Article
Stefan Wittmann · May 31, 2016 12m read

Introducing the Document Data Model in Caché 2016.2

Introduction

The field test of Caché 2016.2 has been available for quite some time and I would like to focus on one of the substantial features that is new in this version: the document data model. This model is a natural addition to the multiple ways we support for handling data including Objects, Tables and Multidimensional arrays. It makes the platform more flexible and suitable for even more use cases.

Let ' s start with some context. You probably are aware of at least one database system that comes under the cover of the NoSQL movement. There are quite a lot and you can group them into a few categories. Key/Value is a pretty simple data model. You can store a value in the database and associate it with a key. If you want to retrieve the value, you have to access it via the key. Choosing a good key can make a huge difference as it often defines the sorting and can be used for simple aggregation if you want to group by something that is part of the key. But the value is just a value. You cannot access specific sub-elements within the value or index them. If you want to do more with the value you have to write application logic. Key/Value is a good fit if you have to work with a huge data set and very simple values, but it has diminishing value if you have to work with more complex records.

A document data model is very similar to Key/Value, but the value can be more complex. Values are still associated with a key, but in addition, you can access sub-elements of the value and index specific elements. This also implies that you can search for specific documents where some sub-elements meet a restriction. Obviously, there are more models in the NoSQL world like Graph, but we will keep the focus on documents for now.

What exactly is a document?

Let ' s get one thing out of the way right at the beginning as it tends to confuse some readers: When I am talking about a document in this post I don ' t mean a physical document, like a PDF file or a Word document.
A document in our context is a structure that allows sub values to be associated with a specific path. There are various well-known serialization formats that can describe a document, e.g. JSON and XML. They usually have the following structures and data types in common:

1. A structure of unordered key/value pair
2. An ordered list of values
3. A scalar value

The first maps to an attributed element in XML and an object in JSON. The second structure is introduced by a listing with sub-elements in XML and an array in JSON. The third just allows native data types like strings, numbers, and Booleans.

It is common to visualize a document in serialized form like JSON, just keep in mind that this is just one potential way of representing the document. For this post, I will use JSON as the primary serialization format and this fits well with our improved JSON support, which is described here in case you have missed this.

Documents are grouped in collections. Documents that have semantics and potentially structure in common should be stored in the same collection. Collections can be created on the fly and don ' t require any schema information upfront.

In order to access collections, you have to retrieve a database handle first. The database handle serves as your connection to the server and provides simple access to collections, but also handles more complex scenarios in the case of a distributed environment.

The basics

Let ' s take a first look how you insert a simple document in Caché Object Script:

```
USER>set db = ##class(%DataModel.Document.Database).$getDatabase()

USER>set superheroes = db.$getCollection("superheroes")

USER>set hero1 = {"name":"Superman","specialPower":"laser eyes"}

USER>set hero2 = {"name":"Hulk","specialPower":"super strong"}

USER>do superheroes.$insert(hero1)

USER>do superheroes.$insert(hero2)

USER>write superheroes.$size()
2
```

The above code sample retrieves the database handle first and gets a collection named " superheroes " . Collections are created implicitly, so you don ' t have to set it up before. After we have access to our new collection, we create two very simple documents representing the heroes Superman and Hulk. Both are stored in the collection with a call to $insert(<document>) and a final check for the collection size reports two documents, as the collection didn ' t exist before.

The $insert() call returns the inserted document on success. This allows you to retrieve the auto-assigned ID in case you want to continue working with the document. This also allows chaining of methods:

```
USER>set hero3 = {"name":"AntMan","specialPower":"can shrink and become super strong"
}

USER>write superheroes.$insert(hero3).$getDocumentID()
3
```

This code snippet creates another hero object and persists it in our superheroes collection. This time, we chain the method call $getDocumentID() to the $insert() call and retrieve the ID the system did assign to this document. $insert() will always assign IDs automatically. If you have the requirement to assign your own ID ' s, you can make use of the $insertAt(<User-ID>,<document>) call.

When you want to retrieve a document with a specific ID, you can call the $get(<ID>) method on a collection:

```
USER>set antMan = superHeroes.$get(3)

USER>write antMan.$toJSON()
{"name":"AntMan","specialPower":"can shrink and become super strong"}
```

Assume we want to update the document representing Superman with his hometown. We can easily update an existing document with the $upsert(<ID>,<document>) call:

```
USER>set hero1.hometown = "Metropolis"

USER>do superheroes.$upsert(1,hero1)

USER>write superheroes.$get(1).$toJSON()
{"name":"Superman","specialPower":"laser eyes","hometown":"Metropolis"}
```

