Using InterSystems IRIS Document Database

Version 2024.2
2024-09-05

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using InterSystems IRIS Document Database
PDF generated on 2024-09-05
InterSystems IRIS® Version 2024.2
Copyright © 2024 InterSystems Corporation
All rights reserved.

Table of Contents

1 Introducing InterSystems IRIS Document Database

InterSystems IRIS® Document Database (%DocDB in the InterSystems Class Library) is a facility for storing and retrieving database data as collections of JSON documents. Data storage in JSON (JavaScript Object Notation) format provides support for web-based data exchange. It is compatible with, but separate from, traditional SQL table and field (class and property) data storage and retrieval.

InterSystems supports developing Document Databases and applications in the several languages and environments. See the following sections for more information:

- Using Document Database (%DocDB) with ObjectScript
- Using Document Database with REST
- Using Document Database with Java
- Using Document Database with .NET

The word "document" is used here as a specific industry-wide technical term, as a dynamic data storage structure. "Document", as used in Document Database, should not be confused with a text document, or with documentation.

1.1 Features and Benefits

By its nature, InterSystems IRIS Document Database is a schema-less data structure. That means that each document has its own structure, which may differ from other documents in the same database. This has several benefits when compared with SQL, which requires a predefined data structure.

Some of the key features of Document Database include:

- *Application Flexibility*: Documents do not require a predefined schema. This allows applications to rapidly set up their data environments, adapt to changes in data structure, and rapidly capture data in various formats. Document Database can begin capturing data immediately, without having to define a structure for that data. This is ideal for unpredictable data feeds, as are often found in web-based and social media data sources. If in capturing a body of data, structures within that data become evident or emerge as useful, your document data structure can evolve. Existing captured data can co-exist with this more-structured data representation. It is up to your application to determine the data structure for each document and process it appropriately. One way to do this is to establish a key:value pair representing the document structure version. Thus, conversion of data from one JSON structure to another can be performed gradually, without interrupting data capture or access.

- *Sparse Data Efficiency*: Document databases are very efficient at storing sparse data because attributes with a particular key do not have to appear in all documents of a collection. A document may have one set of defined keys; another document in the same collection may have a very different set of defined keys. In contrast, SQL requires that every record contain every key; in sparse data many records have keys with NULL values. For example, an SQL patient medical record provides fields for many diagnoses, conditions, and test; for most patients most of these fields are NULL. The system allocates space for all of these unused fields. In a Document Database patient medical record only those keys that contain actual data are present.
- *Hierarchical Data Storage*: Document Database is very efficient at storing hierarchically structured data. In a key:value pair, data can be nested within the data to an unlimited number of levels. This means that hierarchical data can be stored de-normalized. In the SQL relational model, hierarchical data is stored normalized by using multiple tables.
- *Dynamic Data Types*: A key does not have a defined data type. The value assigned to the key has an associated data type. Therefore a key:value pair in one document may have one data type; a key:value pair for the same key in another document may have a different data type. Because data types are not fixed, you can change the data type of a key:value pair in a document at runtime by assigning a new value that has a different data type (see Data Types and Values later in this document).

These features of Document Database have important implications for application development. In a traditional SQL environment, the database design establishes data structure that is followed in developing applications. In Document Database, data structure is largely provided in the applications themselves.

1.2 JSON Structure

Document Database supports InterSystems JSON Dynamic Objects and JSON Dynamic Arrays (see "Creating and Modifying Dynamic Entities" in Using JSON). The examples in this section create JSON structures using the ObjectScript SET command.

The following example shows how hierarchical data can be stored using JSON. The first SET creates a dynamic abstract object containing nested JSON-structured key:value pairs and arrays. The example then converts the dynamic abstract object to a JSON string, then inserts that JSON string into an existing document database as a document.

ObjectScript

```
SET dynAbObj = {
 "FullName":"John Smith",
 "FirstName":"John",
 "Address":{
            "street":"101 Main Street",
            "city":"Mapleville",
            "state":"NY",
            "postal code":10234
           },
 "PhoneNumber":
            [
             {"type":"home","number":"212-456-9876"},
             {"type":"cell","number":"401-123-4567"},
             {"type":"work","number":"212-444-5000"}
            ]
}
SET jstring = dynAbObj.%ToJSON() // dynamic abstract object to JSON string
DO personDB.%FromJSON(jstring)   // JSON string inserted into document database
```

In this example, *FullName* is stored as a simple key:value pair. *Address* has a substructure which is stored as an object consisting of key:value pairs. *PhoneNumber* has a substructure which is stored as an array.

1.2.1 De-Normalized Data Structure

The following is a JSON example of a traditional SQL normalized relational data structure. It consists of two documents, which might be contained in two different collections:

```
{
   "id":123,
   "Name":"John Smith",
   "DOB":"1990-11-23",
   "Address":555
}

{
   "id":555,
   "street":"101 Main Street",
   "city":"Mapleville",
   "state":"NY",
   "postal code":10234
 }
```

The following is the same data de-normalized, specified as a single document in a collection containing a nested data structure:

```
{
   "id":123,
   "Name":"John Smith",
   "DOB":"1990-11-23",
   "Address":{
              "street":"101 Main Street",
              "city":"Mapleville",
              "state":"NY",
              "postal code":10234
             }
 }
```

In SQL converting from the first data structure to the second would involve changing the table data definition then migrating the data.

In Document Database, because there is no fixed schema, these two data structures can co-exist as different representations of the same data. The application code can access either data structure as required. You can either migrate the data to the new data structure, or leave the data unchanged in the old data structure format, in which case Document Database migrates data each time it accesses it using the new data structure.

1.2.2 Data Types and Values

In InterSystems IRIS Document Database, a key does not have a data type. However, a data value imported to Document Database may have an associated data type. Because the data type is associated with the specific value, replacing the value with another value may result in changing the data type of the key:value pair for that record.

Document Database does not have any reserved words or any special naming conventions. In a key:value pair, any string can be used as a key; any string or number can be used as a value. The key name can be the same as the value: `"name":"name"`. A key name can be the same as its index name.

Document Database represents data values as JSON values. The following representations are used:

String values

All string values are represented by String.

