JSON Documents, Collections & Database Comparison

a) JSON Documents

- JSON documents consist of fields, which are name-value pair objects. The fields can be in any order, and be nested or arranged in arrays.
- The field name is always of String type and serves as key for the field, and, therefore, it must be unique. The values can be any of the supported JSON data types.
- Data defined in documents formatted according to the JavaScript Object Notation (JSON) is stored in JSON format.
- The content of a JSON document may be deeply structured, however, unlike XML, a JSON document does not have a schema.

b) JSON Collections

- A collection may store a number of documents.
- Collections contain JSON documents that you can add, find, update, and remove.
- A collection is a group of JSON documents that exist within one database. Documents within a collection can have different fields, although, generally all documents in a collection have a similar or related purpose.
- To uniquely identify each document in a collection, a document identifier is either specified as id-field in the document or is automatically generated. An index, or primary key, is automatically generated for the unique document identifiers.

c) Database Comparison

- A collection may store documents those who are not same in structure.
- This is possible because MongoDB is a Schema-free database.
- In a relational database like MySQL, a schema defines the organization / structure of data in a database.
- MongoDB does not require such a set of formula defining structure of data. So, it is quite possible to store documents of varying structures in a collection.

S.No.)	RDBMS	Non RDBMS
1.	Table	Collections
2.	Column	Key
3.	Value	Value
4.	Rows/Records	Documents / Objects

JSON.parse() & JSON.stringify() methods

There are two useful methods to deal with JSON-formatted content: JSON.stringify() & JSON.parse(), which are useful to learn as a pair.

a) JSON.parse() method

- JSON parse() method, as the name suggests, deserializes a JSON string representation to a JavaScript value or JSON object.
- JSON.parse() takes a JSON string and then transforms it into a JavaScript object.
- The parse() method ,optionally, can use a reviver function to perform a transformation on the resulting object before it is returned.

Syntax

```
JSON.parse(string, reviver)
```

where,
string --> The string to parse as JSON.
reviver --> If a function, this prescribes how the value originally produced by parsing is transformed, before being returned.

Example

```
var jsonString = '{ "x": 5, "y": 6 }';
var obj = JSON.parse( jsonString );
console.log(obj);
```

Example

```
var obj = JSON.parse('{"1": 1, "2": 2, "3": {"4": 4, "5": {"6": 6}}}', (key, value) => {
  if(value % 2 == 0)
    return value * 2; //If value is even number than multiply by 2
  else
    return value;    //else return original value
});
```

Example

```
var jsonArrayString = '["A", "B", "C"]';
 var arrObject = JSON.parse( jsonArrayString );
 console.log(arrObject);
```

Example

```
const json = '{"result":true, "count":42}';
const obj = JSON.parse(json);
console.log(obj.count);
console.log(obj.result);
```

Example

```
const str = '{"name":"Bablu","image":"🐶","age":"6 months"}';
const dog = JSON.parse(str);
console.log(dog.name);
console.log(dog.image);
console.log(dog.age);
```

Example

```
const str = '{"id":99,"name":"Bablu","image":"🐶","age":"6 months"}';
const dog = JSON.parse(str, (key, value) => {
    if(typeof value === 'string') {
        return value.toUpperCase();
    }
    return value;
});
console.log(dog.id);
console.log(dog.name);
console.log(dog.image);
console.log(dog.age);
```

Example

```
const user = {
  name: 'NehaRathore',
  email: 'neha@awesome.com',
  plan: 'Pro'
};
const userStr = JSON.stringify(user);
console.log(
    JSON.parse(userStr, (key, value) => {
  if (typeof value === 'string') {
    return value.toUpperCase();
  }
  return value;
})
)
```

Example

```
const myObj = {
  name: 'Ajay',
  age: 30,
  favoriteFood: 'Puff'
};
const myObjStr = JSON.stringify(myObj);
console.log(
JSON.parse(myObjStr, (key, value) => {
  if (typeof(value) === 'string') {
    return value.toUpperCase();
  }
  else if (typeof(value) === 'number') {
    return value+10;
  }
  return value;
}));
```

b) JSON.stringify() method

- JSON.stringify() takes a JavaScript object and transforms it into a JSON string.
- This method is useful while sending JSON data from a client to a server, it must be converted into a JSON string.

