JSON(JAVA SCRIPT OBJECT NOTATION)

A JSON refers to the **JavaScript Object Notation** format used to store simple objects and data structures. Usually, JSON files are backup files, which is used to take backup of data that restored back to the application when needed.

Why we use JSON files -

- The JSON files are lightweight and take less storage to be stored.
- These files can be usually created and edited by a text editor. These text editors are mostly freely available.
- JSON files are human-readable means the user can read them easily.
- These files can be opened in any simple text editor like Notepad, which is easy to use.
- Almost every programming language supports JSON format because they have libraries and functions to read/write JSON structures.

JSON or JAVASCRIPT object NOTATION is a light weight text-based open standard designed for human-readable data interchange.

Convention used by JSON are known to programmers which include C,C++, Java, Python, Perl etc.

JSON stands for JAVA SCRIPT OBJECT NOTATION.

it was designed for human-readable data interchange.

It has been extended from javascript scripting language.

The file name extention is .json.

JSON internet media type is application/json.

JSON supports mainly 6 data types:

String.

Number

Boolean

null

object

array

KEYS ENCLOSED IN DOUBLE QOUTES(" ") IN JSON

JSON OR XML IS USED FOR DATA PARSING FOR API.

SIMILAR TO THAT OF JAVASCRIPT OBJECT LITERALS

EXAMPLE:{

"name": 'manu',

"age": 20,

"company":'jspiders',

"salary" : 30000,

"designation" : 'nodejs developer',

"skills" : ["java","python","nodejs"]

}

PRACTICAL USE OF JsON FILE.

window

.fetch("http://api.github.com/users")

JSON CANNOT ACCEPT FUNCTION DATA TYPE.

JAVASCRIPT OBJECT LITERALS CAN ACCEPT FUNCTION DATA TYPE.

Advantages of JSON file

Below are some advantages of JSON files -

1. JSON files are computer-readable as well as human-readable. So, both humans and computers can read and write JSON files.
2. JSON format is considered as an independent file format. However, this format was originally based on a subset of JavaScript.
3. Almost every programming language supports JSON format because they have libraries and functions to read/write JSON structures.
4. JSON files are compact.
5. It can easily map with the data structures used by most of the programming languages.

OBJECT enteries:

the object.entries() method returns an array of a given object's own enumeraable

string-keyed property[key,value] pair

this:

what is "this" keyword in js

1.this keyword refers to an object, that object which is executing the xurrent bit of js code.

2.in other words evey js function executing as a reference to its current execution context

called this. Execution context here is how the function is called.

Promises:

What is a Promise?

A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that it's not resolved (e.g., a network error occurred). A promise may be in one of 3 possible states: fulfilled, rejected, or pending. Promise users can attach callbacks to handle the fulfilled value or the reason for rejection.

How Promises Work

A promise is an object which can be returned synchronously from an asynchronous function. It will be in one of 3 possible states:

- **Fulfilled:** `onFulfilled()` will be called (e.g., `resolve()` was called)
- **Rejected:** `onRejected()` will be called (e.g., `reject()` was called)
- **Pending:** not yet fulfilled or rejected

A promise is **settled** if it's not pending (it has been resolved or rejected). Sometimes people use *resolved* and *settled* to mean the same thing: *not pending*.

Once settled, a promise can not be resettled. Calling `resolve()` or `reject()` again will have no effect. The immutability of a settled promise is an important feature

You can optionally `resolve()` or `reject()` with values, which will be passed to the callback functions attached with `.then()`.

When I `reject()` with a value, I always pass an `Error` object. Generally I want two possible resolution states: the normal happy path, or an exception — anything that stops the normal happy path from happening. Passing an `Error` object makes that explicit.

Promises following the spec must follow a specific set of rules:

- A promise or “thenable” is an object that supplies a standard-compliant `.then()` method.
- A pending promise may transition into a fulfilled or rejected state.
- A fulfilled or rejected promise is settled, and must not transition into any other state.
- Once a promise is settled, it must have a value (which may be `undefined`). That value must not change.

Change in this context refers to identity (`===`) comparison. An object may be used as the fulfilled value, and object properties may mutate.

Every promise must supply a `.then()` method with the following signature:

```
promise.then(
  onFulfilled?: Function,
  onRejected?: Function
) => Promise
```

The `.then()` method must comply with these rules:

- Both `onFulfilled()` and `onRejected()` are optional.
- If the arguments supplied are not functions, they must be ignored.
- `onFulfilled()` will be called after the promise is fulfilled, with the promise’s value as the first argument.
- `onRejected()` will be called after the promise is rejected, with the reason for rejection as the first argument. The reason may be any valid JavaScript value, but because rejections are essentially synonymous with exceptions, I recommend using Error objects.
- Neither `onFulfilled()` nor `onRejected()` may be called more than once.

- `.then()` may be called many times on the same promise. In other words, a promise can be used to aggregate callbacks.
- `.then()` must return a new promise, `promise2`.
- If `onFulfilled()` or `onRejected()` return a value `x`, and `x` is a promise, `promise2` will lock in with (assume the same state and value as) `x`. Otherwise, `promise2` will be fulfilled with the value of `x`.
- If either `onFulfilled` or `onRejected` throws an exception `e`, `promise2` must be rejected with `e` as the reason.
- If `onFulfilled` is not a function and `promise1` is fulfilled, `promise2` must be fulfilled with the same value as `promise1`.
- If `onRejected` is not a function and `promise1` is rejected, `promise2` must be rejected with the same reason as `promise1`.

Promise Chaining

Because `.then()` always returns a new promise, it's possible to chain promises with precise control over how and where errors are handled. Promises allow you to mimic normal synchronous code's `try`/`catch` behavior.

Like synchronous code, chaining will result in a sequence that runs in serial. In other words, you can do:

```
fetch(url)
  .then(process)
  .then(save)
  .catch(handleErrors)
;
```

Assuming each of the functions, `fetch()`, `process()`, and `save()` return promises, `process()` will wait for `fetch()` to complete before starting, and `save()` will wait for `process()` to complete before starting. `handleErrors()` will only run if any of the previous promises reject.