With JSON in excellent form

JSON is a very popular data format, because it is easy readable for humans and can be easily parsed by software. It is a data format often used to transfer data in APIs over the Internet. The MBS FileMaker Plugin supplies you with a lot of JSON functions, optimized for performance and usability.

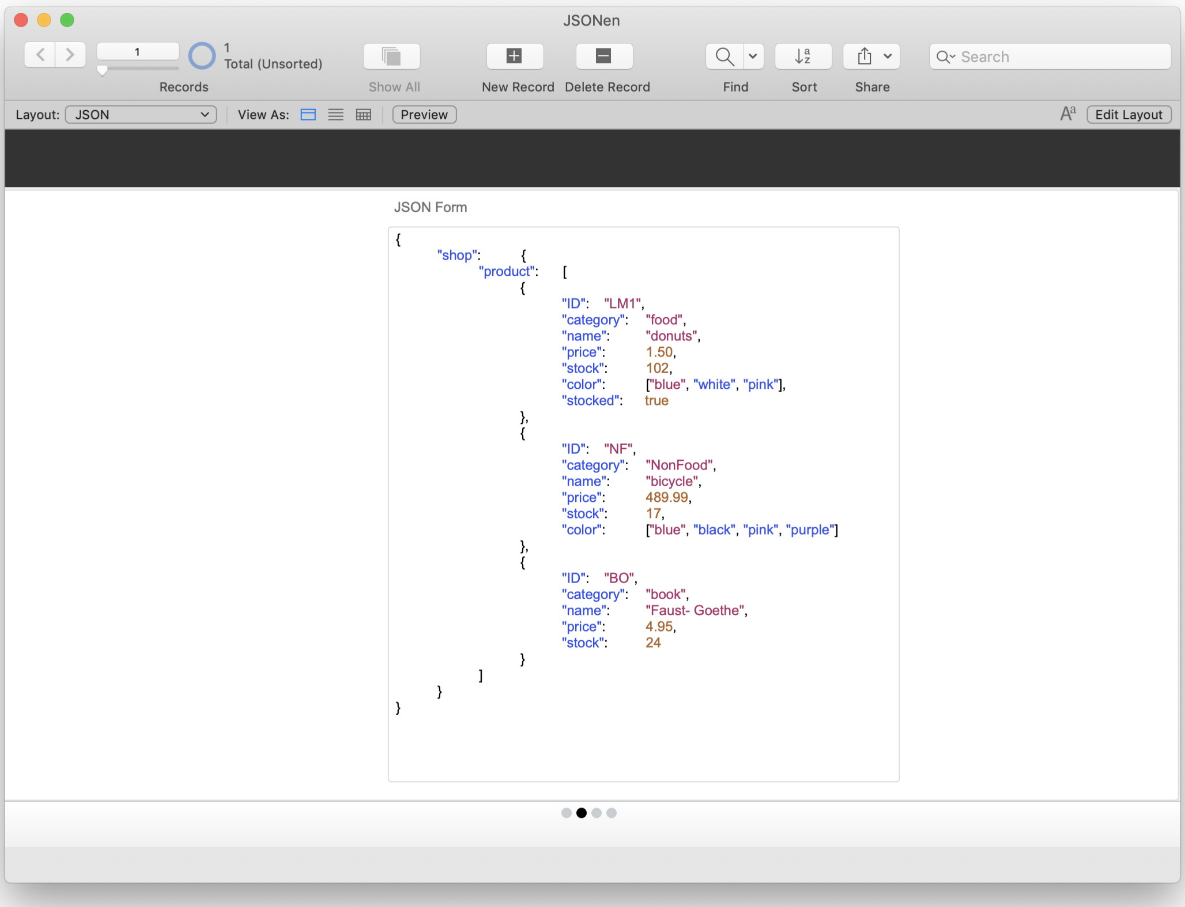
JSON Format

The **J**ava**S**cript **O**bject **N**otation is a compact data format that is easy readable for humans. A JSON text consists of objects written in brackets and arrays (lists). Objects are in curly brackets and can contain several elements separated by comma. An element has a key and a value. They are separate by colons. The value can have different datatypes: null, true and false, numbers (with a sequence of digests from 0-9, positiv and negativ numbers, decimal numbers and exponent E), strings, JSON objects or arrays. Arrays are lists that contains elements. This elements can be objects or one of a forenamed datatypes. The elements of the array are in brackets: e.g. ["grape", -123, true, null]. The typical structure of a JSON you can see in the follow example:

```
{
        "shop": {
                "product":      [
                        {
                                "ID":     "LM1",
                                "category":       "food",
                                "name":"donuts",
                                "price": 1.50,
                                "stock": 102,
                                "color": ["blue", "white", "pink"]
                                „stocked":        true
                        },
                        {
                                "ID":             "NF",
                                "category":       "NonFood",
                                "name":"bicycle",
                                "price": 489.99,
                                "stock": 17,
                                "color": ["blue", "black", "pink", "purple"]
                        },
                ]
        }
}
```

Validating the JSON and format it

If we get a JSON it is usually unformatted and the direct reading can be confusing for you as developer. With the function „JSON.Format“ in combination with „JSON.Colorize“ you can format and color the JSON text to understand the data structure much easier.



```
{
        "shop":      {
                "product":    [
                {
                        "ID":   "LM1",
                        "category":   "food",
                        "name":       "donuts",
                        "price":      1.50,
                        "stock":      102,
                        "color":      ["blue", "white", "pink"],
                        "stocked":    true
                },
                {
                        "ID":   "NF",
                        "category":   "NonFood",
                        "name":       "bicycle",
                        "price":      489.99,
                        "stock":      17,
                        "color":      ["blue", "black", "pink", "purple"]