Syntax

```
JSON.stringify(value, replacer, space)
```

where,

- value --> The JavaScript object which needs to be converted to a JSON string.
- replacer --> The function that alters the behavior of the stringification process. If replace function is not provided, all properties of the object are included in the resulting JSON string.
- space --> For indentation purpose. A String or Number that's used to insert whitespaces into the output JSON string for readability purposes. If this is a Number, it indicates the number of space characters to use as whitespace.

Example

```
console.log( JSON.stringify({ x: 5, y: 6 }) );
console.log( JSON.stringify({ x: 5, y: 6 }, null, '  ') ); //with space
```

Example

```
var obj = { name: "Neha Rathore", age: 23, city: "Agra" };
var myJSON = JSON.stringify(obj);
console.log(myJSON)
```

Example

```
console.log(JSON.stringify({ x: 5, y: 6 }));
// expected output: "{"x":5,"y":6}"
console.log(JSON.stringify([new Number(3), new String('false'), new
Boolean(false)]));
// expected output: "[3,"false",false]"
console.log(JSON.stringify({ x: [10, undefined, function(){}, Symbol('')] }));
// expected output: "{"x":[10,null,null,null]}"
console.log(JSON.stringify(new Date(2006, 0, 2, 15, 4, 5)));
// expected output: ""2006-01-02T15:04:05.000Z"
```

Example

```
var arr = [ "Neha", "Tushar", "Garima", "Soniya" ];
var myJSON = JSON.stringify(arr);
console.log(myJSON);
```

Example

```
const myObj = {
  name: 'Soniya',
  age: 22,
  favoriteFood: 'Momos'
};
const myObjStr = JSON.stringify(myObj);
console.log(myObjStr);
// "{"name":"Soniya","age":22,"favoriteFood":"Momos"}"
console.log(JSON.parse(myObjStr));
// Object {name:"Soniya",age:22,favoriteFood:"Momos"}
```

Example

```
const user = {
  id: 229,
  name: 'Neha Rathore',
  email: 'neha@awesome.com'
};
function replacer(key, value) {
  console.log(typeof value);
  if (key === 'email') {
    return undefined;
  }
  return value;
}
const userStr = JSON.stringify(user, replacer);
```

Example

```
const user = {
  name: 'Neha Rathore',
  email: 'neha@awesome.com',
  plan: 'Pro'
};
const userStr = JSON.stringify(user, null, '...');
```

Example

```
const myObj = {
  name: 'Soniya',
  age: 22,
  favoriteFood: 'Momos'
};
const myObjStr = JSON.stringify(myObj,replacer);
function replacer(key, value) {
  if (typeof(value) === 'string') {
    return value.toUpperCase();
  }
  else if (typeof(value) === 'number' && key === "age" && value>100) {
    return undefined;
  }
  return value;
}
console.log(myObjStr);
```

Example

```
const myObj = {
  name: 'Soniya',
  age: 22,
  favoriteFood: 'Momos'
};
const myObjStr = JSON.stringify(myObj,replacer,"---");
function replacer(key, value) {
  if (typeof(value) === 'string') {
    return value.toUpperCase();
  }
  else if (typeof(value) === 'number' && key === "age" && value>100) {
    return undefined;
  }
  return value;
}
console.log(myObjStr);
```

Example

```
const user = {
    id: 599,
    name: 'Neha Rathore',
    email: 'neha@example.com',
    password: '123abc',
    age: 23,
    address: {
        city: 'Agra',
        country: 'India'
    },
    hobbies: ['ChitChatting', 'Class Bunk', 'Late HomeWork']
};
const str = JSON.stringify(user, (key, value) => {
    // filter-out password from final string
    if (key === 'password') {
        return undefined;
    }
    return value;
});

console.log(str);
```