$upsert() will insert a document if the ID is not currently taken, or update the existing document in the other case. Of course, you can also serialize the complete content of a collection to JSON by just calling $toJSON() on it:

```
USER>write superheroes.$toJSON()
[
{"documentID":1,"documentVersion":4,"content":{"name":"Superman","specialPower":"lase
r eyes","hometown":"Metropolis"}},
{"documentID":2,"documentVersion":2,"content":{"name":"Hulk","specialPower":"super st
rong"}},
{"documentID":3,"documentVersion":3,"content":{"name":"AntMan","specialPower":"can sh
rink and become super strong"}}
]
```

You can observe, that the collection is represented as an array of documents. Each document is wrapped with its document ID and the document version, which is used for properly handling concurrency. The actual document content is stored in the property content of the wrapper. This representation is required to get a complete snapshot of a collection and be able to move it around.
Also, this covers a very important aspect of our model: We do not reserve and inject special properties to your document data. The information that is required by the engine to properly handle a document is stored outside of the document itself. In addition, a document can be either an object or an array. Many other document stores only allow objects as top-level elements.

There are many more API calls to modify documents in a collection and we have seen some of the basic operations. Inserting and modifying documents is fun, but it is getting more interesting when you actually want to analyze a data set or retrieve documents that meet specific restrictions.

Querying

Every data model needs some querying capabilities to be considered useful. To be able to query documents with a dynamic schema, there are two potential paths:

1. Design and implement a proprietary query language that can cope with the dynamic nature of documents
2. Integrate querying into an established and structured query language

For a couple of reasons I will discuss at a later point in this post, we decided to expose collections to our SQL engine. The good news about this is that you can continue to leverage your knowledge of SQL and we are cooking yet another flavor of a query dialect. Actually, the SQL ANSI committee has proposed standard extensions for JSON, which we are complying to. In a nutshell, these extensions include two categories of JSON functions:

1. A set of functions to publish JSON content from relation content
2. A set of functions to query dynamic JSON content

For the scope of this article, we will only cover category two, querying dynamic JSON content and make the result available as a table, so that it can be processed by SQL.

The magic function that allows you to expose dynamic content (content without an associated schema) and make it

available to SQL, which operates on data with a predefined schema, is called JSON_TABLE . In general, this function takes two arguments:

1. A JSON data source
2. A definition that specifies a mapping of JSON paths to columns with names and types

Let ' s take a look at an example to put some flesh on the bones:

```
SELECT name, power FROM JSON_TABLE(
  'superheroes',
  '$' COLUMNS(
    name VARCHAR(100) PATH '$.name',
    power VARCHAR(300) PATH '$.specialPower'
  )
)
```

```
name        power
---------   ------------------------------
Superman    laser eyes
Hulk        super strong
AntMan      can shrink and become super strong
```

The first argument of the JSON_TABLE function defines the source of the virtual table it creates. In this case, we want to query the collection " superheroes " . Every document in this collection will result in a row.
Remember the second argument defines a mapping that exposes a specific value from a document as a table column. This second argument consists of two parts: As a first step, we set the context for our upcoming expressions. The dollar sign '$' has a special meaning and refers to the root of the document. All expressions that follow are based on this context.
What follows is a COLUMNS clause, a comma separated list of COLUMN expressions. Each COLUMN expression creates a column in the virtual table. We are exposing two columns named " name " and " power " in our query. The column " name " is defined with the type VARCHAR(100), while the column " power " is limited to 300 characters. The PATH expression is tying a specific value of the document with a JPL (JSON Path Language) expression to the column. The value of the key " name " is exposed to the column " name " , while the value of the key " specialPower " is mapped to the column " power " . JPL expressions can be very expressive and powerful, but we will save that for a later discussion. The expressions we used in our sample are very basic.

If this syntax is new to you, it may take a while to sink in. But it helps to read your JSON_TABLE function in a natural way. As an example take our above query. What we are saying is basically this:

I want to query the collection ' superheroes ' and set the context for my expressions at the root of each document. I want to expose two columns:

1. A column " name " with the type VARCHAR(100) and feed it with the value of the key " name "
2. A column " power " with the type VARCHAR(300) and feed it with the value of the key " specialPower "

As mentioned before, JPL expressions get can get complex, or you just want to expose a lot of columns. Therefore, we have built in an extension to the standard allowing you to refer to a type definition, which is basically a predefined COLUMNS clause. This is how you can register the above COLUMNS clause:

```
do db.$createType("heropower",{"columns":[{"column":"name","type":"VARCHAR(100)","pat
h":"$.name"},{"column":"power","type":"VARCHAR(300)","path":"$.specialPower"}]})
```

After registering the type information you can refer to it in the JSON_TABLE function with the %TYPE extension:

```
SELECT name, power FROM JSON_TABLE(
  'superheroes',
  '$' %TYPE 'heropower'
)
```

This obviously helps you to provide a consistent view of your documents to SQL queries and it greatly simplifies the query itself.