Numbers

Numbers are represented in canonical form, with the following exception: JSON fractional numbers between 1 and -1 are represented with a leading zero integer (for example, `0.007`); the corresponding InterSystems IRIS numbers are represented without the leading zero integer (for example, `.007`).

$DOUBLE numbers

Represented as IEEE double-precision (64–bit) floating point numbers.

Non-printing characters

JSON provides escape code representations of the following non-printing characters (represented in ObjectScript by calls to the $CHAR function):

- `$CHAR(8): "\b"`
- `$CHAR(9): "\t"`
- `$CHAR(10): "\n"`
- `$CHAR(12): "\f"`
- `$CHAR(13): "\r"`

All other non-printable characters are represented by an escaped hexadecimal notation. For example, `$CHAR(11)` as `"\u000b"`. Printable characters can also be represented using escaped hexadecimal (Unicode) notation. For example, the Greek lowercase letter alpha can be represented as `"\u03b1"`.

Other escaped characters

JSON escapes two printable characters, the double quote character and the backslash character:

- `$CHAR(34): "\""`
- `$CHAR(92): "\\"`

1.2.3 JSON Special Values

JSON special values can only be used within JSON objects and JSON arrays. They are different from the corresponding ObjectScript special values. JSON special values are specified without quotation marks (the same values within quotation marks is an ordinary data value). They can be specified in any combination of uppercase and lowercase letters; they are stored as all lowercase letters.

The following examples demonstrate special values as used in ObjectScript code.

- JSON represents the absence of a value by using the `null` special value. Because Document Database does not normally include a key:value pair unless there is an actual value, `null` is only used in special circumstances, such as a placeholder for an expected value. This use of `null` is shown in the following example:

  ObjectScript

  ```
  SET jsonobj = {"name":"Fred","spouse":null}
  WRITE jsonobj.%ToJSON()
  ```

- JSON represents a boolean value by using the `true` and `false` special values. This use of boolean values is shown in the following example:

  ObjectScript

  ```
  SET jsonobj = {"name":"Fred","married":false}
  WRITE jsonobj.%ToJSON()
  ```

  ObjectScript specifies boolean values using 0 and 1. (Actually “true” can be represented by 1 or by any non-zero number.) These values are not supported as boolean values within JSON documents.

In a few special cases, JSON uses parentheses to clarify syntax:

- If you define a local variable with the name null, true, or false, you must use parentheses within JSON to have it treated as a local variable rather than a JSON special value. This is shown in the following example:

  ObjectScript

  ```
  SET true=1
  SET jsonobj = {"bool":true,"notbool":(true)}
  WRITE jsonobj.%ToJSON()
  ```

- If you use the ObjectScript Follows operator (]) within an expression, you must use parentheses within JSON to have it treated as this operator, rather than as a JSON array terminator. In the following example, the expression `b]a` tests whether b follows a in the collation sequence, and returns an ObjectScript boolean value. The Follows expression must be enclosed in parentheses:

  ObjectScript

  ```
  SET a="a",b="b"
  SET jsonarray=[(b]a)]
  WRITE jsonarray.%ToJSON()
  ```

2 Using Document Database with ObjectScript

The InterSystems IRIS® data platform Document Database (%DocDB) supplies methods that enable you to work with Document Databases in ObjectScript. For further details on invoking JSON methods from ObjectScript, refer to the *Using JSON* manual.

2.1 Components of ObjectScript %DocDB

The ObjecScript package name for the Document Database is %DocDB. It contains the following classes:

- %DocDB.Database — an ObjectScript persistent class used to manage the documents. A Database is a set of Document objects, implemented by a persistent class that extends %DocDB.Document. You use methods of this class to create a database, retrieve an existing database, or delete a database, and within a database to insert a document, retrieve a document, or delete a document.
- %DocDB.Document — a structure used to store document data. It consists of the document ID, the last modified date, and the document contents. Content is stored as a dynamic object or dynamic array:
  - %Library.DynamicObject — multiple JSON key:value pairs
  - %Library.DynamicArray — an ordered list of JSON values

  Dynamic objects and arrays are based on abstract class %Library.DynamicAbstractObject.

For full details on all methods and properties, see the entries in the Class Library.

2.2 Using the $SYSTEM.DocDB Interface

In ObjectScript, some important %DocDB methods can optionally be accessed via the special $SYSTEM interface (implemented in %SYSTEM.DocDB). For example, you can get a list of all entry points by calling:

```
do $SYSTEM.DocDB.Help()
```

The following methods are available:

- **CreateDatabase**`(databaseName,documentType,resource)` — creates a new Database in the current namespace. If a Database of that name already exists then return an error.
- **DropAllDatabases**`()` — deletes all Databases defined in the current namespace and their currently visible extent (user data). Returns an array containing the names of all Databases successfully dropped.
- **DropDatabase**`(databaseName)` — deletes the definition and currently visible extent of an existing Database. Returns `true` if the Database exists and the drop is successful,
- **Exists**`(databaseName)` — returns `true` if database `databaseName` is defined in the current namespace,
- **GetAllDatabases**`()` — returns a %Library.DynamicArray containing the names of all Databases defined in the current namespace.
- **GetDatabase**`(databaseName)` — retrieves the Database definition, this includes the Name, Class, Resource and Document Type Class. If the requested Database does not exist then an exception is thrown.
- **Help**`()` — writes a list of available %DocDB methods to the console.

2.3 Creating a Database with ObjectScript

A Database is an instance of an ObjectScript persistent class that extends abstract class %DocDB.Document. A separate Database must be created for each namespace used by Document Database. Only one Database is required per namespace. Commonly, it is assigned the same name as the namespace name.

The following example shows how to create a Database through class definition:

```
Class MyDBs.People Extends %DocDB.Document [ DdlAllowed ]
{
}
```

The following example shows how to create a Database using the **CreateDatabase()** method, specifying a package name:

ObjectScript

```
  set personDB = $SYSTEM.DocDB.CreateDatabase("MyDBs.People")
```

This example creates the Database using the default package name:

ObjectScript

```
  set personDB = $SYSTEM.DocDB.CreateDatabase("People")
```

2.4 Document Storage

A Document is stored in the *%Doc* property of an instance of the %DocDB.Document class you create. This is shown in the following example, which stores a JSON array in the %Doc property:

ObjectScript

```
  do $SYSTEM.DocDB.DropDatabase("MyDBs.People")
  set personDB = $SYSTEM.DocDB.CreateDatabase("MyDBs.People")
  set myOref = ##class(MyDBs.People).%New()
  set myOref.%Doc = ["Anne","Bradford","Charles","Deborah"]
  set docOref = myOref.%Doc
  write "%Doc property oref: ",docOref,!
  write "%Doc Property value: ",docOref.%ToJSON()
```

By default, the %DocDB.Document.%Doc data type is %Library.DynamicAbstractObject, which is the data type used to store a JSON object or a JSON array. You can specify a different data type in the **CreateDatabase()** method.

