON string 10 const jsonString = JSON.stringify(car); 11 12 console.log(jsonString); -----</pre>

# Basic Data Types

---

- ▶ **Strings**
    - ▶ Enclosed in single or double quotation marks
  - ▶ **Numbers**
    - ▶ Integer or decimal, positive or negative
  - ▶ **Booleans**
    - ▶ true or false
    - ▶ No quotation marks
  - ▶ **null**
    - ▶ Means “nothing”.
    - ▶ No quotation marks
-

# Arrays

---

- ▶ Arrays are lists
  - ▶ In square brackets [ ]
  - ▶ Comma-separated
  - ▶ Can mix data types
  - ▶ Examples:
    - ▶ [4, 6, 23.1, -4, 0, 56]
    - ▶ ["red", "green", "blue"]
    - ▶ [65, "toast", true, 21, null, 100]
-

# Objects

---

- ▶ Objects are JSON's dictionaries
  - ▶ They are enclosed in curly brackets { }
  - ▶ Keys and values are separated by a colon :
  - ▶ Pairs are separated by commas.
  - ▶ Keys and values can be any data type.
  - ▶ Example:
    - ▶ {"red":205, "green":123, "blue":53}
-

# Nesting

---

- ▶ Nesting involves putting arrays and objects inside each other:
  - ▶ You can put arrays inside objects, objects inside arrays, arrays inside arrays, etc.
  - ▶ Often a JSON file is one big object with lots of objects and arrays inside.
-

# Example JSON: Describing a song

---

```
{  
  "song":  
    {  
      "title": "Hey Jude",  
      "artist": "The Beatles",  
      "musicians":  
        ["John Lennon", "Paul McCartney",  
          "George Harrison", "Ringo Starr"]  
    }  
}
```

---

# Example JSON: Describing a menu

---

```
{
  "menu": [
    {
      "header": "File",
      "items": [
        {"id": "Open", "label": "Open"},
        {"id": "New", "label": "New"},
        {"id": "Close", "label": "Close"}
      ]
    },
    {
      "header": "View",
      "items": [
        {"id": "ZoomIn", "label": "Zoom In"},
        {"id": "ZoomOut", "label": "Zoom Out"},
        {"id": "OriginalView", "label": "Original View"}
      ]
    }
  ]
}
```

## File

Open

New

Close

## View

Zoom In

Zoom Out

Original View

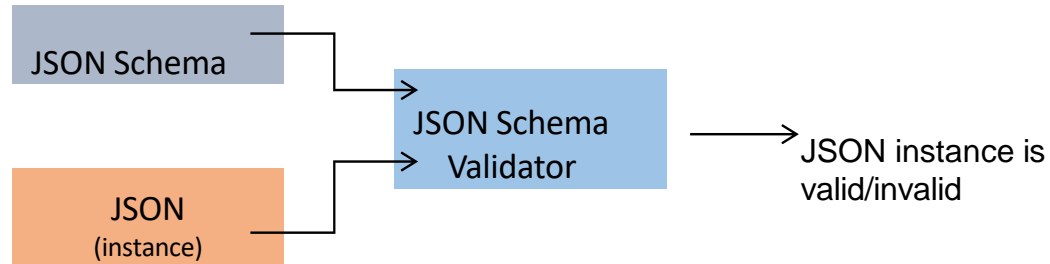
# White Space and Indentation

---

- ▶ “White space” means spaces, new lines, etc.
  - ▶ White space doesn't matter
    - ▶ Unless it's inside quotation marks
  - ▶ Good JSON formatting
    - ▶ In general, add an indent for every new level of brackets
    - ▶ End lines with commas
    - ▶ Lots of exceptions to this!
-

# Validate JSON docs against JSON Schema

---



# Example “book”

---

```
{  
  "Book":  
    {  
      "Title": "Echoes of Algorithms: Decoding the Future of AI",  
      "Authors": [ "Sarah Al-Mansouri", "Liam Nguyen" ],  
      "Date": "2023",  
      "Publisher": "Nexus AP"  
    }  
}
```

---

```
{
  "$schema": "http://json-schema.org/draft-04/schema",
  "type": "object",
  "properties": {
    "Book": {
      "type": "object",
      "properties": {
        "Title": {"type": "string"},
        "Authors": {"type": "array", "minItems": 1, "maxItems": 5, "items": {"type": "string"}},
        "Date": {"type": "string", "pattern": "^[0-9]{4}$"},
        "Publisher": {"type": "string", "enum": ["Springer", "MIT Press", "Nexus AP"]}
      },
      "required": ["Title", "Authors", "Date"],
      "additionalProperties": false
    }
  },
  "required": ["Book"],
}
```

The properties (key-value pairs) on an object are defined using the properties keyword. The value of properties is an object, where each key is the name of a property and each value is a schema used to validate that property. Any property that doesn't match any of the property names in the properties keyword is ignored by this keyword.

