Tutorial: Hello Spider!

# What is Spider?

Spider is one of the *storage services* offered by Doradus. Spider is flexible and easy-to-use, so it’s great for learning about the Doradus data model and query language. In this tutorial, we’ll create a simple Spider application, add some data, and perform some queries. Along the way, we’ll also explore the basics of the Doradus REST API.

# Creating a Spider Application

To store data in Doradus, we first need a database instance, which Doradus calls an *application*. Each application has a schema, which defines the application’s name and the storage service that will manage its data. Spider is the default storage service, so we don’t have to explicitly choose it. Furthermore, Spider allows dynamically-added tables and fields, so the only thing our schema needs is the application name. The simplest Spider application schema is the following XML document:

<application name="HelloSpider"/>

Doradus also supports JSON for all input and output messages. The equivalent JSON document is:

{"HelloSpider": null}

We’ll mostly use JSON in this tutorial: see the [Doradus Spider](https://github.com/dell-oss/Doradus/blob/master/docs/Doradus%20Spider%20Database.pdf) documentation for more details on JSON and XML message formats.

To create the application, we need to POST the schema to the system resource called \_applications. If we have the Doradus server running on our local machine and we have curl, we can use the following command:

curl -X POST -d '{"HelloSpider": null}' -H "content-type: application/json" http://localhost:1123/\_applications

This creates an application named HelloSpider, managed by the Spider storage service. We can verify that the application exists by getting its schema with this command:

curl http://localhost:1123/\_applications/HelloSpider

By default, output messages are returned in XML, so we’ll get the following schema:

<application name="HelloSpider">

<options>

<option name="AutoTables">true</option>

<option name="StorageService">SpiderService</option>

</options>

</application>

We can request JSON formatting either by adding the header accept: application/json to the request or by adding ?format=json to the URI:

curl http://localhost:1123/\_applications/HelloSpider?format=json

The response in JSON is shown below:

{"HelloSpider": {

"options": {

"AutoTables": "true",

"StorageService": "SpiderService"

}

}}

Notice that our application’s schema is automatically assigned default options: The StorageService option defaults to SpiderService, and the AutoTables option defaults to true. This option is described next.

# Adding Some Data

Doradus stores an application’s data in *tables*, whose members are called *objects*. An object’s values are stored in *fields*. When a Spider application’s AutoTables option is true, tables can be dynamically added. And, since Spider supports dynamically-added fields, we can objects with any fields we want.

Let’s create some “spider name” objects by creating a JSON-formatted file called spiders.data:

{"batch": {

"docs": [

{"doc": {"Name": "Tarantula"}},

{"doc": {"Name": "Alpha Romeo"}},

{"doc": {"Name": "Itsy Bitsy"}}

]

}}

The outer batch group contains an array called docs, which contains one doc group per object. In this example, each object assigns a single field called Name. We can add these objects to a table called Spiders using the REST URI: POST /*application*/*table*. Using curl:

curl -X POST -H "content-type: application/json" -d @spiders.data http://localhost:1123/HelloSpider/Spiders

When we send an update to the Spiders table, if it doesn’t exist, the Spider service creates it automatically. The objects are then added, setting the Name field for each one. Since objects must have an ID and we didn’t assign any, Doradus automatically assigns a unique ID. In case we need to know what these are right away, the IDs are returned in the REST response of our POST command:

{"batch-result": {

"status": "OK",

"has\_updates": "true",

"docs": [

{"doc": {"updated": "true", "status": "OK", "\_ID": "AAFNRJ9GsmZtp5adLQAA"}},

{"doc": {"updated": "true", "status": "OK", "\_ID": "AAFNRJ9GwmZtp5adLQAA"}},

{"doc": {"updated": "true", "status": "OK", "\_ID": "AAFNRJ9Gw2Ztp5adLQAA"}}

]

}}

The batch-result group contains a docs array with one doc group for each object. Each object’s ID is given in the \_ID field along with the status for that update. The updated flag and status field show that an update actually occurred. These might seem superfluous, but some updates might not actually change anything: if we set the \_ID field and the corresponding object already exists, it is updated. If none of the other fields we set are new or different from the existing object, nothing is changed, which is reflected in the REST response.

What happens if we fetch the application schema now? In JSON, it looks like this:

{"HelloSpider": {

"options": {

"AutoTables": "true",

"StorageService": "SpiderService"

},

"tables": {

"Spiders": {}

}

}}

This shows that our schema was automatically extended to declare the Spiders table without any explicitly-defined fields. This means the Name field we assigned is anonymous. Such fields are implicitly considered text and indexed using the full text analyzer. This means it is efficient to query such fields even though they are undeclared.

