

HCL Informix 15.0.0

JSON Compatibility Guide



Contents

Chapter 1. HCL Informix® JSON compatibility.....	3	Index.....	152
About the Informix® JSON compatibility.....	3		
Getting started with Informix® JSON.....	4		
Requirements for JSON compatibility.....	5		
Support for dots in field names.....	5		
Manipulate BSON data with SQL statements.....	6		
Wire listener.....	7		
Configuring the wire listener for the first time.....	8		
The wire listener configuration file.....	9		
Wire listener command line options.....	42		
User authentication with the wire listener.....	47		
Encryption for wire listener communications.....	52		
High availability support in the wire listener.....	54		
JSON data sharding.....	54		
Preparing shard servers.....	55		
Shard-cluster definitions for distributing data.....	56		
Shard cluster management.....	60		
MongoDB API.....	65		
Getting Started with Informix®'s MongoDB Solution.....	65		
MongoDB to Informix® term mapping.....	68		
Language drivers.....	69		
Command utilities and tools.....	69		
Collection methods.....	69		
Index creation.....	71		
Database commands.....	73		
Informix® JSON commands.....	82		
Running Informix queries through the MongoDB API.....	91		
Operators.....	95		
Informix® REST API	103		
About the Informix® REST API.....	103		
Getting started with REST.....	104		
REST API syntax.....	115		
Running SQL passthrough queries through REST.....	123		
Informix® REST API			
Troubleshooting the REST API server.....	124		
Manage time series through the wire listener.....	125		
Creating a time series through the wire listener.....	126		
Example queries of time series data by using the wire listener.....	136		
Aggregate or slice time series data.....	141		
Loading time series data with the MQTT protocol.....	148		
Troubleshooting Informix® JSON compatibility.....	150		

Chapter 1. HCL Informix® JSON compatibility

The Informix JSON Compatibility Guide describes how applications that use the popular JSON-oriented query language created by MongoDB can interact with data stored in HCL Informix®.

This information is intended for application programmers.

About the Informix® JSON compatibility

You can combine relational and JSON data into a single query by using the Informix® JSON compatibility features.

Applications that use the JSON-oriented query language can interact with relational and non-relational data that is stored in Informix® databases by using the wire listener. The Informix® database server also provides built-in JSON and BSON (binary JSON) data types.

You have the following options for accessing relational tables, including time series tables and tables with WebSphere® MQ data, and JSON collections:

SQL API

You can insert, update, and query data relational tables through the SQL language and standard ODBC, JDBC, .NET, OData, and other clients.

You can access JSON collections through direct SQL access and the JDBC driver. You can use the SQL BSON processing functions to convert JSON collections to relational data types for use with ODBC, .NET, OData, and other clients.

MongoDB API

You can insert, update, and query data in relational tables and JSON collections through MongoDB APIs for Java™, JavaScript™, C++, C#, Python, and other clients.

REST API

You can insert, update, and query data relational tables and JSON collections through the driverless REST API. You can run command documents that include MongoDB API commands or SQL queries. You can use the REST API to load time series data from sensor devices.

MQTT protocol

You can insert JSON data into relational tables and JSON collections through the MQTT protocol for Java™, JavaScript™, C++, PHP, Python, Ruby, and other clients. You can use the MQTT protocol to load time series data from sensor devices.