Other Database properties:

- %DocDB.Document.*%DocumentId* is an IDENTITY property containing a unique integer that identifies a document; %DocumentId counts from 1. In most cases, %DocumentId values are system-assigned. A %DocumentId must be unique; %DocumentIds are not necessarily assigned sequentially; gaps may occur in an assignment sequence. Document Database also automatically generates an IdKey index for %DocumentId values.
- %DocDB.Document.*%LastModified* records a UTC timestamp when the Document instance was defined.

2.5 Using DocDB Databases with ObjectScript

The following sections describe how to use some important %DocDB.Database methods:

- **%GetDatabase()** — get an existing document database by name.
- **%CreateProperty()** — define the property for a key/value pair.
- **%SaveDocument()** — insert or replace a document.
- **%Size()** — count the documents in a database.
- **%GetDocument()** — get a document by id.
- **%DeleteDocument()** — delete by id or other criteria.
- **%ExecuteQuery()** — return document data as a result set.
- **%FindDocuments()** — return documents that match specified criteria.

2.5.1 Get a Database: %GetDatabase()

See Creating a Document Database in ObjectScript for various ways to create a document database.

To get an existing document database in the current namespace, invoke the **%GetDatabase()** method.

You assign a document database a unique name within the current namespace. The name can be qualified `"packagename.docdbname"` or unqualified. An unqualified database name defaults to the `ISC.DM` package.

The following example gets a Database if a Database with that name exists in the current namespace; otherwise, it creates a new one. It then uses the **%GetDatabaseDefinition()** method of the *db* object to display the database definition information, which is stored as a JSON Dynamic Object:

ObjectScript

```
  set EXISTS = $SYSTEM.DocDB.Exists("People")
  if EXISTS set db = $SYSTEM.DocDB.%GetDatabase("People")
  if 'EXISTS set db = $SYSTEM.DocDB.%CreateDatabase("People")

  set defn = db.%GetDatabaseDefinition()
  write defn.%ToJSON()
```

You can use **GetAllDatabases** to return a JSON array containing the names of all databases defined in this namespace. For example:

```
  write $SYSTEM.DocDB.GetAllDatabases().%ToJSON()
```

2.5.2 Define a Property: %CreateProperty()

In order to retrieve a document by a key:value pair, you must use **%CreateProperty()** to define a property for that key. Defining a property automatically creates an index for that key which InterSystems IRIS maintains when documents are inserted, modified, and deleted. A property must specify a data type for the key. A property can be specified as accepting only unique values (1), or as non-unique (0); the default is non-unique.

The following example assigns two properties to the database. It then displays the database definition information:

ObjectScript

```
  do $SYSTEM.DocDB.DropDatabase("People")
  set db = $SYSTEM.DocDB.CreateDatabase("People")
  do db.%CreateProperty("firstName","%String","$.firstName",0) // creates non-unique property
  do db.%CreateProperty("lastName","%String","$.lastName",1)   // create a unique property and index
  write db.%GetDatabaseDefinition().%ToJSON()
```

2.5.3 Insert or Replace a Document: %SaveDocument()

You can insert or replace a document in a database using either the documentID or data selection criteria.

The **%SaveDocument()** and **%SaveDocumentByKey()** methods save a document, inserting a new document or replacing an existing document. **%SaveDocument()** specifies the document by documentId; **%SaveDocumentByKey()** specifies the document by key name and key value.

If you do not specify a documentId, **%SaveDocument()** inserts a new document and generates a new documentId. If you specify a documentId, it replaces an existing document with that documentId. If you specify a documentId and that document does not exist, it generates an `ERROR #5809` exception.

The document data consists of one or more key:value pairs. If you specify a duplicate value for a key property that is defined as unique, it generates an `ERROR #5808` exception.

The **%SaveDocument()** and **%SaveDocumentByKey()** methods return a reference to the instance of the database document class. This is always a subclass of %DocDB.Document. The method return data type is %DocDB.Document.

The following example inserts three new documents and assigns them documentIds. It then replaces the entire contents of the document identified by documentId 2 with the specified contents:

ObjectScript

```
  do $SYSTEM.DocDB.DropDatabase("People")
  set db = $SYSTEM.DocDB.CreateDatabase("People")
  write db.%Size(),!
  do db.%CreateProperty("firstName","%String","$.firstName",0)
  set val = db.%SaveDocument({"firstName":"Serena","lastName":"Williams"})
  set val = db.%SaveDocument({"firstName":"Bill","lastName":"Faulkner"})
  set val = db.%SaveDocument({"firstName":"Fred","lastName":"Astare"})
  write "Contains ",db.%Size()," documents: ",db.%ToJSON()
  set val = db.%SaveDocument({"firstName":"William","lastName":"Faulkner"},2)
  write !,"Contains ",db.%Size()," documents: ",db.%ToJSON()
```

The following example chains the **%Id()** method (inherited from %Library.Persistent) to each **%SaveDocument()**, returning the documentId of each document as it is inserted or replaced:

ObjectScript

```
  do $SYSTEM.DocDB.DropDatabase("People")
  set db = $SYSTEM.DocDB.CreateDatabase("People")
  do db.%CreateProperty("firstName","%String","$.firstName",0)
  write db.%SaveDocument({"firstName":"Serena","lastName":"Williams"}).%Id(),!
  write db.%SaveDocument({"firstName":"Bill","lastName":"Faulkner"}).%Id(),!
  write db.%SaveDocument({"firstName":"Fred","lastName":"Astare"}).%Id(),!
  write "Contains ",db.%Size()," documents: ",db.%ToJSON()
  write db.%SaveDocument({"firstName":"William","lastName":"Faulkner"},2).%Id(),!
  write !,"Contains ",db.%Size()," documents: ",db.%ToJSON()
```

The final lines of both the previous examples call the Document.%Size() method.

