CHAPTER 6

Understanding JSON

As SQL Server evolves, more and more nonrelational features are being added to the product, blurring the lines between relational and NoSQL technologies. JSON is an example of this. JSON (JavaScript object notation) is a document format, designed as a method of lightweight data interchange. It is similar to XML, in the respect that it is a self-describing, hierarchical data-interchange format. Unlike XML, however, JSON tags are minimal, making JSON documents shorter and both easier to read and quicker to parse.

In this chapter, I will introduce the JSON format. I will discuss the structure of a JSON document and compare it to an XML document. Finally, I will discuss usage scenarios for JSON data within SQL Server.

Understanding the JSON Format

The basic JSON syntax uses name/value pairs, separated by a colon. The JSON object is then enclosed by braces. The name must be a string, enclosed with double quotes, and the value must be

- A string (enclosed by double quotes)
- A number
- A nested JSON object
- A Boolean value
- An array (enclosed by square brackets)
- `NULL`

© Peter A. Carter 2018

For instance, consider the simple example in Listing 6-1.

Listing 6-1. Simple JSON Document

```
{ "FirstName" : "Pete" }
```

If multiple name/value pairs occur within a JSON document, they are separated by a comma. For example, consider the JSON document in Listing 6-2. You will notice that the value for age is not enclosed in double quotes, because it is a number, as opposed to a string.

Listing 6-2. Simple JSON Document with Multiple Name/Value Pairs

```
{ "FirstName" : "Pete" , "LastName" : "Carter" , "Age" : 38 }
```

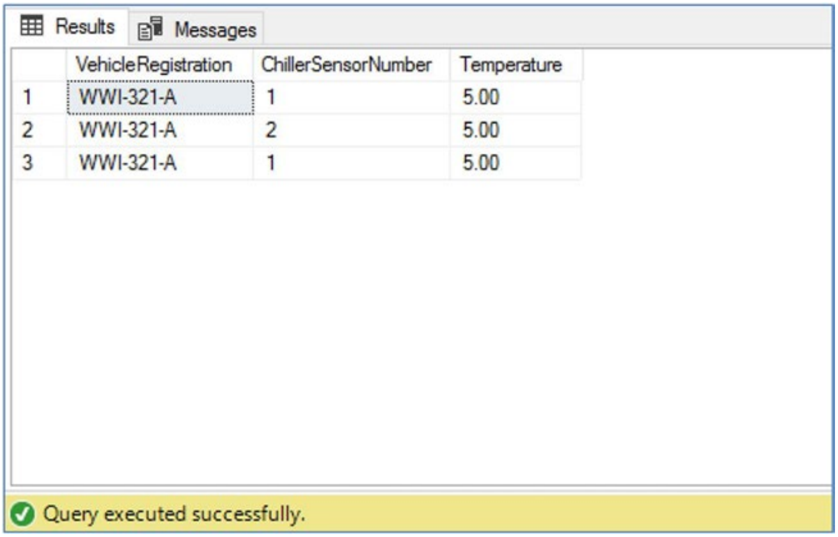
If you were to represent as a JSON flat document a row set in a table, the result would be an array of JSON objects. For example, consider the query in Listing 6-3.

Listing 6-3. Top Vehicle Temperatures

```
USE WideWorldImporters
GO

SELECT TOP 3
          VehicleRegistration
        , ChillerSensorNumber
        , Temperature
FROM Warehouse.VehicleTemperatures
ORDER BY Temperature DESC ;
```

This query will produce the results displayed in Figure 6-1.



	VehicleRegistration	ChillerSensorNumber	Temperature
1	WWI-321-A	1	5.00
2	WWI-321-A	2	5.00
3	WWI-321-A	1	5.00

Query executed successfully.

Figure 6-1. *Top vehicle temperatures*

If this result set were to be expressed as a JSON document, it would look like the document in Listing 6-4.

Listing 6-4. Top Vehicle Temperatures Expressed As JSON

```
[
   {
      "VehicleRegistration": "WWI-321-A",
      "ChillerSensorNumber": 1,
      "Temperature": 5
   },
   {
      "VehicleRegistration": "WWI-321-A",
      "ChillerSensorNumber": 2,
      "Temperature": 5
   },
   {
      "VehicleRegistration": "WWI-321-A",
      "ChillerSensorNumber": 1,
      "Temperature": 5
   }
]
```

You can see that the results are an array of JSON objects; therefore, the document is enclosed in square brackets. Each JSON object (representing a single row in the table) is enclosed in braces and separated by commas. Within each JSON object, comma separated name/value pairs represent each column in the tabular representation of the results.