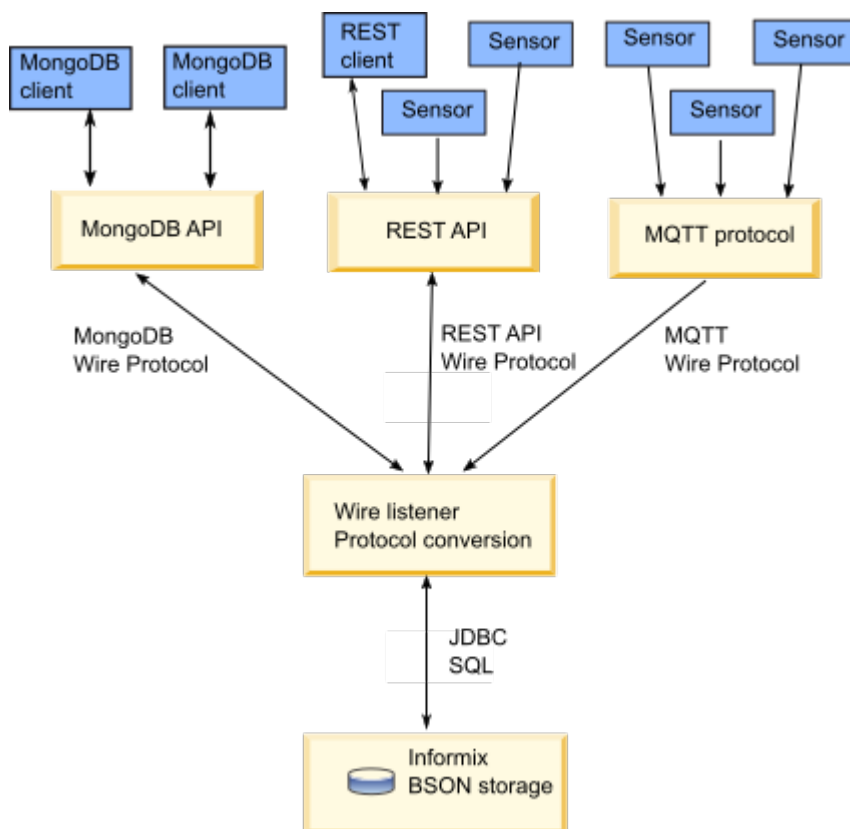
The JSON document format provides a way to transfer object information in a way that is language neutral, similar to XML. Language-neutral data transmission is a requirement for working in a web application environment, where data comes from various sources and software is written in various languages. With Informix®, you can choose which parts of your application data are better suited unstructured, non-relational storage, and which parts are better suited in a traditional relational framework.

You can enable dynamic scaling and high-availability for data-intensive applications by taking the following steps:

- Define a sharded cluster to easily add or remove servers as your requirements change.
- Use shard keys to distribute subsets of data across multiple servers in a sharded cluster.
- Query the correct servers in a sharded cluster and return the consolidated results to the client application.
- Use secondary servers (similar to subordinates in MongoDB) in the sharded cluster to maximize availability and throughput. Secondary servers also have update capability.

You can choose to authenticate users through the wire listener or in the database server.

You can configure multiple wire listeners for multiple client protocols. The following illustration shows the architecture of the wire listeners and the database server.



Getting started with Informix® JSON

You can begin using the Informix® JSON features after installing Informix®.

If you create the Informix® server instance as a part of your installation, the wire listener is automatically started and connected to the MongoDB API and the database server with the default operational instance. You can use the MongoDB shell and any of the standard MongoDB command utilities and tools. To use the REST API or the MQTT protocol, you must modify the default configuration.

If you create the Informix® server instance outside of the installation process, you must configure and start the wire listener manually.

Requirements for JSON compatibility

Informix® JSON compatibility has specific software dependencies and database server requirements.

Java requirements

Informix® JSON compatibility requires HCL Informix® version 12.10.xC2 or later, with the component, which enables services that use Java™.

You must use a supported [Java™ runtime environment on page](#) . Java version 1.8 is recommended.

MongoDB version

Informix® JSON compatibility is based on MongoDB version 2.4, 2.6, and 3.0.

You set the version of the MongoDB API that the wire listener uses by setting the `mongo.api.version` parameter in the wire listener configuration file. The MongoDB API version affects the type of authentication that you can use. For example, MongoDB version 3.0 supports the MongoDB SCRAM-SHA-1 authentication method, but does not support database server authentication or connections with the REST API.

Database server requirements

JSON and BSON data is stored in sbspaces. You can specify the sbspace for JSON and BSON storage in the PUT clause of the INSERT statement. However, you must set a default sbspace with the SBSPACENAME configuration parameter. When you insert JSON or BSON data that exceeds 4 K in size, the data is temporarily saved in the default sbspace for processing before being saved in the sbspace that you specified.

Support for dots in field names

Unlike MongoDB, which does not allow dots, (.), in JSON or BSON field names, HCL Informix® conforms to the JSON standard and allows dots. For example: `{"user.fn" : "Jake"}`. However, you cannot run a query or an operation directly on a field that has a dot in its name. In queries, a dot in between field names indicates a hierarchy.

Here the rules of using field names with dots in them with Informix®:

- You can insert a document that has a field name with a dot in it. You do not get an error.
- You cannot use a field name with a dot in it in a query or operation. Informix® ignores the field. The query does not return the matching document. The operation does not affect the value of the field.
- You can return a document that includes a field name with a dot in it by querying on a field name in the same document that does not have a dot in it.