                },
                {
                        "ID":   "BO",
                        "category":   "book",
                        "name":       "Faust- Goethe",
                        "price":      4.95,
                        "stock":      24
                }
        ]
    }
}
```

If you want to write a JSON by your own, you have different possibilities to do this. You can enter a JSON as a text and remove needles blankets with „JSON.Compact“ function.

Attention: Strings are in quotes in JSON. If you want to write a quote in FileMaker you must escape the quotes with a prefix backslash (\"), else you get an error. Some special tip for Mac users: The escaping in huge texts can take lots of time, because of that you can use the search and replace function that the MBS Plugin provides you for Mac. You can search for " and replace them with \" using our find bar, saving you some time. But you should pay attention on the quotes that should't be escape and change them back manually.

It is advisable, to check a JSON for errors and validate the JSON. For that you have the „JSON.IsValid“ function. Here we parse the JSON, but don't return an error message, but just 0 for an error and 1 for okay.
The script can look like this:

```
Set Variable [ $JSON ; Value: MBS( "JSON.Compact"; "{ \"shop\": { \"product\": [
{ \"ID\": \"LM1\",  \"category\":\"food\", \"name\":\"donuts\", \"price\": 1.50,
\"stock\": 102, \"color\": [\"blue\", \"white\", \"pink\"], \"stocked\": true },
{\“ID\": \“NF\“, \"category\":\"NonFood\", \"name\":  \“bicycle\", \“price\":
489.99,\“stock\":17, \"color\":[\"blue\", \"black\", \"pink\", \“purple\"] },
{ \"ID\": \“BO\", \"category\": \“book\", \"name\": \"Faust- Goethe\",
\"price\": 4.95, \"stock\": 24 } ] } }“)

Set Variable [ $r ; Value: MBS( "JSON.IsValid"; $JSON ) ]
If [ $r ≠ 1 ]
      Show Custom Dialog [ "Error occurred " ; "The JSON is not valide" ]
```

```
Else
      Show Custom Dialog [ "congratulations" ; "Your JSON is valide!" ]
End If

Set Field [ JSON::JSON ; $JSON ]

Set Variable [ $JSONForm ; Value: MBS( "JSON.Format"; $JSON ) ]
Set Variable [ $JSONForm ; Value: MBS( "JSON.Colorize"; $JSONForm ) ]
```

Insert arrays, elements and objects into an existing JSON

If we want to add an object to a JSON, at first we create an empty object with „JSON.CreateObject“. We want to fill this object with values, so we use the „JSON.AddStringToObject“ function multiply times, which creates JSON text values for us automatically:

```
Set Variable [ $NewObject ; Value: MBS( "JSON.CreateObject" ) ]
Set Variable [ $NewObject ; Value: MBS( "JSON.AddStringToObject"; $NewObject;
"ID"; "BU 2" ) ]
Set Variable [ $NewObject ; Value: MBS( "JSON.AddStringToObject"; $NewObject;
"category"; "book" ) ]
Set Variable [ $NewObject ; Value: MBS( "JSON.AddStringToObject"; $NewObject;
"name"; "Wilhelm Tell- Schiller" ) ]
Set Variable [ $NewObject ; Value: MBS( "JSON.AddStringToObject"; $NewObject;
"price"; "5.60" ) ]
Set Variable [ $NewObject ; Value: MBS( "JSON.AddStringToObject"; $NewObject;
"stock"; "5" ) ]
```