# Querying Objects

Doradus supports two kinds of queries: *object* queries, which select and return objects, and *aggregate* queries, which select objects and perform metric computations such as COUNT. Object queries use the general URI format:

GET /*application*/*table*/\_query?{params}

Where *application* and *table* specify which objects we want, \_query is the system resource for object queries, and {params} are the query parameters. The only required parameter is ‘q’, which is the Doradus Query Language (DQL) expression. The simplest object query is to return all objects using the query parameter q=\*:

curl -H "accept: application/json" http://localhost:1123/HelloSpider/Spiders/\_query?q=\*

(Note that because queries are GET commands and all parameters can be passed via the URI, browsers work fine as a query tool.) This returns a JSON message like this:

{"results": {

"docs": [

{"doc": {"Name": "Tarantula", "\_ID":"AAFNRJ9GsmZtp5adLQAA"}},

{"doc": {"Name": "Itsy Bitsy", "\_ID":"AAFNRJ9Gw2Ztp5adLQAA"}},

{"doc": {"Name": "Alpha Romeo", "\_ID": "AAFNRJ9GwmZtp5adLQAA"}}

]

}}

What we actually get is a *page* of results. If there were more than 100 objects, the response would get a *continue* token, which we can use to fetch the next page. We can control the page size with the ‘s’ parameter:

curl -H "accept: application/json" http://localhost:1123/HelloSpider/Spiders/\_query?q=\*&s=50

Here’s a query that finds all Spiders whose name contain the *term* “alpha”:

curl -H "accept: application/json" http://localhost:1123/HelloSpider/Spiders/\_query?q=Name:alpha

Note that the term alpha will select the object whose name is Alpha Romeo because text searches are case-insensitive. Object queries support several other parameters, and DQL supports a side range of query expressions. See the full [Doradus Spider](https://github.com/dell-oss/Doradus/blob/master/docs/Doradus%20Spider%20Database.pdf) documentation for details.

Now let’s try an aggregate query, which uses the system resource \_aggregate. The simplest aggregate query counts all objects using the ‘m’ parameter:

curl -H "accept: application/json" http://localhost:1123/HelloSpider/Spiders/\_aggregate?m=COUNT(\*)

Here’s a typical response for this query:

{"results": {

"aggregate": {"metric": "COUNT(\*)"},

"totalobjects": "3",

"value": "3"

}}

The metric function result is given in the field called value. The aggregate field echoes the query we requested, and the totalobjects field indicates how many objects were scanned by this query. (For some queries, value and totalobjects can be different.) Like object queries, aggregate queries support several parameters and a wide range of functions, so see the full documentation for details.

# Explicitly Defining Fields

When might we want to explicitly define fields in a Spider schema? There are things we can change when we explicitly define a field: (1) the field’s type, (2) how it is indexed, and (3) the field’s cardinality. By default, the *type* of an undefined field is considered text, it is indexed with a full text *analyzer*, and it is considered *single-valued*. Let’s demonstrate by modifying the schema of our HelloSpider application.

With this change, we also demonstrate that schema updates use *replacement* semantics. This means the old schema is completely replaced by the new one. So, if we omit a previously-defined table, it is deleted, along with its data. Any new tables are added, and new fields in an existing table are added. Here’s the schema that will replace our previous HelloSpider schema:

{"HelloSpider": {

"options": {

"AutoTables": "true",

"StorageService":"SpiderService"

},

"tables": {

"Movies": {

"fields": {

"Budget": {"type": "integer", "analyzer": "IntegerAnalyzer"},

"Cancelled": {"type": "boolean", "analyzer":"BooleanAnalyzer"},

"Director": {"type":"text", "analyzer": "TextAnalyzer"},

"Leads": {"type": "text", "collection": "true", "analyzer": "TextAnalyzer"},

"Name": {"type": "text", "analyzer": "TextAnalyzer"},

"ReleaseDate": {"type": "timestamp", "analyzer": "DateAnalyzer"}

}

}

}

}}

This schema drops the Spiders table and adds a new one called Movies. Six fields are declared, showing some of the scalar types supported: integer, boolean, text, and timestamp. For Leads, the collection property is set to true: this makes the field *multi-valued*, allowing it to hold a set of values per object. Each field’s analyzer is also declared, though in this example the default analyzer is declared for each field. Other analyzers such as the OpaqueTextAnalyzer and NullAnalyzer are available depending on the field type. See the full documentation for more details on analyzers.

If the schema above resides in a file called updated-schema.json, we can apply it with the following PUT command:

curl -X PUT -H "content-type: application/json" -d @updated-schema.json http://localhost:1123/\_applications/HelloSpider

Now let’s insert some data about Spider-Man movies. This time, we’ll assign the object IDs. See the full batch data in the sample [HelloWorld](https://github.com/dell-oss/Doradus/blob/master/doradus-client/src/main/java/com/dell/doradus/client/utils/HelloSpider.java) application included with the doradus-client package. But here’s a sample:

{"batch": {

"docs": [

{"doc": {

"\_ID": "Spidy1",

"Name": "Spider-Man",

"ReleaseDate": "2002-05-03",

"Cancelled": false,

"Director": "Sam Raimi",

"Leads": {"add": ["Tobey Maguire", "Kirsten Dunst", "Willem Dafoe"]},

"Budget": 240000000

}},

{"doc": {

"\_ID": "Spidy2",

"Name": "Spider-Man 2",

"ReleaseDate": "2004-06-04",

"Cancelled": false,

"Director": "Sam Raimi",

"Leads": {"add": ["Tobey Maguire", "Kirsten Dunst", "Alfred Molina"]},

"Budget": 200000000

}},

...

]

}}

To finish off this tutorial, let’s perform an aggregate query that computes the average budget of the six Spider-Man movies based on our loaded data:

curl 'http://localhost:1123/HelloSpider/Movies/\_aggregate?m=AVERAGE(Budget)&format=json'

This produces the result:

{"results": {

"aggregate": {"metric": "AVERAGE(Budget)"},

"totalobjects": "6",

"value": "232000000"

}}

In other words, $232 million per movie on average. It’s no wonder that 4 of the 25 most-expensive movies are Spider-Man!