Allowing dots in field names is useful when you do not have control over the field names because your data comes from external sources, for example, the Google API. You still want to store those documents in your database, even though some fields might have dots in their names.

The following examples to illustrate how dots in field names work in Informix®. The table name is **tab1** and the column that contains JSON data is named **data**.

Suppose that you have the following document:

```
{user : {fn : "Bob", ln : "Smith"}, "user.fn" : "Jake"}
```

You run the following statement to update a field:

```
SELECT data::json FROM tab1 WHERE BSON_UPDATE(data, '$set : {"user.fn" :  
    "John:}}');
```

The following document is returned:

```
{user : {fn : "John", ln : "Smith"}, "user.fn" : "Jake"}
```

The value of the **fn** field that is in a subdocument to the **user** field is updated. The value of the **user.fn** field is not updated, but the value is returned. You cannot update the value of a field with a dot in its name, but you can retrieve the value.

Suppose that you have the following document:

```
{"user.firstname" : "Jake"}
```

You run this query to return the value of the **user.firstname** field:

```
SELECT data::json FROM tab1 WHERE BSON_KEYS_EXIST(data,  
    "user.firstname");
```

No documents are returned.

If you have documents where all the fields have dots in their names, you must run a query to return all documents in the database to see them: for example:

```
SELECT data::json FROM tab1;
```

Manipulate BSON data with SQL statements

As an alternative to using the MongoDB API, you can use Informix® SQL to manipulate BSON data. However, if you plan to query JSON and BSON data through the wire listener, you must create your database objects, such as collections and indexes, through the wire listener. You can use SQL statements to query JSON and BSON data whether you created your database objects through the wire listener or with SQL statements.

You might have an existing application on relational tables that uses SQL to access the data, but you want to add BSON data to your database. You can create a table with a BSON column, insert the data, and manipulate the data with SQL statements. BSON documents that you insert through SQL statements or Informix® utilities do not contain generated ObjectId field-value pairs or other MongoDB metadata.

Alternatively, you might use a MongoDB client for daily data processing, but need the querying capabilities of SQL for data analysis. For example, you can use SQL statements to join tables that have BSON columns with other tables based on BSON field values. You can create views that have columns of BSON field values. You can run warehouse queries on BSON data with . If you have time series data, you can use the corresponding specialized SQL routines to analyze the data.

You can use BSON processing functions to manipulate BSON data in SQL statements. The BSON value functions convert BSON field values to standard SQL data types, such as INTEGER and LVARCHAR. The BSON_GET and BSON_UPDATE functions manipulate field-value pairs. You can convert all or part of a relational table to a BSON document with the genBSON function.

Example: Using SQL to query a collection

In the following example, a table that is named **people** is created with **names** and **ages** fields that are inserted by using the interactive JavaScript™ shell interface to MongoDB:

```
db.createCollection("people");
db.people.insert({"name": "Anne", "age": 31});
db.people.insert({"name": "Bob", "age": 39});
db.people.insert({"name": "Charlie", "age": 29});
```

For SQL statements, the table name is **people** and the BSON column name is **data**. When you create a collection through a MongoDB API command, the name of the BSON column is set to **data**.

The following statement selects the **name** and **age** fields with dot notation and displays the results in a readable format by casting the results to JSON:

```
> SELECT data.name::JSON, data.age::JSON FROM people;

(expression)  {"name": "Anne"}
(expression)  {"age": 31}

(expression)  {"name": "Bob"}
(expression)  {"age": 39}

(expression)  {"name": "Charlie"}
(expression)  {"age": 29}

3 row(s) retrieved.
```

Wire listener

The wire listener is a mid-tier gateway server that enables communication between MongoDB clients and the Informix® database server.

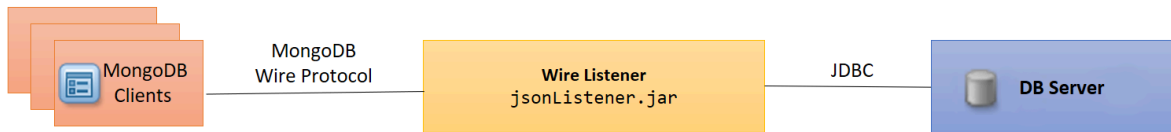
The wire listener is a Java™ application and is provided as an executable JAR file as part of the Informix® APIs package. The JAR file provides access to the MongoDB API.