We append this object to the product array by using the "JSON.SetPathItem" function. You pass a path to locate the JSON item to change within a JSON data structure. New nodes may be created on the way. For appending something to an array, we use the keyword append in the path:

```
Set Variable [ $NewJSON ; Value: MBS( "JSON.SetPathItem";
     $JSON;"shop¶product¶append"; $newObject ) ]
```

We can add other elements to our object. For example we can add elements that have a key, but not assigned a value yet. We use in this case the „JSON.AddNullToObject“ function. It is also possible to add numbers („JSON.AddNumberToObject“), boolean values („JSON.AddBooleanToObject“, „JSON.AddTrueToObject“, „JSON.AddFalseToObject“) and other elements („JSON.AddItemToObject“) to an object. This way we can add arrays or sub objects into an object as value. To construct a new array from scratch, we create a new and empty array and append a string, a number or an other object to this array:

```
Set Variable [ $Array ; Value: MBS( "JSON.CreateArray" ) ]
Set Variable [ $Array ; Value: MBS( "JSON.AddStringToArray"; $Array; "Friedrich"
) ]
Set Variable [ $Array ; Value: MBS( "JSON.AddNumberToArray"; $Array; 1759 ) ]
Set Variable [ $Array ; Value: MBS( "JSON.AddItemToArray"; $Array;
"{\"placeOfBirth\":\"Marbach am Neckar\", \"placeOfDeath\":\"Weimar\"}" ) ]
Set Variable [ $NewObject ; Value: MBS( "JSON.AddItemToObject"; $NewObject;
"Info"; $Array ) ]
```

There is the possibility to pass more than one value directly in the parameters. For example the functions „JSON.CreateDoubleArray“ and „JSON.CreateIntegerArray“ pass any number of parameters and create a new array with this values. With „JSON.CreateStringArray“ you create a new array with any number of string values while „JSON.CreateStringArrayWithList“ takes the values from a list passed in one parameter.

You can also add variant JSON Nodes with certain types e.g. a boolean node which value depends on a condition:

```
Set Variable [ $Array ; Value: MBS("JSON.CreateBoolean"; Length($text) > 0 )]
```

If you use „JSON.CreateNumber“, you can create a number node that is encoded for JSON. For other datatypes you will finde convenient functions. The MBS JSON Functions can handle arbitrarily large numbers and avoid the rounding of 15 digits.
You can clone a JSON, e.g. you can us that, if you have a JSON part that you want use multiply times with a minimum of changes. By using the "JSON.Clone" function, you get a new reference number for a JSON structure in memory that you can use for your next steps.

Delete, search and replace

You can not only add elements, also you can delete elements from an array or an object.

```
Set Variable [ $Delete ; Value: MBS( "JSON.DeleteItemFromObject"; $NewObject;
     "Place" ) ]

Set Variable [ $DeleteArray ; Value: MBS( "JSON.DeleteItemFromArray"; $Array;
     1 ) ]
```

In the first script step we delete an element from the object. We pass the JSON and the name of the element in the parameters. For deleting an element from an array we pass the index of the element instead of the name.

All JSON arrays start with index 0, so passing 1 deletes the second element. Both delete functions return us a new JSON without the elements or when using reference numbers the reference number.

For deleting an object from an array, it would be nice to find the object in the array by key and value. Here you use the "JSON.FindValueInObjectArray" function to get the index of the object. If you want to find a value in an array, you can use the „JSON.FindValueInArray“ function.

Perhaps you only want to change a value. You won't need to delete the value and add it with another value, but rather you can change the value with the „JSON.ReplaceItemInArray“ function. If you want to change an element in an object, you can use the „JSON.ReplaceItemInObject“ function. For this you pass the index for an array and the key for an element in an object and of course the new value.

