JSON

Portable data with a small, strict syntax.

01 A JSON document

```
{
  "name": "Maya",
  "age": 28,
  "active": true
}
```

JSON is text data. A document contains one value: object, array, string, number, Boolean, or null. Parse JSON; do not evaluate it as code.

02 The six value types

```
[
  "text",
  42,
  true, null,
  [1, 2],
  {"id": "u1"}
]
```

Strings use double quotes; numbers are unquoted. true, false, and null are lowercase. Objects/arrays nest values. No date, function, or undefined type exists.

03 Objects & member names

```
{
  "id": "u1",
  "displayName": "Maya Chen",
  "hasAccess": false
}
```

Objects use { } with string names, colons, and comma-separated members. Names should be unique; duplicate behavior varies. Do not depend on member order.

04 Arrays & order

```
[
  "Maya",
  "Leo",
  "Sam"
]
```

Arrays use [ ] and preserve order; [] is empty. Mixed types are allowed, though an API may require one shape. The host language supplies indexing syntax.

05 Strings & escapes

```
{
  "quote": "Say \"hi\"",
  "path": "C:\\temp",
  "lines": "first\nsecond",
  "tab": "a\tb"
}
```

Inside a string, escape quotes as \" and backslashes as \\. \n, \r, and \t encode controls; \u0041 encodes A. Raw control characters such as literal newlines are forbidden inside strings.

06 Number syntax

```
{
  "count": 42,
  "balance": -12.5,
  "scale": 1.2e3
}
```

No leading +, 01, hex, NaN, or Infinity. Fractions need digits after the dot. Parsers limit range/precision. Use strings for exact large identifiers.

07 Null, empty, or missing

```
{
  "middleName": null,
  "nickname": "",
  "tags": []
}
```

null is explicit; missing members differ from null. Empty text/collections, false, and zero are distinct. Their meaning belongs to the API contract.

08 Nested data

```
{
  "user": {
    "id": "u1",
    "roles": ["reader", "editor"]
  },
  "total": 1
}
```

Each object/array is a value inside its parent. Closers must match openers. Parse first, then traverse with your language's object/array operations.

09 Whitespace & strict syntax

```
{
  "enabled": true,
  "retries": 3,
  "options": {}
}
```

Whitespace outside strings does not change values. No comments, trailing commas, or single quotes. Use UTF-8 for interchange; send one value per document.

JSON

Parse, validate, and exchange data.

10 Parse in JavaScript

```
const raw = '{"name":"Maya"}';
try {
  console.log(JSON.parse(raw));
} catch (error) {
  console.error("Invalid JSON", error);
}
```

JavaScript: JSON.parse returns a value; invalid syntax throws SyntaxError. This fixture is an object. Validate untrusted types and fields before access.

11 Serialize in JavaScript

```
const user = { name: "Maya", active: true };
const compact = JSON.stringify(user);
const view = JSON.stringify(user, null, 2);
console.log(compact);
console.log(view);
```

JavaScript: argument three controls indentation. BigInt/cycles throw by default. undefined object properties disappear; undefined array values become null.

12 Validate the parsed shape

```
function isUser(value) {
  return value !== null &&
    typeof value === "object" &&
    !Array.isArray(value) &&
    typeof value.name === "string";
}
console.log(isUser({ name: "Maya" }));
```

JavaScript: parsing does not check expected fields. This guard requires an object with a string name but allows extra members. Also check domain rules.

13 Parse in Python

```
import json
raw = '{"active":true,"value":null}'
try: print(json.loads(raw))
except json.JSONDecodeError as error:
    print(f"Invalid JSON: {error}")
```

Python: loads reads text. Objects become dict, arrays list, Booleans bool, null None. By default NaN/Infinity extensions are accepted; reject them for strict input.

14 Serialize in Python

```
import json
user = {"name": "Maya", "active": True}
text = json.dumps(user, allow_nan=False)
pretty = json.dumps(user, indent=2)
print(text)
print(pretty)
```

Python: dumps returns text; dump writes a file. allow_nan=False rejects non-finite floats. ensure_ascii=False keeps Unicode visible. Unsupported types raise TypeError.

15 Read a JSON file

```
import json
path = "in.json"
try:
  with open(path, encoding="utf-8") as f:
    data = json.load(f)
except (OSError, ValueError) as error:
  print(f"Read failed: {error}")
```

Python: with closes in.json. Handlers cover I/O, text decoding, and JSON syntax failures. Validate the loaded shape; this uses Python's default decoder.

16 Write a JSON file

```
import json, tempfile
from pathlib import Path
with tempfile.TemporaryDirectory() as tmp:
  path = Path(tmp) / "data.json"
  text = json.dumps({"ok": True})
  path.write_text(text, encoding="utf-8")
```

Python: writes stay in a temporary directory and are cleaned up. I/O failures propagate. write_text replaces the file; json.dump can write to an open file.

17 Read an API response

```
const res = await fetch("/api/user");
if (!res.ok) {
  throw new Error(`HTTP ${res.status}`);
}
const data = await res.json();
console.log(data);
```

JavaScript: inside an async browser function. Use your API endpoint. json() parses; validate afterward. Network/parse failures reject; handle in the caller.

18 Interoperability pitfalls

Large integers	JS Numbers may round them.
Duplicate names	Avoid: parsers can disagree.
undefined	Not a JSON value.
NaN / Infinity	Not valid JSON numbers.
Dates	Use an agreed string format.

Agree on types, nulls, units, and dates. JS non-finite numbers stringify as null; Python can reject.