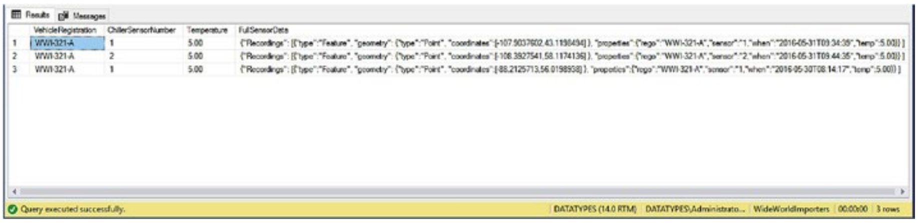
The Warehouse.VehicleTemperatures table, in the WideWorldImporters database also includes a column with the JSON data type, which records the full sensor data. Consider the query in Listing 6-5.

Listing 6-5. Vehicle Temperatures with Full Sensor Data

```
USE WideWorldImporters
GO

SELECT TOP 3
          VehicleRegistration
        , ChillerSensorNumber
        , Temperature
        , FullSensorData
FROM Warehouse.VehicleTemperatures
ORDER BY Temperature DESC ;
```

This query returns the results displayed in Figure 6-2.



	VehicleRegistration	ChillerSensorNumber	Temperature	FullSensorData
1	WWI-321-A	1	5.00	{"Recordings": [{"type":"Feature", "geometry": {"type":"Point", "coordinates":[-107.9037602,43.1198494] }, "properties":{"rego":"WWI-321-A","sensor":"1,"when":"2016-05-31T09:34:39","temp":5.00}}]
2	WWI-321-A	2	5.00	{"Recordings": [{"type":"Feature", "geometry": {"type":"Point", "coordinates":[-108.3927541,58.1174136] }, "properties":{"rego":"WWI-321-A","sensor":"2,"when":"2016-05-31T09:44:35","temp":5.00}}]
3	WWI-321-A	1	5.00	{"Recordings": [{"type":"Feature", "geometry": {"type":"Point", "coordinates":[-88.2125713,56.0198938] }, "properties":{"rego":"WWI-321-A","sensor":"1,"when":"2016-05-30T08:14:17","temp":5.00}}]

Query executed successfully. | DATATYPES (14.0 RTM) | DATATYPES\Administrato... | WideWorldImporters | 00:00:00 | 3 rows

Figure 6-2. *Results of temperatures with full sensor data*

If we were to represent this result set as a JSON document, we would have an array of JSON objects in which one of the objects is a nested JSON object, as shown in Listing 6-6.

Listing 6-6. Vehicle Temperatures with Full Sensor Data Expressed As JSON

```
[
   {
      "VehicleRegistration": "WWI-321-A",
      "ChillerSensorNumber": 1,
      "Temperature": 5,
      "FullSensorData": "{\"Recordings\": [{\"type\":\"Feature\",
      \"geometry\": {\"type\":\"Point\",
      \"coordinates\":[-107.9037602,43.1198494] },
      \"properties\":{\"rego\":\"WWI-321-A\",
      \"sensor\":\"1,\"when\":\"2016-05-31T09:34:39\",
      \"temp\":5.00}} ]"
   },
   {
      "VehicleRegistration": "WWI-321-A",
      "ChillerSensorNumber": 2,
      "Temperature": 5,
      "FullSensorData": "{\"Recordings\": [{\"type\":\"Feature\",
      \"geometry\": {\"type\":\"Point\",
      \"coordinates\":[-108.3927541,58.1174136] },
      \"properties\":{\"rego\":\"WWI-321-A\",
      \"sensor\":\"2,\"when\":\"2016-05-31T09:44:35\",
      \"temp\":5.00}} ]"
   },
   {
      "VehicleRegistration": "WWI-321-A",
      "ChillerSensorNumber": 1,
      "Temperature": 5,
      "FullSensorData": "{\"Recordings\": [{\"type\":\"Feature\",
      \"geometry\": {\"type\":\"Point\",
      \"coordinates\":[-88.2125713,56.0198938] },
      \"properties\":{\"rego\":\"WWI-321-A\",
      \"sensor\":\"1,\"when\":\"2016-05-30T08:14:17\",
      \"temp\":5.00}} ]"
   }
]
```

Tip The nested JSON objects contain a backslash before each double quote, as an escape character.