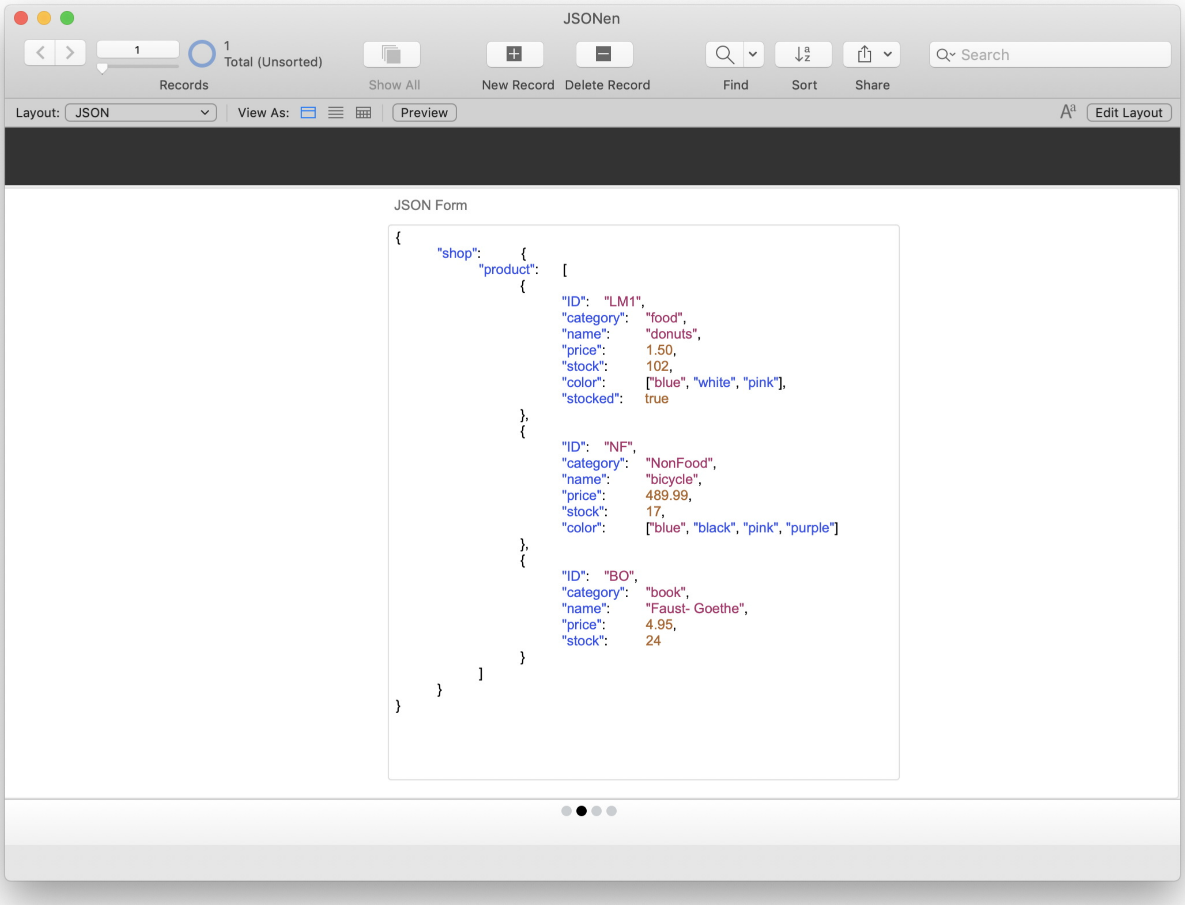
```
Set Variable [ $ObjReplace ; Value: MBS( "JSON.ReplaceItemInObject"; $NewObject;
     "price";"\"3,69\"" ) ]

Set Variable [ $ArrayReplace ; Value: MBS( "JSON.ReplaceItemInArray"; $Array;
     1 ;1805) ]
```