MongoDB API access

The wire listener implements the MongoDB Wire Protocol. This allows you to connect MongoDB applications and client drivers to the Informix® database through the wire listener. The MongoDB applications send MongoDB operations and commands to the wire listener, which automatically translates those commands to SQL which it runs against the Informix® database using JDBC.

You can use the MongoDB API to access Informix® JSON/BSON collections, relational tables, or TimeSeries tables.

The [wire listener configuration properties file on page 9](#) defines every operational characteristic. By default, when you create a database or a table through the wire listener, automatic location and fragmentation are enabled. Databases are stored in the dbspace that is chosen by the server. Tables are fragmented among dbspaces that are chosen by the server. More fragments are added when tables grow.



Configuring the wire listener for the first time

You must configure the wire listener by specifying an authorized user and customizing the wire listener configuration file.

Before you begin

The wire listener JAR file is included in the database server installation at \$INFORMIXDIR/bin/jsonListener.jar.

To configure the wire listener for the first time:

1. Choose an authorized user.

An authorized user is required in wire listener connections to the database server. The authorized user must have access to the databases and tables that are accessed through the wire listener.

Choose from:

- **Windows™**: Specify an operating system user.
- **UNIX™ or** : Specify an operating system user or a database user. For example, here is the command to create a database user in UNIX™ or :

```
CREATE USER userID WITH PASSWORD 'password' ACCOUNT unlock PROPERTIES  
USER daemon;
```

2. **Optional**: If you want to shard data, grant the user REPLICATION privilege by running the admin or task SQL administration API command with the grant admin argument.

Example

For example:

```
EXECUTE FUNCTION task('grant admin','userID','replication');
```

3. Create a wire listener configuration file in \$INFORMIXDIR/etc with the .properties file extension. You can use the \$INFORMIXDIR/etc/jsonListener-example.properties file as a template.
For more information, see [The wire listener configuration file on page 9](#).
4. Customize the wire listener configuration file to your needs.
To include parameters in the wire listener, uncomment the row and customize the parameter. The **url** parameter is required. All other parameters are optional.



Tip: Review the defaults for the following parameters and verify that they are appropriate for your environment: **mongo.api.version**, **authentication.enable**, **listener.type**, **listener.port**, and **listener.hostName**.

5. If you are using a Dynamic Host Configuration Protocol (DHCP) on your IPv6 host, you must verify that the connection information between JDBC and Informix® is compatible.

For example, you can connect from the IPv6 host through an IPv4 connection by using the following steps:

- a. Add a server alias to the DBSERVERALIASES configuration parameter for the wire listener on the local host.

Example

For example: `ol_informix1210`.

- b. Add an entry to the `sqlhosts` file for the database server alias to the loopback address `127.0.0.1`.

Example

For example:

```
ol_informix1210  onsocket 127.0.0.1 9090
```

- c. In the wire listener configuration file, update the **url** entry with the wire listener alias.

Example

For example:

```
url=jdbc:informix-sqli://localhost:9090/sysmaster:
INFORMIXSERVER=ol_informix1210;
```

What to do next

Start the wire listener.

The wire listener configuration file

The settings that control the wire listener and the connection between the client and database server are set in the wire listener configuration file.

The default name for the configuration file is `$INFORMIXDIR/etc/jsonListener.properties`. You can rename this file, but the suffix must be `.properties`.

If you create a server instance during the installation process, a configuration file that is named `jsonListener.properties` is automatically created with default properties, otherwise you must manually create the configuration file. You can use the `$INFORMIXDIR/etc/jsonListener-example.properties` file as a template.

In the configuration file that is created during installation, and in the template file, all of the parameters are commented out by default. To enable a parameter, you must uncomment the row and customize the parameter.



Important: The `url` parameter is required. All other parameters are optional.