2.5.4 Count Documents in a Database: %Size()

To count the number of documents in a database, invoke the %DocDB.Document **%Size()** method:

ObjectScript

```
  set db = $SYSTEM.DocDB.GetDatabase("People")
  write db.%Size()
```

See Insert or Replace a Document for more examples.

2.5.5 Get Document in a Database: %GetDocument()

To retrieve a single document from the database by %DocumentId, invoke the **%GetDocument()** method, as shown in the following example:

ObjectScript

```
  do db.%GetDocument(2).%ToJSON()
```

This method returns only the %Doc property contents. For example:

```
{"firstName":"Bill","lastName":"Faulkner"}
```

The method return type is %Library.DynamicAbstractObject.

If the specified %DocumentId does not exist, **%GetDocument()** generates an ERROR #5809 exception: “Object to load not found”.

You can retrieve a single document from the database by key value using **%GetDocumentByKey()**.

You can also return a single document from the database by %DocumentId, using the %FindDocuments() method. For example:

ObjectScript

```
  do db.%FindDocuments(["%DocumentId",2,"="]).%ToJSON()
```

This method returns the complete JSON document, including its wrapper:

```
{"sqlcode":100,"message":null,"content":[{"%Doc":"{\"firstName\":\"Bill\",\"lastName\":\"Faulkner\"}",
"%DocumentId":2,"%LastModified":"2018-03-06 18:59:02.559"}]}
```

2.5.6 Delete a Document: %DeleteDocument()

You can delete documents from a database either by documentID or by data selection criteria.

- The **%DeleteDocument()** method deletes a single document identified by documentId:

  ObjectScript

  ```
  set val = db.%DeleteDocument(2)
  write "deleted document = ",val.%ToJSON()
  ```

  Successful completion returns the JSON value of the deleted document. Failure throws a StatusException error.

- The **%DeleteDocumentByKey()** method deletes a document identified by document contents. Specify a key:value pair.

- The **%Clear()** method deletes all documents in the database and returns the oref (object reference) of the database, as shown in the following example:

  ObjectScript

  ```
  set dboref = db.%Clear()
  write "database oref: ",dboref,!
  write "number of documents:",db.%Size()
  ```

  This permits you to chain methods, as shown in the following example:

  ObjectScript

  ```
  write db.%Clear().%SaveDocument({"firstName":"Venus","lastName":"Williams"}).%Id(),!
  ```

2.5.7 Query Documents in a Database: %ExecuteQuery()

You can use the **%ExecuteQuery()** method to return document data from a database as a result set. You specify a standard SQL query SELECT, specifying the database name in the **FROM** clause. If the database name was created with no package name (schema), specify the default `ISC_DM` schema.

The following example retrieves the %Doc content data as a result set from all documents in the database:

```
set rval=db.%ExecuteQuery("SELECT %Doc FROM ISC_DM.People")
do rval.%Display()
```

The following example uses a WHERE clause condition to limit what documents to retrieve by documentId value:

```
set rval=db.%ExecuteQuery("SELECT %DocumentId,%Doc FROM ISC_DM.People WHERE %DocumentId > 2")
do rval.%Display()
```

The following example retrieve the %DocumentId and the lastName property from all documents that fulfill the **WHERE** clause condition. Note that to retrieve the values of a user-defined key, you must have defined a document property for that key:

```
 set rval=db.%ExecuteQuery("SELECT %DocumentId,lastName FROM ISC_DM.People WHERE lastName %STARTSWITH
'S'")
 do rval.%Display()
```

For further information on handling a query result set, refer to Returning the Full Result Set or Returning Specific Values from the Result Set in *Using InterSystems SQL*.

2.5.8 Find Documents in a Database: %FindDocuments()

To find one or more documents in a database and return the document(s) as JSON, invoke the **%FindDocuments()** method. This method takes any combination of three optional positional predicates: a restriction array, a projection array, and a limit key:value pair.

The following examples, with no positional predicates, both return all of the data in all of the documents in the database:

ObjectScript

```
write db.%FindDocuments().%ToJSON()
```

ObjectScript

```
write db.%FindDocuments(,,).%ToJSON()
```

2.5.8.1 Restriction Predicate Array

The restriction predicate syntax `["property","value","operator"]` returns the entire contents of the matching documents. You specify as search criteria the property, value, and operator as an array. If you do not specify the operator, it defaults to `"="`. You can specify more than one restriction as an array of restriction predicates with implicit AND logic: `[["property","value","operator"],["property2","value2","operator2"]]`. The restriction predicate is optional.

The following example returns all documents with a documentId greater than 2:

ObjectScript

```
set result = db.%FindDocuments(["%DocumentId",2,">"])
write result.%ToJSON()
```

or by chaining methods:

ObjectScript

```
write db.%FindDocuments(["%DocumentId",2,">"]).%ToJSON()
```

If the contents of documents match the search criteria, It returns results such as the following:

```
{"sqlcode":100,"message":null,"content":[{"%Doc":"{\"firstName\":\"Fred\",\"lastName\":\"Astare\"}","%DocumentId":"3","%LastModified":"2018-03-05
18:15:30.39"},{"%Doc":"{\"firstName\":\"Ginger\",\"lastName\":\"Rogers\"}","%DocumentId":"4","%LastModified":"2018-03-05
 18:15:30.39"}]}
```

If no documents match the search criteria, it returns:

```
{"sqlcode":100,"message":null,"content":[]}
```

To find a document by a key:value pair, you must have defined a document property for that key:

ObjectScript

```
do $SYSTEM.DocDB.DropDatabase("People")
set db = $SYSTEM.DocDB.CreateDatabase("People")
write db.%Size(),!
do db.%CreateProperty("firstName","%String","$.firstName",0)
set val = db.%SaveDocument({"firstName":"Fred","lastName":"Rogers"})
set val = db.%SaveDocument({"firstName":"Serena","lastName":"Williams"})
set val = db.%SaveDocument({"firstName":"Bill","lastName":"Faulkner"})
set val = db.%SaveDocument({"firstName":"Barak","lastName":"Obama"})
set val = db.%SaveDocument({"firstName":"Fred","lastName":"Astare"})
set val = db.%SaveDocument({"lastName":"Madonna"})
set result = db.%FindDocuments(["firstName","Fred","="])
write result.%ToJSON()
```