You will notice, in this example, that the value for each `FullSensorData` node is a JSON object nested inside the JSON object, which represents a row within the tabular result set.

A root node can also be added to a JSON document, sometimes used to represent the name of the object's type or abstraction. This can help give the document context. Listing 6-7 shows the same document as Listing 6-6, but with a root node added.

Listing 6-7. Adding a Root Node

```
{
   "VehicleTemperatures": [
      {
         "VehicleRegistration": "WWI-321-A",
         "ChillerSensorNumber": 1,
         "Temperature": 5,
         "FullSensorData": "{\"Recordings\": [{\"type\":\"Feature\",
         \"geometry\": {\"type\":\"Point\",
         \"coordinates\":[-107.9037602,43.1198494] },
         \"properties\":{\"rego\":\"WWI-321-A\",
         \"sensor\":\"1,\"when\":\"2016-05-31T09:34:39\",
         \"temp\":5.00}} ]"
      },
      {
         "VehicleRegistration": "WWI-321-A",
         "ChillerSensorNumber": 2,
         "Temperature": 5,
         "FullSensorData": "{\"Recordings\": [{\"type\":\"Feature\",
         \"geometry\": {\"type\":\"Point\",
         \"coordinates\":[-108.3927541,58.1174136] },
         \"properties\":{\"rego\":\"WWI-321-A\",
         \"sensor\":\"2,\"when\":\"2016-05-31T09:44:35\",
         \"temp\":5.00}} ]"
      },
      {
         "VehicleRegistration": "WWI-321-A",
         "ChillerSensorNumber": 1,
         "Temperature": 5,
         "FullSensorData": "{\"Recordings\": [{\"type\":\"Feature\",
         \"geometry\": {\"type\":\"Point\",
         \"coordinates\":[-88.2125713,56.0198938] },
         \"properties\":{\"rego\":\"WWI-321-A\",
         \"sensor\":\"1,\"when\":\"2016-05-30T08:14:17\",
         \"temp\":5.00}} ]"
      }
   ]
}
```

JSON vs. XML

Let's compare a simple JSON document against an XML equivalent and examine the differences. Consider the XML document in Listing 6-8, which represents the salespeople within the WideWorldImporters database.