- Required
 - [url on page 12](#)
- Setup and configuration
 - [documentIdAlgorithm on page 13](#)
 - [include on page 14](#)
 - [listener.onException on page 14](#)
 - [listener.hostName on page 15](#)
 - [listener.port on page 15](#)
 - [listener.timezone on page 15](#)
 - [listener.type on page 16](#)
 - [response.documents.count.default on page 16](#)
 - [response.documents.count.maximum on page 16](#)
 - [response.documents.size.maximum on page 16](#)
 - [sharding.enable on page 16](#)
 - [sharding.parallel.query.enable on page 17](#)
- Command and operation configuration
 - [collection.informix.options on page 17](#)
 - [command.listDatabases.sizeStrategy on page 18](#)
 - [update.client.strategy on page 18](#)
 - [update.mode on page 19](#)
- Database resource management
 - [database.buffer.enable on page 19](#)
 - [database.create.enable on page 20](#)
 - [database.dbspace on page 20](#)
 - [database.locale.default on page 20](#)
 - [database.log.enable on page 20](#)
 - [database.onException.errorCodes on page 20](#)
 - [dbspace.strategy on page 21](#)
 - [fragment.count on page 21](#)
 - [jdbc.afterNewConnectionCreation on page 22](#)
- MongoDB compatibility
 - [compatible.maxBsonObjectSize.enable on page 22](#)
 - [mongo.api.version on page 22](#)
 - [update.one.enable on page 22](#)
- Performance
 - [delete.preparedStatement.cache.enable on page 23](#)
 - [insert.batch.enable on page 23](#)
 - [insert.batch.queue.enable on page 24](#)
 - [insert.batch.queue.flush.interval on page 24](#)

- `index.cache.enable` on page 24
- `index.cache.update.interval` on page 25
- `insert.preparedStatement.cache.enable` on page 25
- `preparedStatement.cache.enable` on page 25
- `preparedStatement.cache.size` on page 25
- **Security**
 - `authentication.enable` on page 25
 - `authentication.localhost.bypass.enable` on page 26
 - `command.denylist` on page 26
 - `db.authentication` on page 26
 - `listener.admin.ipAddress` on page 27
 - `listener.authentication.timeout` on page 27
 - `listener.http.accessControlAllowCredentials` on page 27
 - `listener.http.accessControlAllowHeaders` on page 27
 - `listener.http.accessControlAllowMethods` on page 28
 - `listener.http.accessControlAllowOrigin` on page 28
 - `listener.http.accessControlExposeHeaders` on page 29
 - `listener.http.accessControlMaxAge` on page 29
 - `listener.http.headers` on page 29
 - `listener.http.headers.size.maximum` on page 29
 - `listener.rest.cookie.domain` on page 29
 - `listener.rest.cookie.httpOnly` on page 30
 - `listener.rest.cookie.length` on page 30
 - `listener.rest.cookie.name` on page 30
 - `listener.rest.cookie.path` on page 30
 - `listener.rest.cookie.secure` on page 30
 - `listener.ssl.algorithm` on page 31
 - `listener.ssl.ciphers` on page 31
 - `listener.ssl.enable` on page 31
 - `listener.ssl.key.alias` on page 31
 - `listener.ssl.key.password` on page 32
 - `listener.ssl.keyStore.file` on page 32
 - `listener.ssl.keyStore.password` on page 32
 - `listener.ssl.keyStore.type` on page 32
 - `listener.ssl.protocol` on page 32
 - `security.sql.passthrough` on page 33
- **Wire listener resource management**
 - `cursor.idle.timeout` on page 33
 - `listener.connectionPool.closeDelay.time` on page 33
 - `listener.connectionPool.closeDelay.timeUnit` on page 33
 - `listener.idle.timeout` on page 33
 - `listener.idle.timeout.minimum` on page 34

- [listener.input.buffer.size](#) on page 34
- [listener.memoryMonitor.enable](#) on page 34
- [listener.memoryMonitor.allPoint](#) on page 34
- [listener.memoryMonitor.diagnosticPoint](#) on page 35
- [listener.memoryMonitor.zeroPoint](#) on page 35
- [listener.output.buffer.size](#) on page 35
- [listener.pool.admin.enable](#) on page 35
- [listener.pool.keepAliveTime](#) on page 35
- [listener.pool.queue.size](#) on page 36
- [listener.pool.size.core](#) on page 36
- [listener.pool.size.maximum](#) on page 36
- [listener.socket.accept.timeout](#) on page 36
- [listener.socket.read.timeout](#) on page 36
- [pool.connections.maximum](#) on page 37
- [pool.idle.timeout](#) on page 37
- [pool.idle.timeunit](#) on page 37
- [pool.lenient.return.enable](#) on page 38
- [pool.lenient.dispose.enable](#) on page 38
- [pool.semaphore.timeout](#) on page 38
- [pool.semaphore.timeunit](#) on page 38
- [pool.service.interval](#) on page 39
- [pool.service.threads](#) on page 39
- [pool.service.timeunit](#) on page 39
- [pool.size.initial](#) on page 40
- [pool.size.minimum](#) on page 40
- [pool.size.maximum](#) on page 40
- [pool.type](#) on page 40
- [pool.typeMap.strategy](#) on page 41
- [response.documents.size.minimum](#) on page 41
- [timeseries.loader.connections](#) on page 41

Required parameter

You must configure the **url** parameter before using the wire listener.

url

This required parameter specifies the host name, port number, user ID, and password that are used in connections to the database server.