The Get Functions

If we want some values from a JSON, we have some get functions.
For example you can get elements from a partial tree of your JSON. That is very useful for working with a part of the JSON only. We use the “JSON.GetObjectItemTree” function. If we need the first

product from the JSON structure, we pass the way to this element to the parameter list. The first product is the first object in an array named shop. If we invert that way we get the parameter, that we can pass to our function: **„shop“; “product“; 0.** You can use the function“JSON.GetPathItem” in a similar way.



```
{
    "shop":      {
        "product":    [
            {
                "ID":   "LM1",
                "category":   "food",
                "name":       "donuts",
                "price":      1.50,
                "stock":      102,
                "color":      ["blue", "white", "pink"],
                "stocked":    true
            },
            {
                "ID":   "NF",
                "category":   "NonFood",
                "name":       "bicycle",
                "price":      489.99,
                "stock":      17,
                "color":      ["blue", "black", "pink", "purple"]
            },
            {
                "ID":   "BO",
                "category":   "book",
                "name":       "Faust- Goethe",
                "price":      4.95,
                "stock":      24
            }
        ]
    }
}
```

Now we want the number of the elements from the first product and a list of the names of the elements. For this we view the subtree and get the number of the elements with the “JSON.GetObjectSize” function. With “JSON.GetObjectNameList” we get the list with the names of the elements.

```
Set Variable [ $CountElement ; Value: MBS("JSON.GetObjectSize";
     MBS( "JSON.GetObjectItemTree"; $json; "shop";"product";0 )) ]

Set Variable [ $Elemente ; Value: MBS( "JSON.GetObjectNameList";
     MBS( "JSON.GetObjectItemTree"; $json; "shop";"product";0 )) ]
```

For an array it it similar. For example we want the number of elements and the elements of the color array of the first product. We get the number of elements for this array with the “JSON.GetArraySize” function and the elements with "JSON.GetArrayItemsAsList". Beside you can get a single element from the array by the index ("JSON.GetArrayItem").

```
Set Variable [ $ArraySize ; Value: MBS( "JSON.GetArraySize";
     MBS("JSON.GetObjectItemTree";$json;"shop";"product";0;"color") ) ]

Set Variable [ $arraylist ; Value: MBS( "JSON.GetArrayItemsAsList";
     MBS("JSON.GetObjectItemTree";$json;"shop";"product";0;"color") ) ]
```

```
Set Variable [ $ArrayElement ; Value: MBS( „JSON.GetArrayItem";
      MBS("JSON.GetObjectItemTree";$json;"shop";"product";0;"color"); 1 ) ]
```

With the “JSON.GetObjectItem" function you can search for an element by name. It’s return the reference number or a JSON text. The decision of the return format you make in the third parameter: Pass 1 for reference number or 0 for JSON text. This parameter is optional and 0 is the default value.

With the "JSON.GetObjectName" function we can return the name of one element from an object. In the parameters we set the element we want by number. If we want the value of an explicit element, you can use the “JSON.GetValue” function.

```
Set Variable [ $price ; Value: MBS("JSON.GetValue";MBS("JSON.GetObjectItemTree";
     $json;"shop";"product";0;"price")) ]
```

In addition we can get the datatype of a value by the “JSON.GetType” function. If we know the datatype we can use the functions with a stated datatype, thats are similar to the “JSON.GetValue” function.