Listing 6-8. Sales People—XML

```
<SalesPeople>
  <SalesPerson>
    <PersonID>2</PersonID>
    <FullName>Kayla Woodcock</FullName>
    <PreferredName>Kayla</PreferredName>
    <LogonName>kaylaw@wideworldimporters.com</LogonName>
    <PhoneNumber>(415) 555-0102</PhoneNumber>
    <EmailAddress>kaylaw@wideworldimporters.com</EmailAddress>
  </SalesPerson>
  <SalesPerson>
    <PersonID>3</PersonID>
    <FullName>Hudson Onslow</FullName>
    <PreferredName>Hudson</PreferredName>
    <LogonName>hudsono@wideworldimporters.com</LogonName>
    <PhoneNumber>(415) 555-0102</PhoneNumber>
    <EmailAddress>hudsono@wideworldimporters.com</EmailAddress>
  </SalesPerson>
  <SalesPerson>
    <PersonID>6</PersonID>
    <FullName>Sophia Hinton</FullName>
    <PreferredName>Sophia</PreferredName>
    <LogonName>sophiah@wideworldimporters.com</LogonName>
    <PhoneNumber>(415) 555-0102</PhoneNumber>
    <EmailAddress>sophiah@wideworldimporters.com</EmailAddress>
  </SalesPerson>
  <SalesPerson>
    <PersonID>7</PersonID>
    <FullName>Amy Trefl</FullName>
    <PreferredName>Amy</PreferredName>
    <LogonName>amyt@wideworldimporters.com</LogonName>
    <PhoneNumber>(415) 555-0102</PhoneNumber>
    <EmailAddress>amyt@wideworldimporters.com</EmailAddress>
  </SalesPerson>
  <SalesPerson>
    <PersonID>8</PersonID>
    <FullName>Anthony Grosse</FullName>
    <PreferredName>Anthony</PreferredName>
    <LogonName>anthonyg@wideworldimporters.com</LogonName>
    <PhoneNumber>(415) 555-0102</PhoneNumber>
    <EmailAddress>anthonyg@wideworldimporters.com</EmailAddress>
  </SalesPerson>
  <SalesPerson>
    <PersonID>13</PersonID>
    <FullName>Hudson Hollinworth</FullName>
    <PreferredName>Hudson</PreferredName>
    <LogonName>hudsonh@wideworldimporters.com</LogonName>
    <PhoneNumber>(415) 555-0102</PhoneNumber>
    <EmailAddress>hudsonh@wideworldimporters.com</EmailAddress>
  </SalesPerson>
  <SalesPerson>
    <PersonID>14</PersonID>
    <FullName>Lily Code</FullName>
    <PreferredName>Lily</PreferredName>
    <LogonName>lilyc@wideworldimporters.com</LogonName>
    <PhoneNumber>(415) 555-0102</PhoneNumber>
    <EmailAddress>lilyc@wideworldimporters.com</EmailAddress>
  </SalesPerson>
  <SalesPerson>
    <PersonID>15</PersonID>
    <FullName>Taj Shand</FullName>
    <PreferredName>Taj</PreferredName>
    <LogonName>tajs@wideworldimporters.com</LogonName>
    <PhoneNumber>(415) 555-0102</PhoneNumber>
    <EmailAddress>tajs@wideworldimporters.com</EmailAddress>
  </SalesPerson>
  <SalesPerson>
    <PersonID>16</PersonID>
    <FullName>Archer Lamble</FullName>
    <PreferredName>Archer</PreferredName>
    <LogonName>archerl@wideworldimporters.com</LogonName>
    <PhoneNumber>(415) 555-0102</PhoneNumber>
    <EmailAddress>archerl@wideworldimporters.com</EmailAddress>
  </SalesPerson>
  <SalesPerson>
```

```
    <PersonID>20</PersonID>
    <FullName>Jack Potter</FullName>
    <PreferredName>Jack</PreferredName>
    <LogonName>jackp@wideworldimporters.com</LogonName>
    <PhoneNumber>(415) 555-0102</PhoneNumber>
    <EmailAddress>jackp@wideworldimporters.com</EmailAddress>
  </SalesPerson>
</SalesPeople>
```

Each salesperson has an opening and closing tag element, containing an opening and closing tag element, for each property related to a salesperson. A root node, called `SalesPeople`, has also been added.

Note This XML document is element-centric. We could, of course, also represent the salespeoples' properties as attributes. Please see Chapter 3, for further details.

Let's compare this XML to the JSON document in Listing 6-9.

Listing 6-9. Sales People—JSON

```
{
   "SalesPeople": [
      {
         "PersonID": 2,
         "FullName": "Kayla Woodcock",
         "PreferredName": "Kayla",
         "LogonName": "kaylaw@wideworldimporters.com",
         "PhoneNumber": "(415) 555-0102",
         "EmailAddress": "kaylaw@wideworldimporters.com"
      },
      {
         "PersonID": 3,
         "FullName": "Hudson Onslow",
         "PreferredName": "Hudson",
         "LogonName": "hudsono@wideworldimporters.com",
         "PhoneNumber": "(415) 555-0102",
         "EmailAddress": "hudsono@wideworldimporters.com"
      },
      {
         "PersonID": 6,
         "FullName": "Sophia Hinton",
         "PreferredName": "Sophia",
         "LogonName": "sophiah@wideworldimporters.com",
         "PhoneNumber": "(415) 555-0102",
         "EmailAddress": "sophiah@wideworldimporters.com"
      },
      {
         "PersonID": 7,
         "FullName": "Amy Trefl",
         "PreferredName": "Amy",
         "LogonName": "amyt@wideworldimporters.com",
         "PhoneNumber": "(415) 555-0102",
         "EmailAddress": "amyt@wideworldimporters.com"
      },
      {
         "PersonID": 8,
         "FullName": "Anthony Grosse",
         "PreferredName": "Anthony",
         "LogonName": "anthonyg@wideworldimporters.com",
         "PhoneNumber": "(415) 555-0102",
         "EmailAddress": "anthonyg@wideworldimporters.com"
      },
      {
         "PersonID": 13,
         "FullName": "Hudson Hollinworth",
         "PreferredName": "Hudson",
         "LogonName": "hudsonh@wideworldimporters.com",
         "PhoneNumber": "(415) 555-0102",
         "EmailAddress": "hudsonh@wideworldimporters.com"
      },
      {
         "PersonID": 14,
         "FullName": "Lily Code",
         "PreferredName": "Lily",
         "LogonName": "lilyc@wideworldimporters.com",
         "PhoneNumber": "(415) 555-0102",
         "EmailAddress": "lilyc@wideworldimporters.com"
      },
      {
         "PersonID": 15,
         "FullName": "Taj Shand",
         "PreferredName": "Taj",
         "LogonName": "tajs@wideworldimporters.com",
         "PhoneNumber": "(415) 555-0102",
         "EmailAddress": "tajs@wideworldimporters.com"
      },
      {
         "PersonID": 16,
         "FullName": "Archer Lamble",
         "PreferredName": "Archer",
         "LogonName": "archerl@wideworldimporters.com",
         "PhoneNumber": "(415) 555-0102",
         "EmailAddress": "archerl@wideworldimporters.com"
      },
      {
         "PersonID": 20,
         "FullName": "Jack Potter",
         "PreferredName": "Jack",
         "LogonName": "jackp@wideworldimporters.com",
         "PhoneNumber": "(415) 555-0102",
         "EmailAddress": "jackp@wideworldimporters.com"
      }
   ]
}
```