The **additionalProperties** keyword is used to control the handling of extra stuff, that is, properties whose names are not listed in the properties keyword. By default any additional properties are allowed.

## Equivalent JSON Schema

The value of the additionalProperties keyword is a schema that will be used to validate any properties in the instance that are not matched by properties. Setting the additionalProperties schema to false means no additional properties will be allowed.

**You can use non-boolean schemas to put more complex constraints on the additional properties of an instance. For example, one can allow additional properties, but only if their values are each a string:**

```
"additionalProperties": { "type": "string" }
```

# JSON - Additional Properties

---

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

Any additional properties must be of type string. Properties "name" and "age" can be present, along with any other string properties.

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

Only "name" and "age" properties are allowed. Any additional properties will cause validation to fail.

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

The object can have any additional properties beyond "name" and "age".

---

# JSON - Additional Properties

Use the following validator to test it out:

<https://www.jsonschemavalidator.net/>

```
{
  "Book": {
    "Title": "Example Book",
    "Authors": [
      {"name": "Ali Yasser"},
      {"name": "Abdullah Mohammed"}
    ],
    "Date": "2022",
    "Publisher": "Springer",
    "Key": "Springer"
  }
}
```

```
{
  "$schema": "http://json-schema.org/draft-04/schema",
  "properties": {
    "Book": {
      "properties": { "Authors": {
        "type": "object", "properties": {
          "name": {
            "type": "string"
          }
        }
      },
      "required": ["name"]
    },
    "Date": {
      "pattern": "^[0-9]{4}$",
      "type": "string"
    },
    "Publisher": {
      "enum": [ "Springer", "MIT Press",
        "Harvard Press"
      ],
      "type": "string"
    },
    "Title": {
      "type": "string"
    }
  },
  "required": [
    "Title", "Authors", "Date"
  ],
  "type": "object", "additionalProperties": {
    "type": "integer"
  }
}
},
"required": [ "Book"],
"type": "object",
}
```

# Authors list

---

```
{
  "$schema": "http://json-schema.org/draft-04/schema",
  "type": "object",
  "properties": {
    "Book": {
      "type": "object",
      "properties": {
        "Title": {"type": "string"},
        "Authors": {"type": "array", "minItems": 1, "maxItems": 5, "items": {"type": "string"}},
        "Date": {"type": "string", "pattern": "^[0-9]{4}$"},
        "Publisher": {"type": "string", "enum": ["Springer", "MIT Press", "Harvard Press"]}
      },
      "required": ["Title", "Authors", "Date"],
      "additionalProperties": false
    }
  },
  "required": ["Book"],
  "additionalProperties": false
}
```