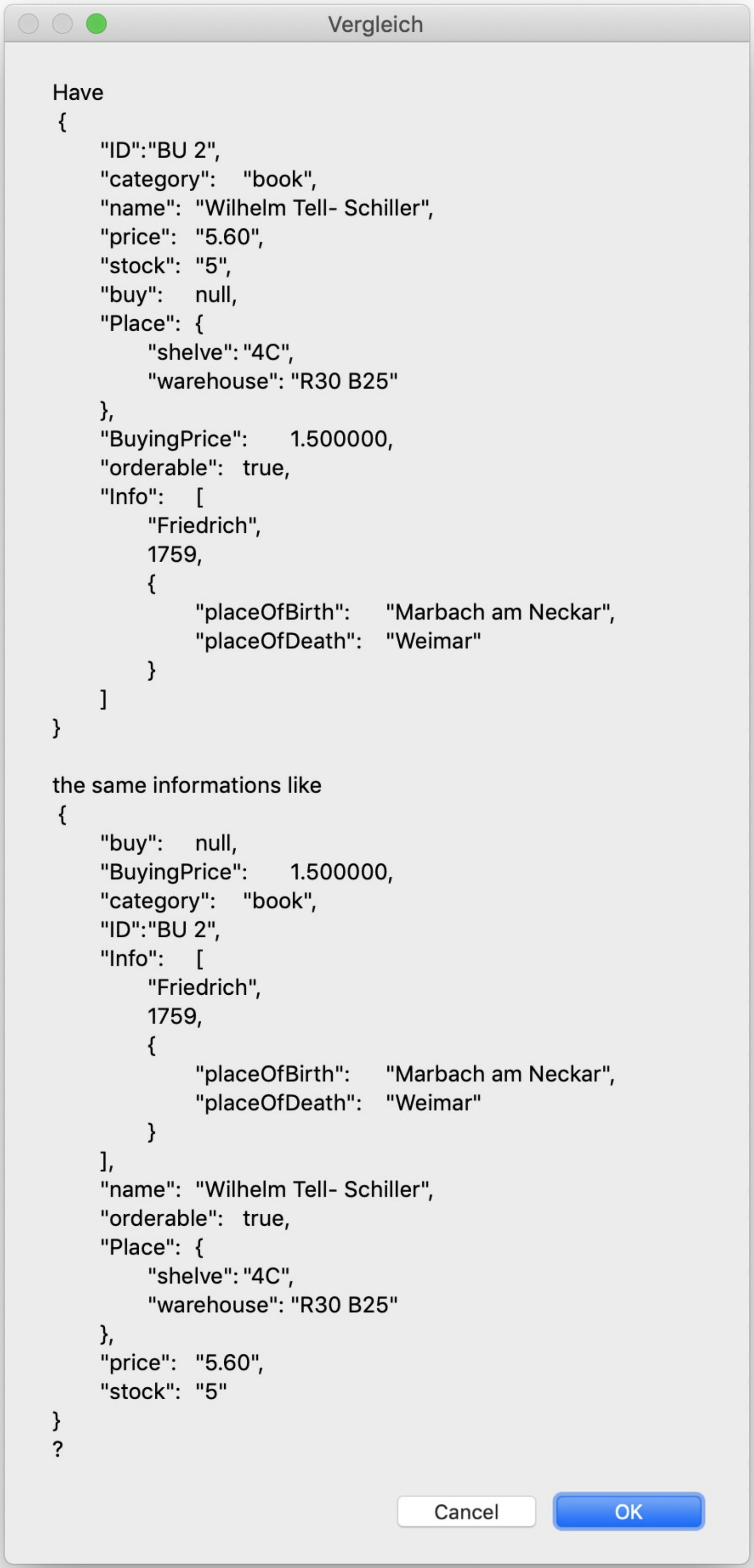
Reference numbers

For a lot of functions we used, we passed a JSON text in the parameters, but for most functions we can pass the JSON structure by a reference number, too. By using the reference number you reference the JSON object hold in memory by the plugin. This increase performance as we don't need to parse JSON text or create text from the objects. And as we store arrays and objects indexed in memory, we can access them very quickly.
We get the reference number of a JSON text with the „JSON.Parse" function. When creation an object or array, we can get a reference number as return, too. The number would be reference to an empty Object/Array. For that we use the „JSON.CreateArrayRef" und „JSON.CreateObjectRef“ function.
If you use references, you need to free them in the main storage, if you don’t need them anymore. For that please use the „JSON.Release“ function, to release a single reference number and „JSON.ReleaseAll“ for all reference numbers.

Sort and compare

You can sort the elements of an array or an object. If you have for example an array with numbers they would be sort beginning with the smallest up to the biggest number. If you want to sort text values, they would be sort alphabetically. In JSON itself the order of elements in objects is not defined and should not be relevant. As a human, it is easier to interpret a JSON if it has a visible order of the keys, so you can find them yourself easier. For this you can use the“JSON.Sort” sorting function. The array “[9, 6, 108, 4, 78, 45, 24, 9388, 788, 167]” would be “[4, 6, 9, 24, 45, 78, 108, 167, 788, 9388]” after using this function.

If you want to compare two JSON texts, they don’t need to be in the same order. You can use the „JSON.Equals“ function for comparison. If both JSON texts carry the same information the return value is 1, otherwise zero.