You can use a variety of predicate operators, including %STARTSWITH, IN, NULL, and NOT NULL, as shown in the following examples:

ObjectScript

```
set result = db.%FindDocuments(["firstName","B","%STARTSWITH"])
write result.%ToJSON()

set result = db.%FindDocuments(["firstName","Bill,Fred","IN"])
write result.%ToJSON()

set result = db.%FindDocuments(["firstName","NULL","NULL"])
write result.%ToJSON()

set result = db.%FindDocuments(["firstName","NULL","NOT NULL"])
write result.%ToJSON()
```

2.5.8.2 Projection Predicate Array

To return only some of the values in returned documents you use a projection. The optional projection predicate, `["prop1","prop2",...]`, is an array listing the keys for which you wish to return the corresponding values. If you specify a user-defined key in the projection array, you must have defined a document property for that key.

The syntax `["property","value","operator"],["prop1","prop2",...]` returns the specified properties from the matching documents.

ObjectScript

```
set result = db.%FindDocuments(["firstName","Bill","="],["%DocumentId","firstName"])
write result.%ToJSON()
```

You can specify a projection with or without a restriction predicate. Thus both of the following are valid syntax:

- `db.%FindDocuments(["property","value","operator"],[prop1,prop2,...])` restriction and projection.
- `db.%FindDocuments(,[prop1,prop2,...])` no restriction, projection.

2.5.8.3 Limit Predicate

You can specify a limit key:value predicate, `{"limit":int}`, to return, at most, only the specified number of matching documents.

The syntax `["property","value","operator"],["prop1","prop2",...],{"limit":int}` returns the specified properties from the specified limit number of documents.

ObjectScript

```
set result = db.%FindDocuments(["firstName","Bill","="],["%DocumentID","firstName"],{"limit":5})
write result.%ToJSON()
```

This returns data from, at most, 5 documents.

You can specify a limit with or without a restriction predicate or a projection predicate. Thus all the following are valid syntax:

- `db.%FindDocuments(["property","value","operator"],,{"limit":int})` restriction, no projection, limit.
- `db.%FindDocuments(,[prop1,prop2,...],{"limit":int})` no restriction, projection, limit.
- `db.%FindDocuments(,,{"limit":int})` no restriction, no projection, limit.

The following example specifies no restriction, a projection, and a limit:

ObjectScript

```
write db.%FindDocuments(,["%DocumentId","lastName"],{"limit":3}).%ToJSON()
```

3

Using Document Database with REST

The InterSystems IRIS® data platform Document Database REST client supplies methods that enable you to work with Document Databases from REST (see Creating REST Services for detailed information on InterSystems REST support). The REST API differs from Document Database clients in other programming languages because REST acts on resources while the other clients act on objects. This resource orientation is fundamental to the nature of REST.

The curl examples in this chapter specify port number 57774. This is only an example. You should use the port number appropriate for your InterSystems IRIS instance.

In curl the GET command is the default; therefore, a curl command that omits -X GET defaults to -X GET.

For Document Database the only valid Content-Type is application/json. If an unexpected Content-Type is requested then an HTTP Response Code = 406 is issued.

To use the REST API, you must enable the *%Service_DocDB* service. To do so, navigate to System Administration > Security > Services, select *%Service_DocDB*, and select the *%Service_DocDB* checkbox.

For further details on the REST API refer to the %Api.DocDB.v1 in the *InterSystems Class Reference*.

3.1 Managing Databases

- GetAllDatabases: Returns a JSON array which contains the names of all of the databases in the namespace.

  ```
  curl -i -X GET -H "Content-Type: application/json" http://localhost:57774/api/docdb/v1/namespaceName
  ```

- DropAllDatabases: Deletes all of the databases in the namespace for which the current user has Write privileges.

  ```
  curl -i -X DELETE -H "Content-Type: application/json"
  http://localhost:57774/api/docdb/v1/namespaceName
  ```

- CreateDatabase: Creates a database in the namespace.

  ```
  curl -i -X POST -H "Content-Type: application/json"
  http://localhost:57774/api/docdb/v1/namespaceName/db/databaseName ?type= documentType& resource=
  databaseResource
  ```

- GetDatabase: Returns the database definition of a database in the namespace.

  ```
  curl -i -X GET -H "Content-Type: application/json"
  http://localhost:57774/api/docdb/v1/namespaceName/db/databaseName
  ```

- DropDatabase: Drops a database from the namespace.

  ```
  curl -i -X DELETE -H "Content-Type: application/json"
  http://localhost:57774/api/docdb/v1/namespaceName/db/databaseName
  ```

3.2 Managing Properties

- CreateProperty: Creates a new property or replaces an existing property in the specified database. The property is defined by URL parameters and not Content. All parameters are optional.

  ```
  curl -i -X POST -H "Content-Type: application/json"
  http://localhost:57774/api/docdb/v1/namespaceName/prop/databaseName/
  propertyName?type= propertyType& path= propertyPath& unique=propertyUnique
  ```

  The following example creates the City property in the Wx database:

  ```
  http://localhost:57774/api/docdb/v1/mysamples/prop/wx/city?type=%String&path=query.results.channel.location.city&0
  ```

- GetProperty: Returns a property definition from the specified database.

  ```
  curl -i -X GET -H "Content-Type: application/json"
  http://localhost:57774/api/docdb/v1/namespaceName/prop/databaseName/propertyName
  ```

  The returned property definition is a JSON structure such as the following:

  ```
  {"content":{"Name":"city","Type":"%Library.String"}}
  ```

- DropProperty: Deletes a property definition from the specified database.

  ```
  curl -i -X DELETE -H "Content-Type: application/json"
  http://localhost:57774/api/docdb/v1/namespaceName/prop/databaseName/propertyName
  ```

The following is a JSON property definition:

```
{"content":{"Name":"mydocdb","Class":"DB.MyDocDb","properties":[Array[5]
   0: {"Name":%%OTD,"Type":"%RawString"},
   1: {"Name":%Concurrency,"Type":"%RawString"},
   2: {"Name":%Doc,"Type":"%Library.DynamicAbstractObject"},
   3: {"Name":%DocumentId,"Type":"%Library.Integer"},
   4: {"Name":%LastModified,"Type":"%Library.UTC"}
   ]}}
```

3.3 Inserting and Updating Documents

- SaveDocument: Insert a new document into the specified database.

  ```
  curl -i -X POST -H "Content-Type: application/json"
  http://localhost:57774/api/docdb/v1/namespaceName/doc/databaseName/
  ```

  Note there is a slash at the end of this URI.