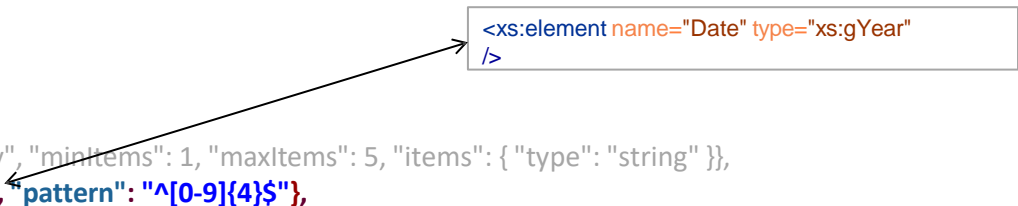
The diagram shows an arrow pointing from the `"Authors": {"type": "array", "minItems": 1, "maxItems": 5, "items": {"type": "string"}}` property in the JSON Schema to the corresponding XML Schema definition.

```
<xs:element name="Authors">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Author" type="xs:string" maxOccurs="5"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

# Date with year type

---

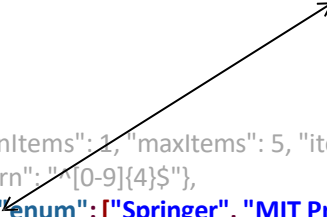
```
{
  "$schema": "http://json-schema.org/draft-04/schema",
  "type": "object",
  "properties": {
    "Book": {
      "type": "object",
      "properties": {
        "Title": {"type": "string"},
        "Authors": {"type": "array", "minItems": 1, "maxItems": 5, "items": {"type": "string"}},
        "Date": {"type": "string", "pattern": "^[0-9]{4}$"},
        "Publisher": {"type": "string", "enum": ["Springer", "MIT Press", "Harvard Press"]}
      },
      "required": ["Title", "Authors", "Date"],
      "additionalProperties": false
    }
  },
  "required": ["Book"],
  "additionalProperties": false
}
```



A diagram consisting of a black arrow pointing from the `"Date": {"type": "string", "pattern": "^[0-9]{4}$"}` property in the JSON Schema to a box containing the XML Schema element `<xs:element name="Date" type="xs:gYear" />`. The box has a thin black border and the text is color-coded: `<xs:element` is blue, `name="Date"` is orange, `type="xs:gYear"` is orange, and `/>` is blue.

# Publisher with enumeration

```
{
  "$schema": "http://json-schema.org/draft-04/schema",
  "type": "object",
  "properties": {
    "Book": {
      "type": "object",
      "properties": {
        "Title": {"type": "string"},
        "Authors": {"type": "array", "minItems": 1, "maxItems": 5, "items": { "type": "string" }},
        "Date": {"type": "string", "pattern": "[0-9]{4}$"},
        "Publisher": {"type": "string", "enum": ["Springer", "MIT Press", "Harvard Press"]}
      },
      "required": ["Title", "Authors", "Date"],
      "additionalProperties": false
    }
  },
  "required": ["Book"],
  "additionalProperties": false
}
```



The diagram illustrates the mapping between the JSON Schema `enum` property and the XSD `xs:enumeration` elements. An arrow points from the `enum` array in the JSON Schema to the XSD code block.

```
<xs:element name="Publisher" minOccurs="0">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="Springer" />
      <xs:enumeration value="MIT Press" />
      <xs:enumeration value="Harvard Press" />
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

# Online JSON Schema validator

---

The image shows the user interface of an online JSON Schema validator. It is divided into two main sections: 'Schema' on the left and 'Validation results' on the right. The 'Schema' section contains two text input fields. The top field is labeled 'Schema:' and contains the text 'Paste your JSON Schema in here' with a circled '1' next to it. The bottom field is labeled 'Data:' and contains the text 'Paste your JSON in here' with a circled '2' next to it. Below the 'Data' field is a 'Validate' button and a link '(load sample data)'. An arrow points from the text 'Click on the validate button' with a circled '3' to the 'Validate' button. The 'Validation results' section is a large gray area labeled 'Validation results:' at the top, containing the text 'Results of validation is shown here' with a circled '4' next to it.

Schema:

Paste your JSON Schema in here 1

Data:

Paste your JSON in here 2

Validate (load sample data)

Validation results:

Results of validation is shown here 4

Click on the validate button 3

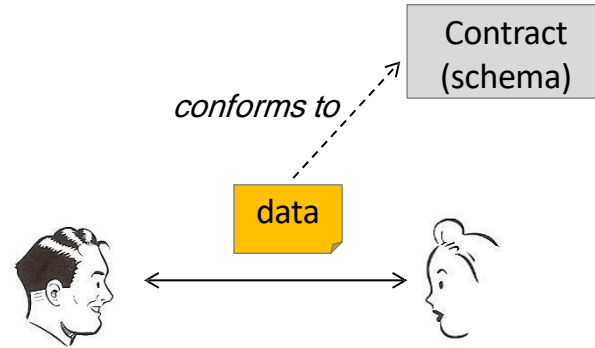
---

<https://www.jsonschemavalidator.net/>

<https://jsonformatter.org/#>

# A contract for data exchanges

Both XML Schema and JSON Schema may be used as a contract for data exchanges:





# Python and JSON

# Working with JSON in Python

---

- ▶ Reading JSON
- ▶ Writing JSON

# Comparison of Data Types

---

## **JSON**

object

array

string

number (int)

number (real)

true

false

null

## **Python**

dict

list

str

int

float

True

False

None

# JSON Read

```
import json

f = open('data.json')
data = json.load(f)
f.close()
print(data)
print(data["features"])
print(data["features"][0]["geometry"])

for i in data["features"]:
    print(i["geometry"]["coordinates"][0])
```

```
{
  "type": "FeatureCollection",
  "features": [ {
    "type": "Feature",
    "geometry": {
      "type": "Point",
      "coordinates": [42.0, 21.0]
    },
    "properties": {
      "prop0": "value0"
    }
  } ]
}
```

```
{'type': 'FeatureCollection', 'features': [{'type': 'Feature', 'geometry': {'type': 'Point', 'coordinates':
[42.0, 21.0]}, 'properties': {'prop0': 'value0'}}]}
-----
[{'type': 'Feature', 'geometry': {'type': 'Point', 'coordinates': [42.0, 21.0]}, 'properties': {'prop0': 'v
alue0'}}]
-----
{'type': 'Point', 'coordinates': [42.0, 21.0]}
-----
42.0
```

# JSON write

---

```
import json
```

```
f = open('data.json')
```

```
data = json.load(f)
```

```
f.close()
```

```
f = open('out.json', 'w')
```

```
json.dump(data, f)
```

```
f.close()
```

---

# Serialisation

---

**Serialisation** is the converting of code objects to a **storage format**; usually some kind of file.

**Deserialization** (~unmarshalling): the conversion of storage-format objects back into working code.

The json code essentially does this for simple and container Python variables.

For more complicated objects, see **pickle**:

<https://docs.python.org/3/library/pickle.html>

---

# Formatted printing

`json.loads` and `json.dumps` convert Python objects to JSON strings.

Dumps has a nice print formatting option:

```
print(json.dumps(data["features"], sort_keys=True, indent=4))
```

```
[
  {
    "geometry": {
      "coordinates": [
        42.0,
        21.0
      ],
      "type": "Point"
    },
    "properties": {
      "prop0": "value0"
    },
    "type": "Feature"
  }
]
```

More on the JSON library at:

<https://docs.python.org/3/library/json.html>

# JSON Conversions in Python

---

- ▶ JSON  $\leftrightarrow$  Python Dictionary
- ▶ JSON  $\leftrightarrow$  XML
- ▶ JSON  $\leftrightarrow$  BSON

# JSON vs. Python Dictionary

**JSON:** {"name": "John", "age": 30, "active": true}

**Python Dictionary:** {"name": "John", "age": 30, "active": True}

<u>Aspect</u>	<u>JSON</u>	<u>Python Dictionary</u>
Type	Text-based format (string)	In-memory Python object
Keys	Strings only	Any hashable type (str, int, etc.)
Values	Basic types (str, num, etc.)	Any Python type
Usage	Data exchange, storage	In-program data manipulation
Mutability	Immutable (as string)	Mutable
Syntax	"key": "value", true, null	'key': 'value', True, None
Serialization	Native (it's a string)	Requires json.dumps()

# JSON and Python Dictionary conversion

```
import json
```

---

```
# 1. JSON string to Python dictionary
```

```
json_string = '{"name": "John", "age": 30, "active": true}'
```

```
python_dict = json.loads(json_string)
```

```
print("JSON to Dictionary:")
```

```
print(python_dict) # Output: {'name': 'John', 'age': 30, 'active': True}
```

```
print(type(python_dict)) # Output: <class 'dict'>
```

```
# 2. Python dictionary to JSON string
```

```
python_dict["city"] = "New York" # Adding a new key-value pair
```

```
json_output = json.dumps(python_dict)
```

```
print("\nDictionary to JSON:")
```

```
print(json_output) # Output: {"name": "John", "age": 30, "active": true, "city": "New York"}
```

```
print(type(json_output)) # Output: <class 'str'>
```

---



# JSON vs XML

Aspect	XML	JSON
Format	Tag-based ( <code>&lt;tag&gt;value&lt;/tag&gt;</code> )	Key-value ( <code>"key": "value"</code> )
Readability	Verbose, readable	Compact, highly readable
Data Types	Text only, no native types	Strings, numbers, booleans, etc.
Attributes	Supported (e.g., <code>id="1"</code> )	Not supported
Size	Larger, more overhead	Smaller, lightweight
Parsing	Complex (DOM/SAX)	Simple (native in most langs)
Use Cases	Enterprise, SOAP, documents	Web APIs, mobile, NoSQL
Schema	Strong (XSD, DTD)	Optional (JSON Schema)
Comments	Yes ( <code>&lt;!-- --&gt;</code> )	No