Example

```
const dog = {
    name: 'Bablu',
    image: '🐶',
    age: '6 months'
};

const str = JSON.stringify(dog, null, '----');

console.log(str);
```

Example -- Undefined Value

JSON.stringify() does one of the following with an undefined value :-

1) Omits the value if it's part of an Object.
2) Converts the value to null if that value belongs to an Array.

```
var siteInfo = {
    "name"   : undefined,
    "users" : [
                [ "admins", "1", "2" , undefined],
                    [ "editors", "4", "5" , "6"],
              ]
}

console.log( JSON.stringify(siteInfo) );
```

Schema

- Design of a database is called the schema.
- A database schema is the skeleton structure that represents the logical view of the entire database. It defines how the data is organized and how the relations among them are associated.
- A schema contains schema objects, which could be tables, columns, data types, views, stored procedures, relationships, primary keys, foreign keys, etc.
- Database schema is designed for both RDBMS & Non-RDBMS databases.

A database schema can be divided broadly into three categories:-

a) View Schema

- Define as the design of database at view level.
- At view level, a user can able to interact with the system.

b) Logical Schema

- Programmer work on that level.
- This schema defines all the logical constraints that need to be applied on the data stored. It defines tables, views, and integrity constraints.

c) Physical Schema

- This schema pertains to the actual storage of data and its form of storage like files, indices, etc.
- It defines how the data will be stored in a secondary storage.

JSON Schema

- JSON Schema is a specification for JSON based format for defining the structure of JSON data.
- JSON Schema is a specification for JSON based format for defining the structure of JSON data. It was written under IETF draft which expired in 2011.
- JSON schema is basically a structure for our JSON documents that defines the expected data types & format of each field in the response.
- JSON Schema is a declarative way of writing validation rules. It contains all the steps that are necessary to be performed in order to validate something.
- JSON schema is generally used to automate or validate the response of JSON.

Validates data which is useful for :-

a) Automated testing.
b) Ensuring quality of client submitted data

JSON Schema Benefits

JSON Schema includes :-

a) Describes your existing data format.
b) Clear, human- and machine-readable documentation.
c) Complete structural validation, useful for automated testing.
d) Complete structural validation, validating client-submitted data.

JSON Schema Validators

1 - Json Schema Validator --> https://www.jsonschemavalidator.net/
2 - Liquid Technologies --> https://www.liquid-technologies.com/online-json-schema-validator
3 - Json Schema Lint --> https://jsonschemalint.com/#!/version/draft-07/markup/json
4 - JSON Schema Validator --> https://jsonschema.dev/

JSON Schema Syntax

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",

  "properties": {

    "id": {
      "description": "The unique identifier for a product",
      "type": "integer"
    },

    "name": {
      "description": "Name of the product",
      "type": "string"
    },

    "price": {
      "type": "number",
      "minimum": 0,
      "exclusiveMinimum": true
    }
  },

  "required": ["id", "name", "price"]
}
```

where,

- "$schema" & "$id" are the schema keywords.
- "title" & "description" are the schema annotations.
- "type" is the validation keyword.

Syntax Keywords Explanations

1) $schema

- The '$schema' keyword states that this schema is written according to the draft v7 specification.
- The value of this keyword must be a string representing an URI.
- This keyword is not required and if it is missing, the URI of the latest schema version will be used instead.
- The only difference between 'draft 06' and 'draft 07' is that 'draft 06' does not support 'if-then-else' keywords.

2) $id

- This keyword is used to specify an unique ID for a document or a document subschemas.
- The value of this keyword must be a string representing an URI.
- It is not a required keyword, but for best practice use it.

3) title

- The 'title' keyword give us the title for our schema.
- The value of this keyword must be a string.

4) description

- The 'description' keyword defines a little description of the schema.
- The value of this keyword must be a string.

5) type, const & enum keywords

The type, const, and enum generic keywords, allows us to validate JSON data, by checking if its value, or its type, matches a given value or type.

5.1) 'type' keyword

- The type keyword specifies the type of data that the schema is expecting to validate.
- This keyword is not mandatory and the value of the keyword must be a string, representing a valid data type, or an array of strings, representing a valid list of data types.

Example

JSON Schema

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

JSON Status

a) "some text" --> valid
b) "" --> valid - empty string
c) 12 --> invalid - is integer/number
d) null --> invalid - is null

Example

JSON Schema

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

JSON Status

a) {"a": 1} --> valid - is object
b) null --> valid - is null
c) "1, 2, 3" --> invalid - is string
d) [{"a": 1}, {"b": 2}] --> invalid - is array

Example

JSON Schema

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

JSON Status

a) -10.5 --> valid - is number
b) "some string" --> valid - is string
c) null --> valid - is null
d) false --> invalid - is boolean
e) {"a": 1} --> invalid - is object
f) [1, 2, 3] --> invalid - is array

5.2) 'const' keyword

An instance validates against this keyword if its value equals to the value of this keyword. The value of this keyword can be anything.

Example

JSON Schema