  To insert one or more documents, perform a POST. The body of the request is either a JSON document object or a JSON array of document objects. Note that the document objects may be in the unwrapped form with just content. This unwrapped form always results in an insert with a new %DocumentId. Specifying a wrapped document object whose %DocumentId is not present in the database results in an insert with that %DocumentId. Otherwise, the

%DocumentId property is ignored and an insert with a new %DocumentId takes place. If the request is successful, a single JSON document header object or an array of JSON document header objects is returned. If the request fails, the JSON header object is replaced by an error object.

- SaveDocument: Replace an existing document in the specified database.

  ```
  curl -i -X PUT -H "Content-Type: application/json"
  http://localhost:57774/api/docdb/v1/namespaceName/doc/databaseName/id
  ```

  To insert a single JSON document object at a specific id location. If the specified %DocumentId already exists, the system replaces the existing document with the new document.

- SaveDocumentByKey: Replace an existing document in the specified database.

  ```
  curl -i -X PUT -H "Content-Type: application/json"
  http://localhost:57774/api/docdb/v1/namespaceName/doc/databaseName/keyPropertyName/keyValue
  ```

3.4 Deleting Documents

- DeleteDocument: Delete a document from the specified database.

  ```
  curl -i -X DELETE -H "Content-Type: application/json"
  http://localhost:57774/api/docdb/v1/namespaceName/doc/databaseName/id
  ```

  Deletes the document specified by %DocumentId. If the request is successful, the specified document is deleted, the document wrapper metadata `{"%DocumentId":<IDnum>,"%LastModified":<timestamp>}` is returned, and a 200 (OK) status.

- DeleteDocumentByKey: Delete a document from the specified database.

  ```
  curl -i -X DELETE -H "Content-Type: application/json"
  http://localhost:57774/api/docdb/v1/namespaceName/doc/databaseName/keyPropertyName/keyValue
  ```

3.5 Retrieving a Document

- GetDocument: Return the specified document from the database.

  ```
  curl -i -X GET -H "Content-Type: application/json"
  http://localhost:57774/api/docdb/v1/namespaceName/doc/databaseName/id? wrapped=true|false
  ```

- GetDocumentByKey: Return a document by a property defined as a unique key from the database.

  ```
  curl -i -X POST -H "Content-Type: application/json"
  http://localhost:57774/api/docdb/v1/namespaceName/doc/databaseName/keyPropertyName/keyValue
  ```

  FindDocuments: Return all documents from the database that match the query specification.

  ```
  curl -i -X POST -H "Content-Type: application/json"
  http://localhost:57774/api/docdb/v1/namespaceName/find/databaseName? wrapped=true|false
  ```

The following curl script example supplies full user credentials and header information. It returns all of the documents in the Continents document database in the MySamples namespace:

```
curl --user _SYSTEM:SYS -w "\n\n%{http_code}\n" -X POST
-H "Accept: application/json" -H "Content-Type: application/json"
http://localhost:57774/api/docdb/v1/mysamples/find/continents
```

It returns JSON data from the Document database such as the following:

```
{"content":{"sqlcode":100,"message":null,"content":[
  {"%Doc":"{\"code\":\"NA\",\"name\":\"North America\"}","%DocumentId":"1","%LastModified":"2018-02-15
 21:33:03.64"},
  {"%Doc":"{\"code\":\"SA\",\"name\":\"South America\"}","%DocumentId":"2","%LastModified":"2018-02-15
 21:33:03.64"},
  {"%Doc":"{\"code\":\"AF\",\"name\":\"Africa\"}","%DocumentId":"3","%LastModified":"2018-02-15
21:33:03.64"},
  {"%Doc":"{\"code\":\"AS\",\"name\":\"Asia\"}","%DocumentId":"4","%LastModified":"2018-02-15
21:33:03.64"},
  {"%Doc":"{\"code\":\"EU\",\"name\":\"Europe\"}","%DocumentId":"5","%LastModified":"2018-02-15
21:33:03.64"},
  {"%Doc":"{\"code\":\"OC\",\"name\":\"Oceana\"}","%DocumentId":"6","%LastModified":"2018-02-15
21:33:03.64"},
  {"%Doc":"{\"code\":\"AN\",\"name\":\"Antarctica\"}","%DocumentId":"7","%LastModified":"2018-02-15
21:33:03.64"}]}}
```

Restriction: The following curl script example restricts the documents returned from the Continents document database by specifying a restriction object. This restriction limits the documents returned to those whose name begins with the letter A:

```
curl --user _SYSTEM:SYS -w "\n\n%{http_code}\n" -X POST
-H "Accept: application/json" -H "Content-Type: application/json"
http://localhost:57774/api/docdb/v1/mysamples/find/continents -d
'{"restriction":["Name","A","%STARTSWITH"]}'
```

It returns the following JSON documents from the Document database:

```
{"content":{"sqlcode":100,"message":null,"content":[
  {"%Doc":"{\"code\":\"AF\",\"name\":\"Africa\"}","%DocumentId":"3","%LastModified":"2018-02-15
21:33:03.64"},
  {"%Doc":"{\"code\":\"AS\",\"name\":\"Asia\"}","%DocumentId":"4","%LastModified":"2018-02-15
21:33:03.64"},
  {"%Doc":"{\"code\":\"AN\",\"name\":\"Antarctica\"}","%DocumentId":"7","%LastModified":"2018-02-15
21:33:03.64"}]}}
```

Projection: The following curl script example projects which JSON properties to return from each document in the Continents document database. It uses the same restriction as the previous example:

```
curl --user _SYSTEM:SYS -w "\n\n%{http_code}\n" -X POST
-H "Accept: application/json" -H "Content-Type: application/json"
http://localhost:57774/api/docdb/v1/mysamples/find/continents -d
'{"restriction":["Name","A","%STARTSWITH"],"projection":["%DocumentId","name"]}'
```

It returns JSON data from the Document database such as the following:

```
{"content":{"sqlcode":100,"message":null,"content":[
  {"%Doc":"{"%DocumentId":"3","name":"Africa"}},
  {"%Doc":"{"%DocumentId":"4","name":"Asia"}},
  {"%Doc":"{"%DocumentId":"7","name":"Antarctica"}}]}}
```

4

Using Document Database with Java

The InterSystems IRIS® Document Database driver for Java (com.intersystems.document) is an interface to Document Database on the server that enable you to work with Document Databases, collections, and individual documents.