Some advanced stuff

There is much more to say about almost every bit we have covered so far, but I would like to focus on the most important pieces for now. While reading through the last section you may have spotted some of the clues that make a very strong point for the JSONTABLE function:

1. It creates a virtual table
2. It can consume JSON-like data as source data

The first bullet point by itself is a big deal, because it allows you to easily query a collection and join it with another JSONTABLE call, or – yes, there it is – a table. Joining collections with tables is a huge benefit; it allows you to chose the perfect data model for your data depending on the requirements.
Need type safety, integrity checks and your model is not evolving a lot? Pick relational. You have to deal with data from other sources and you have to consume the data – no matter what, your model is changing at a rapid pace or you want to store a model that can be influenced by the application user? Pick the document data model. You can rest assured that you can bring your models together with SQL.

The second benefit of the JSONTABLE function is actually unrelated to the underlying data model. What I have shown you so far demonstrated querying of a collection with JSONTABLE . The first argument can also be any valid JSON input. Consider the following example:

```
SELECT name, power FROM JSON_TABLE(
  '[
    {"name":"Thor","specialPower":"smashing hammer"},
    {"name":"Aquaman","specialPower":"can breathe underwater"}
  ]',
  '$' %TYPE 'heropowers'
)

name        power
---------   ------------------------------
Thor        smashing hammer
Aquaman     can breathe underwater
```

The input is a regular JSON string, which represents an array of objects. As the structure matches our superheroes collection, we can reuse our stored type identifier 'heropowers'.
This enables powerful use cases. You can actually query in-memory JSON data without persisting it on-disk. You can request JSON data over a REST call and then query and join it with a collection or a table. With this feature, you can query the Twitter timeline, GitHub repository statistics, stock information or just the weather feed. I will pick this up in a later dedicated post as we have made this very convenient.

REST-enabled

The document data model comes with a REST enabled interface out of the box. All CRUD (Create, Read, Update, Delete) and query capabilities are available over HTTP. I am not going to cover the complete API, but here is a

sample cURL command that retrieves all documents of the collection “ superheroes ” in the namespace USER:

```
curl -X GET -H "Accept: application/json" -H "Cache-Control: no-
cache" http://localhost:57774/api/document/v1/user/superheroes
```

Is this for me?

The document data model is a significant addition to our platform. It represents a completely new model, next to Objects and Tables. The fact that it integrates nicely with SQL makes it easy to leverage it in existing applications. The new JSON capabilities introduced in Caché 2016.1 make JSON handling in Caché fun and simple.

Still, this is a new model. You have to understand when and why you are using it. As always, pick the right tool for a given task.

This data model excels when you have to deal with dynamic data. Here are the major technical benefits in a nutshell:

- Flexibility and Ease-of-Use

Schemas don ’ t have to be defined upfront and therefore, you can quickly setup work environments for your data and easily adapt to changes in a data structure.

- Sparseness

Remember that table with 300 columns, but each row just populates 15 of those? That is a sparse data set and relational systems are not optimal in handling them. Documents are sparse by design and can be stored and handled efficiently.

- Hierarchical

Structured types, such as arrays and objects can be nested arbitrarily deep. That means you can store related data within a document, potentially reducing I/O for reads when you need access to that record anyway. Data can be stored de-normalized, whereas data is stored normalized in a relational model.

- Dynamic Types

A particular key does not have a fixed data type like a column does. A name can be a string in one document, but a complex object in the next. Make simple things simple. You can always make them more complex – or simplify them again.

Each of these bullet points is important and a good use case requires at least one of them, but it is not uncommon to match all of them.

Assume you are building a backend for mobile applications. The clients (end-users) can update at their will, so you have to support multiple versions of your interface at the same time. Developing by contract, e.g. with WebServices can reduce your ability to adapt your data interface quickly (but makes it potentially more stable). The flexibility of the document data model allows you to rapidly evolve your schema, handle multiple versions of a specific record type and still correlate them in queries.

Further resources

If you want to learn more about this exciting new feature grab one of the available field test versions 2016.2 or 2016.3.

Make sure to take a look at the Document Data Model Learning Path:
https://beta.learning.intersystems.com/course/view.php?id=9

Don ' t miss the opportunity to watch the session from this year ' s Global Summit. The Data Modeling Resource Guide groups all relevant sessions together:
https://beta.learning.intersystems.com/course/view.php?id=106

And finally, engage at the developer community, ask questions and let us know your feedback.

#Caché #Data Model #Document Data Model (NoSQL) #JSON #SQL