```
{
  "const": "test"
}
```

JSON Status

a) "test" --> valid
b) "Test" --> invalid
c) "tesT" --> invalid
d) 3.4 --> invalid

Example

JSON Schema

```
{
  "const": {
    "a": 1,
    "b": "2"
  }
}
```

JSON Status

a) {"a": 1, "b": "2"} --> valid
b) {"b": "2", "a": 1} --> valid
c) {"a": 1, "b": "2", "c": null} --> invalid
d) 5.10 --> invalid

5.3) 'enum' keyword

- An instance validates against this keyword if its value equals can be found in the items defined by the value of this keyword.
- The value of this keyword must be an array containing anything.
- An empty array is not allowed.

Example

JSON Schema

```
{
  "enum": ["a", "b", 1, null]
}
```

JSON Status

a) "a" --> valid
b) "b" --> valid
c) 1 --> valid
d) null --> valid
e) "A" --> invalid
f) -1 --> invalid
g) false --> invalid
h) ["a", "b", 1, null] --> invalid

6) properties

The 'properties' keyword defines various keys and their value types, minimum and maximum values to be used in JSON file.

7) required

The 'required' keyword keeps a list of required properties.

Example

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>JSON Schema Validation Example</title>
  <!-- Use the UMD version of Ajv for browser compatibility -->
  <script src="https://cdn.jsdelivr.net/npm/ajv@6.12.6/dist/ajv.min.js"></script>
</head>
<body>
  <h2>JSON Schema Validation with Plain JavaScript</h2>
  <form id="jsonForm">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name" required><br><br>
    <label for="age">Age:</label>
    <input type="number" id="age" name="age" required><br><br>
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" required><br><br>
    <button type="submit">Submit</button>
  </form>

  <p id="validationMessage"></p>

  <script>
    // Define the JSON Schema
    const schema = {
      type: "object",
      properties: {
        name: { type: "string" },
        age: { type: "integer", minimum: 0 },
        email: { type: "string", format: "email" }
      },
      required: ["name", "age", "email"],
      additionalProperties: false
    };
```

```
    // Create a new Ajv instance (UMD version works here)
    const ajv = new Ajv();

    // Form submission event listener
    document.getElementById('jsonForm').addEventListener('submit', function(event) {
      event.preventDefault(); // Prevent the form from submitting

      // Get the form data
      const formData = {
        name: document.getElementById('name').value,
        age: parseInt(document.getElementById('age').value),
        email: document.getElementById('email').value
      };

      console.log("Form Data:", formData); // Debug: log form data

      // Validate form data against the schema
      const validate = ajv.compile(schema);
      const valid = validate(formData);

      const validationMessage = document.getElementById('validationMessage');

      if (valid) {
        validationMessage.textContent = "JSON data is valid!";
        validationMessage.style.color = "green";
        console.log("Valid JSON Data");
      } else {
        validationMessage.textContent = "JSON data is invalid: " +
JSON.stringify(validate.errors, null, 2);
        validationMessage.style.color = "red";
        console.log("Validation Errors:", validate.errors);
      }
    });
  </script>

</body>
</html>
```