Instead of using elements, as in the XML document, the JSON document consists simply of name/value pairs in an array of JSON objects. A root node, called SalesPeople, has also been added.

The most obvious observation about the JSON document is that the character count is much shorter, largely due to the lack of closing tags. The main consequence of this is that the document is easier to parse, and there is less information to be transferred between application tiers. Arguably, the document is also more human-readable. These advantages have made JSON a very popular choice with application developers.

The main disadvantage of the JSON document is that it cannot be bound to a schema in the way that XML can. The impact of this is that although the document can be parsed, to ensure that it has valid syntax, it isn't possible (without custom code) to ensure that it meets the contract expected by the recipient before sending.

Other than the differences mentioned, despite their different appearance, the documents are actually quite similar. They are both self-describing, extensible documents that can be used for data-interchange. Both formats are widely used and work with many REST APIs and web service end points.

Tip REST is short for representational state transfer. It is an architectural style of providing a uniform interface, often between layers of an application. It provides a stateless approach, with client/server separation.

JSON Usage Scenarios

There are many use cases for JSON data within SQL Server. The following sections will introduce some of these potential uses.

n-Tier Applications with Rest APIs

Modern apps often have a lot of logic at the client side. The application tier of the application often has to have complex code, or even multiple sublayers, to broker a conversation between the client and the back-end RDBMS. This is because you will have to use an object relational mapper to execute the query against the database, write these results into data transfer object, and then serialize the results in JSON format before they can be sent to the client.

With JSON support in SQL Server, however, you can simply expose the data from SQL Server to a REST API and return the data in JSON format, meaning that the application tier can simply send the data as is to the client. While there may be resistance to this approach from middle-tier purists, it certainly allows architects to simplify the application design.

De-Normalizing Data

Using a normalized data model is perfect when high-frequency updates are made to data. In a normalized model, data is separated into multiple tables, which are joined together using primary and foreign key constraints, with the intention of storing data only once. For example, if customers have multiple addresses, then core details about a customer, such as name and phone number, may be stored in a table called Customers. Their addresses may then be stored in a separate table called CustomerAddresses, which contains a foreign key (CustomerID), which joins to the primary key of the Customers table. This means that the core details about the customer are only stored once, as opposed to having to repeat the details for every address.

Tip While a detailed discussion of normalization is beyond the scope of this book, a full discussion can be found in *Expert Scripting and Automation for SQL Server DBAs* (Apress, 2016).

A traditional normalized model can cause issues in some instances, however. For example, performance can decrease when data is split across multiple tables and joined together in a `SELECT` statement, owing to the matching of primary and foreign key values that is required. Also, when data is updated across multiple tables, a transaction must be used, to ensure a consistent update. This can lead to locking issues, in which pessimistic isolation levels are in use, or IO performance issues, in which optimistic isolation levels are used.