# Convert XML to JSON in Python

```
import xmltodict
import json

# Example XML string
xml_string = '''
<person id="1">
    <name>John</name>
    <age>30</age>
    <active>true</active>
</person>
'''

# Convert XML to Python dictionary
xml_dict = xmltodict.parse(xml_string)

# Convert dictionary to JSON string
json_output = json.dumps(xml_dict, indent=2)

# Print the result
print("XML to JSON:")
print(json_output)
```

---

## Output:

```
json

{
  "person": {
    "@id": "1",
    "name": "John",
    "age": "30",
    "active": "true"
  }
}
```

---

# Convert JSON to XML in Python

```
import json
from dicttoxml import dicttoxml

# JSON string with a list
json_string = '''
{
    "person": {
        "id": 1,
        "name": "John",
        "age": 30,
        "active": true,
        "skills": ["Python", "JavaScript", "SQL"]
    }
}
'''

# Convert JSON to Python dictionary
json_dict = json.loads(json_string)

# Custom function to name list items "skill" instead of "item"
xml_output = dicttoxml(json_dict, custom_root='root', attr_type=False,
item_func=lambda x: 'skill')

# Decode bytes to string and print
xml_string = xml_output.decode('utf-8')
print("JSON to XML with Custom List Tags:")
print(xml_string)
```

## Output:

```
xml

<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <person>
    <id>1</id>
    <name>John</name>
    <age>30</age>
    <active>true</active>
    <skills>
      <skill>Python</skill>
      <skill>JavaScript</skill>
      <skill>SQL</skill>
    </skills>
  </person>
</root>
```

# JSON vs. BSON

Aspect	JSON	BSON	MongoDB	Firebase
Format	Text-based, human-readable	Binary-encoded, machine-readable	Uses BSON internally	Uses JSON
Data Types	Basic (string, number, etc.)	Extended (Date, ObjectId, etc.)	Full BSON support	JSON types only
Readability	Easy to read/edit	Not human-readable	BSON (binary)	JSON (readable)
Performance	Slower, less compact	Faster, more compact	Optimized with BSON	JSON-based, real-time optimized
Primary Use	Data exchange (APIs, configs)	Storage/processing (databases)	Document storage (BSON)	Realtime & Firestore (JSON)
Database Use	Widely supported	MongoDB, EJDB, TokuMX	Native BSON	No BSON, JSON only

# Converting JSON to BSON in Python

---

```
from bson import encode
```

```
import json
```

```
# Example JSON string
```

```
json_string = '{"name": "John", "age": 30, "active": true}'
```

```
# Parse JSON string into a Python dictionary
```

```
python_dict = json.loads(json_string)
```

```
# Convert Python dictionary to BSON
```

```
bson_data = encode(python_dict)
```

```
# The result is a binary string
```

```
print(bson_data) # Outputs something like b'\x1e\x00\x00\x00\x02name\x00\x05...' (
```

---

# Converting BSON to JSON in Python

---

```
from bson import decode, encode
import json
```

```
# Example JSON string
```

```
json_string = '{"name": "John", "age": 30, "active": true}'
```

```
# Convert JSON to Python dict and then to BSON
```

```
python_dict = json.loads(json_string)
```

```
bson_data = encode(python_dict)
```

```
# Convert BSON back to a Python dictionary
```

```
decoded_dict = decode(bson_data)
```

```
# Convert Python dictionary to JSON string
```

```
json_output = json.dumps(decoded_dict)
```

```
print(decoded_dict)  # Outputs: {'name': 'John', 'age': 30, 'active': True}
```

```
print(json_output)   # Outputs: {"name": "John", "age": 30, "active": true}
```

---

# Review

---

- ▶ JSON represents structured data
  - ▶ Basic types: string, number, Boolean, null
  - ▶ Arrays (lists) use square brackets [ ]
  - ▶ Objects (dictionaries) use curly brackets { }
  - ▶ Collections can contain collections
  - ▶ White space doesn't matter
  - ▶ Indent for every level of collection
-