- Connecting to the Database — demonstrates DataSource method createDataSource().
- Java Collections — demonstrates Collection methods getCollection(), drop(), size(), getAll()
- Java Document Transactions (Add/Create/Insert/Delete) — demonstrates Collection methods insert(), get(), remove(), replace(), upsert()
- Index and Query Java Collections — demonstrates Collection methods createIndex(), dropIndex(), createQuery()

4.1 Connecting to the Database

This section demonstrates document.DataSource method **createDataSource()** and Collection method **getConnection()**.

Get a Database Handle: createDataSource()

The database handle is the entry point for using the document data model in all client driver implementations. In Java, the database handle is a DataSource:

```
DataSource datasrc = DataSource.createDataSource();
datasrc.setPortNumber(1972);
datasrc.setServerName("localhost");
datasrc.setDatabaseName("USER");
datasrc.setUser("_SYSTEM");
datasrc.setPassword("SYS");
System.out.println("\nCreated datasrc\n");
```

This creates a DataSource object with the default pool size (10). The actual connections are lazily created at the time the first **getConnection()** method is invoked. At that time, the pool of connections will be created and will be available to service database requests. Default pool size can by modified by

```
datasrc.preStart(2);
datasrc.getConnection();
System.out.println("\nDataSource size =" + datasrc.getSize());
```

In this example, **getSize()** returns the current pool size (which may be smaller than the actual pool size if there are any asynchronous requests pending).

4.2 Java Collections

This section demonstrates Collection methods **getCollection()**, **drop()**, **size()**, **getAll()**

Get a Collection: getCollection()

A collection is a container for Document Database documents. It is the primary interface for working with a Document Database. You do not have to explicitly create a collection. **getCollection()** creates the specified collection if that collection does not yet exist:

```
Collection collectedDocs = Collection.getCollection(datasrc,"collectionName");
```

Count Documents in a Collection: size()

To count the number of documents with a collection instance invoke the **size()** method:

```
long size = collectedDocs.size();
```

Iterate through the Documents in a Collection: getAll()

To iterate through all of the documents in a collection, invoke the **getAll()** method, then sequence through that iterator:

```
List<Document> allDocuments = collectedDocs.getAll();
allDocuments.stream().forEach(doc ->
   System.out.println(doc.toJSONString())
   );
```

Drop a Collection: drop()

Dropping a collection will delete the extent of the collection and its metadata, except index definitions. This deletes all documents in the collection, but does not delete the collection itself. Any object references to the collection will stay valid and you can continue to work with it (e.g. store new documents). To drop (delete) a collection invoke the **drop()** method:

```
collectedDocs.drop();
```

4.3 Java Documents and Transactions (Add/Create/Insert/Delete)

document.Document class serves as a base class for both JSONArray and JSONObject. Documents are the main building blocks in the Document API. They can be saved, retrieved, updated etc. to/from Document using the Collections class.

Insert a Document: insert()

Use the Collection.**Insert()** method to persist a Document object.

To insert a single document invoke the **insert()** method:

```
Document doc = new JSONObject().put("Rank",1);
collectedDocs.insert(doc);
```

To insert an array of elements as individual documents:

```
List<Document> topTwo = new ArrayList<>();
  topTwo.add(new JSONObject().put("Rank",1));
  topTwo.add(new JSONObject().put("Rank",2));
BulkResponse bulk = collectedDocs.insert(topTwo);

System.out.println("Newly inserted document ids:");
for (String id : bulk.getIds())
    {
     System.out.print(id + " ");
    }
```

Retrieve a Document by id: get()

To retrieve a document in a collection, invoke the **get()** method and specify the document's integer id:

```
Document doc = collectedDocs.get("<id>");
```

where <id> is a quoted integer value (for example, `Get("23")`.)

Delete a Document by id: remove()

To delete a document from a collection, invoke the **remove()** method and specify the document's integer id:

```
collectedDocs.remove("<id>");
```

Update a Document by id: replace()

To update a document in a collection, invoke the **replace()** method and specify the document's integer id and the new data value:

```
collectedDocs.replace("<id>",new JSONObject().put("Rank",4));
```

Insert or Update a Document by id: upsert()

To either insert a document (if it does not exist) or update the document (if it does exist), invoke the **upsert()** method and specify the document's integer id and the new data value:

```
Document upsertDoc = new JSONObject().put("Rank",1);
String upsertId = collectedDocs.upsert(upsertDoc);
```

4.4 Index and Query Java Collections

Create or change an Index: createIndex()

To create an index to documents in a collection, invoke the **createIndex()** method and specify an index name and an index request object. The index request object must be JSON encoded. It can be simple or complex. To create a simple index, set the property "key" to the JSON path that you want to index. The default index type is bitmap. The following example creates an index:

```
collectedDocs.createIndex("indexName","{\"key\":\"HomeCity\"}");
```

If the specified index name already exists, **createIndex()** will redefine this index with the new index definition.

Delete an Index: dropIndex()

To delete an index to documents in a collection, invoke the **dropIndex()** method and specify an existing index name:

```
collectedDocs.dropIndex("indexName");
```

If the specified index name does not exist, **dropIndex()** performs no operation and does not return an error.