Have

```
{
     "ID":"BU 2",
     "category":   "book",
     "name":  "Wilhelm Tell- Schiller",
     "price":  "5.60",
     "stock":  "5",
     "buy":    null,
     "Place":  {
          "shelve": "4C",
          "warehouse": "R30 B25"
     },
     "BuyingPrice":     1.500000,
     "orderable":  true,
     "Info":    [
          "Friedrich",
          1759,
          {
               "placeOfBirth":     "Marbach am Neckar",
               "placeOfDeath":     "Weimar"
          }
     ]
}
```

the same informations like

```
{
     "buy":    null,
     "BuyingPrice":     1.500000,
     "category":   "book",
     "ID":"BU 2",
     "Info":    [
          "Friedrich",
          1759,
          {
               "placeOfBirth":     "Marbach am Neckar",
               "placeOfDeath":     "Weimar"
          }
     ],
     "name":  "Wilhelm Tell- Schiller",
     "orderable":  true,
     "Place":  {
          "shelve": "4C",
          "warehouse": "R30 B25"
     },
     "price":  "5.60",
     "stock":  "5"
}
```

?

JSON to HTML

With „JSON.toHTML“ you can convert JSON to HTML and show it via data URL in a web viewer. This is very useful for debugging web services with REST APIs. The objects are written into separate tables. The rows are the objects. You can set a CSS in the parameters for the formatting.

Import of JSON

If you get a JSON text, sometimes you want to import the informations to your database in a table. For this you can use divers functions, that the plugin provide for the import. We want to import the single products to a table. For that we look with „JSON.GetObjectItemTree“ to the tree level of the products.

```
Set Variable [ $text ; Value: MBS("JSON.GetObjectItemTree";JSON::JSON;"shop") ]
```

Then we start the import. If we have no error, we want the number of objects of the JSON. With „JSON.Import.Todo“ we test how many records still need to be imported. With the pause of the script the program get time to work. After the pause we proof the status and test if the import is still running. The status is „Working“ in that period of time. If the status is different, the import is finish or we got an error.

```
Set Variable [ $r ; Value: MBS("JSON.Import";$text) ]
Set Variable [ $text ; Value: "" ]
If [ MBS("IsError")=0 ]
      Set Variable [ $total ; Value: MBS("JSON.Import.Total") ]
      Loop
            Set Variable [ $todo ; Value: MBS("JSON.Import.Todo") ]
            Pause/Resume Script [ Duration (seconds): 1 ]
            Set Variable [ $s ; Value: MBS("JSON.Import.Status") ]
            Exit Loop If [ $s ≠ "Working" ]
      End Loop
```

If we got an error, we determine the number and the list of errors.

```
Set Variable [ $errorcount ; Value: MBS( "JSON.Import.ErrorCount" ) ]
Set Variable [ $errorlist ; Value: MBS("JSON.Import.Errors") ]
```

FileMaker have a new table with the products. The fields names like the single elements of the object. If you want to give the table name a prefix, we can set this prefix in the parameters of the „JSON.Import“ function. If we want to cancel an import, we call the „JSON.Import.Cancel“ function.

Attention: On the server you can’t use the pause script step, because of this you call directly JSON.Import.Work in the loop. We can only create fields and tables with FileMaker Pro (Advanced).

SQL

With the “FM.SQL.Execute” function you can send requests to a Database. With “FM.SQL.JSONRecords” you get the response as JSON Objekt. We set in the parameters the SQL reference, that we get from the first function and names of the single elements. The names must be in the same order like the SELECT part of the SQL query.

The function “FM.SQL.JSONRecord” returns only one Record as a JSON. The MBS FileMaker Plugin provide a lot of SQL functions that you can use FileMaker internally or in connection to a SQL databases like MySQL.

A library with JSON

For data exchange it can be useful to have the library as JSON Code. For that you use the ”Dictionary.ToJSON“ function. In the parameters you set the reference number of the dictionary.

A dictionary is an associative array, where values are stored based on a key. The lookup is very fast due to using an index.

LDAP

You can use LDAP functions with JSON structures. When you query LDAP data, you can use the “LDAP.JSON” function to get the data as JSON block.
For adding, changing or deleting of LDAP data in the JSON you use "LDAP.AddJSON" and "LDAP.ModifyJSON" functions. One request e.g. can look like this:

```
[{
    ”operation”: “Add”,
    ”type”: “cn”,
    ”value”: “Your Company name”
}]
```

It is a JSON array with objects.
Every object in that array has an element operation. This element set, if we want to add, replace, increment or delete thus object. In Addition we have the nodes type and value. That define the type and value of an entry. The values can be single values and lists.

How you can use this function?

The presented functions and many more are included with the MBS FileMaker Plugin. If you need a license please visit our shop on our website.
If you have any questions, don't hesitate to ask. Have a lot of fun with this functions!

(Written by Stefanie Juchmes)