Tip A full discussion of transaction isolation levels can be found in *Pro SQL Server Administration* (Apress, 2015).

To work around this issue, data architects will sometimes use NoSQL structures to store details of entities such as customers, so that logical entities can be stored as a single record. Coincidently, JSON is often the format used for these records. This approach creates its own issues, however, when the NoSQL data must be combined with data that is still stored in a relational format.

JSON can help resolve these kinds of modeling challenges, by allowing the JSON record to be stored in a table in SQL Server, meaning that updates can be made to a single table, while the table can still easily be joined back to relational data.

Config As Code

In DevOps environments, there is a requirement to have infrastructure as code, platforms as code, and config as code. Essentially, an entire virtual estate will be written in code, so that it is highly portable between data centers or, more commonly, between data centers and the cloud. It is also highly recoverable, in the event of a disaster, such as the loss of a data center.

SQL Server management has been slow to be incorporated into the DevOps space, because there is a lack of crossover skills between SQL Server and desired state configuration tooling, such as Chef and Puppet. The DBA world is slowly starting to get on board, however, and as part of an SQL-Server-platform-as-code approach, a configuration management database (CMDB) will often be used on a central management server, to store the details of member servers (SQL Server VMs) within the estate. A central management server (in this context) refers to an instance of SQL Server that is used to help DBAs manage the rest of the SQL Server estate. It will often be the master server in SQL Server Agent master/target job configurations and may have other management features installed, such as Management Data Warehouse, which is used as a central hub for SQL Server monitoring.

When a platform-as-code approach is being used, however, a config-as-code approach should be applied alongside. In the SQL Server world, this involves being able to rebuild the CMDB from code, which is stored in a source control provider, such as GitHub of TFS.

Config as code for the SQL Server CMBD can easily be managed with a circular process. The first step in this process is a Server Agent job, which will periodically run and export the data from the tables into JSON files in the operating system. These files will then be pushed to a source-control repository and checked in.

When a desired state configuration management tool such as Chef or Puppet builds a new central management server, it will look, in the source control repository, to find the JSON files containing the configuration data. It will create the config database and then repopulate the tables with the data from the repo. This provides an easily configurable RPO (recover point objective) for the config database, without the challenges that are often associated with enterprise backup management tools and the lead times often associated with recovering data from tape robots. It also means that the core management utility for SQL Server can be easily moved to the cloud or other data centers, helping to make the SQL Server estate extremely portable.

Tip The principle of desired state management tools, such as Puppet and Chef, is that they run periodically on a server, with a manifest that describes the desired state of the server. At the Windows level, this may include code to disable the guest account or ensure that user rights assignments are configured correctly. At the SQL Server level, it may check that a specific login exists or that `xp_cmdshell` is disabled. First, the tool will check to see if a resource is already configured as expected. If not, it will correct the configuration. This means that if an unauthorized change is made to a server, it will be corrected the next time the manifest is applied. The result is that Windows engineers and DBAs can also be sure of the state of their servers.

Analyzing the Log Data

Devices such as sensors or RFID (radio frequency identification) can generate very large amounts of data. This means that data architects will often choose to store this data in NoSQL solutions. Often, JSON is used as the file format for such logging.

When large logs are stored in JSON format, the SQL Server's native JSON support means that these logs can easily be read into SQL Server and analyzed using T-SQL, without any complex parsing requirements. This can reduce the time to market for reports against log data.

Summary

JSON is a lightweight data-interchange format that is supported by many REST APIs and web service end points. It is similar to XML, in that it is an extensible, self-describing, hierarchical document format, but it differs in the following ways:

- It cannot be validated against a schema.
- It does not have closing tags.
- It is shorter.
- It is easier to parse.
- It supports arrays.

JSON's basic format is a series of name/value pairs, which are separated by commas. JSON objects are always enclosed in braces, and arrays are always enclosed in brackets. JSON objects can be nested inside other JSON objects. When this is the case, a backslash should be used to escape characters, such as double quotes and braces.

JSON has many use cases in SQL Server, including the simplification of REST APIs interacting with back-end databases. JSON is also a good choice when NoSQL solutions must be integrated with SQL Server, such as when device logs must be analyzed or NoSQL semi-structured data must be stored alongside structured data. JSON is also very useful when DBAs or platform engineers implement config as code solutions.