Query a Collection: createQuery()

To query an instance of a collection, invoke the **createQuery()** method that specifies the SELECT query. You then issue an **execute()** method that executes the query and creates a cursor that allows you to iterate through the query result set:

```
Query querySlams = collectedDocs.createQuery(
           "select Rank,\"First Name\",\"Last Name\",Country,Points,
            \"Grand Slams\" from JSON_TABLE(('ATPList'),
            ('{\"columns\":\"ATPList[*]\"}'))");
Cursor slamCursor = querySlams.execute();
System.out.println("Executed query on ATPList - results follow: ");
List<Document> slamList = slamCursor.getAll();
slamList.forEach(doc -> System.out.println(doc.toJSONString()));
```

5

Using Document Database with .NET

The InterSystems IRIS® Document Database driver for .NET (Document.Document) is an interface to Document Database on the server that enable you to work with Document Databases, collections, and individual documents.

- Connecting to the Database — demonstrates DataSource method CreateDataSource().
- .NET Collections — demonstrates Collection methods GetCollection(), Drop(), Size(), GetAll()
- .NET Document Transactions (Add/Create/Insert/Delete) — demonstrates Collection methods Insert(), Get(), Remove(), Replace(), Upsert()
- Index and Query .NET Collections — demonstrates Collection methods CreateIndex(), DropIndex(), CreateQuery()

5.1 Connecting to the Database

This section demonstrates DataSource method CreateDataSource() and Collection methods GetCollection(), Drop(), Size(), GetAll()

Get a Database Handle: CreateDataSource()

The database handle is the entry point for using the document data model in all client driver implementations. In .NET, the database handle is a DataSource:

```
DataSource datasrc = DataSource.CreateDataSource();
datasrc.SetConnectionString(
   "Server = 127.0.0.1;
   Port=1972;
   Namespace=USER;
   Password = SYS;
   User ID = _SYSTEM;"
);
Connection connection = datasrc.GetConnection();
```

5.2 .NET Collections

This section demonstrates Collection methods **GetCollection()**, **Drop()**, **Ssize()**, **GetAll()**

Get a Collection: GetCollection()

A collection is a container for Document Database documents. It is the primary interface for working with a Document Database. You do not have to explicitly create a collection. **GetCollection()** creates the specified collection if that collection does not yet exist:

```
Collection collectedDocs = Collection.GetCollection(datasrc,"collectionName");
```

Count Documents in a Collection: Size()

To count the number of documents with a collection instance invoke the **Size()** method:

```
Console.Writeln("Size = " + collectedDocs.Size());
```

Iterate through the Documents in a Collection: GetAll()

To iterate through all of the documents in a collection, invoke the **GetAll()** method, then sequence through that iterator:

```
DocumentList allDocuments = collectedDocs.GetAll();
allDocuments.ForEach(document=>
   Console.Out.WriteLine(Document.ToJSONString(document)));
```

Drop a Collection: Drop()

Dropping a collection will delete the extent of the collection and its metadata, except index definitions. This deletes all documents in the collection, but does not delete the collection itself. Any object references to the collection will stay valid and you can continue to work with it (e.g. store new documents). To drop (delete) a collection invoke the **Drop()** method:

```
collectedDocs.Drop();
```

5.3 .NET Documents and Transactions (Add/Create/Insert/Delete)

Document.Document class serves as a base class for both JSONArray and JSONObject. Documents are the main building blocks in the Document API. They can be saved, retrieved, updated etc. to/from Document using the Collections class.

Insert a Document: Insert()

Use the Collection.**Insert()** method to persist a Document object.

To insert a single document invoke the **Insert()** method:

```
Document doc = new JSONObject().put("Rank",1);
collectedDocs.Insert(doc);
```

To insert an array of elements as individual documents:

```
List<Document> topTwo = new ArrayList<>();
  topTwo.add(new JSONObject().put("Rank",1));
  topTwo.add(new JSONObject().put("Rank",2));
BulkResponse bulk = collectedDocs.Insert(topTwo);

System.out.println("Newly inserted document ids:");
for (String id : bulk.getIds())
    {
     System.out.print(id + " ");
    }
```

Retrieve a Document by id: Get()

To retrieve a document in a collection, invoke the **Get()** method and specify the document’s integer id:

```
Document doc = collectedDocs.Get("<id>");
```

where <id> is a quoted integer value (for example, `Get("23")`.)

Delete a Document by id: Remove()

To delete a document from a collection, invoke the **Remove()** method and specify the document’s integer id:

```
collectedDocs.Remove("<id>");
```

Update a Document by id: Replace()

To update a document in a collection, invoke the **Replace()** method and specify the document’s integer id and the new data value:

```
collectedDocs.Replace("<id>",new JSONObject().put("Rank",4));
```

Insert or Update a Document by id: Upsert()

To either insert a document (if it does not exist) or update the document (if it does exist), invoke the **Upsert()** method and specify the document’s integer id and the new data value:

```
Document upsertDoc = new JSONObject().put("Rank",1);
String upsertId = collectedDocs.Upsert(upsertDoc);
```

5.4 Index and Query .NET Collections

Create or change an Index: CreateIndex()

To create an index to documents in a collection, invoke the **CreateIndex()** method and specify an index name and an index request object. The index request object must be JSON encoded. It can be simple or complex. To create a simple index, set the property “key” to the JSON path that you want to index. The default index type is bitmap. The following example creates an index:

```
collectedDocs.CreateIndex("indexName","{\"key\":\"HomeCity\"}");
```

If the specified index name already exists, **CreateIndex()** will redefine this index with the new index definition.

Delete an Index: DropIndex()

To delete an index to documents in a collection, invoke the **DropIndex()** method and specify an existing index name:

```
collectedDocs.DropIndex("indexName");
```

If the specified index name does not exist, **DropIndex()** performs no operation and does not return an error.

Query a Collection: CreateQuery()

To query an instance of a collection, invoke the **CreateQuery()** method that specifies the SELECT query. You then issue an **execute()** method that executes the query and creates a cursor that allows you to iterate through the query result set:

```
Query querySlams = collectedDocs.CreateQuery(
           "select Rank,\"First Name\",\"Last Name\",Country,Points,
            \"Grand Slams\" from JSON_TABLE(('ATPList'),
            ('{\"columns\":\"ATPList[*]\"}'))");
Cursor slamCursor = querySlams.execute();
System.out.println("Executed query on ATPList - results follow: ");
List<Document> slamList = slamCursor.GetAll();
slamList.forEach(doc -> System.out.println(doc.toJSONString()));
```