You must specify the **sysmaster** database in the **url** parameter. That database is used for administrative purposes by the wire listener.

```
url = jdbc : informix-sqli : // hostname : portnum / sysmaster : [ USER = userid ; PASSWORD = password NONCE = value ]
```

You can include additional JDBC properties in the **url** parameter such as INFORMIXCONTIME, INFORMIXCONRETRY, LOGINTIMEOUT, and IFX_SOC_TIMEOUT. For a list of Informix® environment variables that are supported by the JDBC driver, see [Informix® environment variables with the HCL Informix® JDBC Driver on page](#).

hostname:portnum

The host name and port number of your computer. For example, `localhost:9090`.

USER=userid

This optional attribute specifies the user ID that is used in connections to the Informix® database server. If you plan to use this connection to establish or modify collection shards by using the Informix® sharding capability, the specified user must be granted the REPLICATION privilege group access.

If you do not specify the user ID and password, the JDBC driver uses operating system authentication and all wire listener actions are run by using the user ID and password of the operating system user who runs the wire listener start command.

PASSWORD=password

This optional attribute specifies the password for the specified user ID.

NONCE=value

This optional attribute specifies a 16-character value that consists of numbers and the letters a, b, c, d, e, and f. This property triggers password encoding when a pluggable authentication module is configured for the wire listener. Applicable only if the `db.authentication` parameter is set to `informix-mongodb-cr`.

Setup and configuration

These parameters provide setup and configuration options.

documentIdAlgorithm

This optional parameter determines the algorithm that is used to generate the unique Informix® identifier for the ID column that is the primary key on the collection table. The `_id` field of the document is used as the input to the algorithm. The default value is `documentIdAlgorithm=ObjectId`.

```
documentIdAlgorithm= { ObjectId | SHA-1 | SHA-256 | SHA-512 }
```

ObjectId

Indicates that the string representation of the ObjectId is used if the `_id` field is of type ObjectId; otherwise, the MD5 algorithm is used to compute the hash of the contents of the `_id` field.

- The string representation of an ObjectId is the hexadecimal representation of the 12 bytes that comprise an ObjectId.
- The MD5 algorithm provides better performance than the secure hashing algorithms (SHA).

ObjectId is the default value and it is suitable for most situations.



Important: Use the default unless a unique constraint violation is reported even though all documents have a unique `_id` field. In that case, you might need to use a non-default algorithm, such as SHA-256 or SHA-512.

SHA-1

Indicates that the SHA-1 hashing algorithm is used to derive an identifier from the `_id` field.

SHA-256

Indicates that the SHA-256 hashing algorithm is used to derive an identifier from the `_id` field.

SHA-512

Indicates that the SHA-512 hashing algorithm is used to derive an identifier from the `_id` field. This option generates the most unique values, but uses the most processor resources.

include

This optional parameter specifies the properties file to reference. The path can be absolute or relative. For more information, see [Running multiple wire listeners on page 45](#).

```
include= properties_file
```

listener.onException

This optional parameter specifies an ordered list of actions to take if an exception occurs that is not handled by the processing layer.

```
listener.onException = { reply | closeSession | shutdownListener }
```

reply

When an unhandled exception occurs, reply with the exception message. This is the default value.

closeSession


When an unhandled exception occurs, close the session.

shutdownListener

When an unhandled exception occurs, shut down the wire listener.

listener.hostName

This optional parameter specifies the host name of the wire listener. The host name determines the network adapter or interface that the wire listener binds the server socket to.

 **Tip:** If you enable the wire listener to be accessed by clients on remote hosts, turn on authentication by using the **authentication.enable** parameter.

```
listener.hostName= { localhost | hostname | * }
```

localhost

Bind the wire listener to the localhost address. The wire listener is not accessible from clients on remote machines. This is the default value.

hostname


The host name or IP address of host machine where the wire listener binds to.

*

The wire listener can bind to all interfaces or addresses.

listener.port


This optional parameter specifies the port number to listen on for incoming connections from clients. This value can be overridden from the command line by using the **-port** argument. The default value is 27017.

 **Important:** If you specify a port number that is less than 1024, the user that starts the wire listener might require additional operating system privileges.

```
listener.port= { 27017 | port_number }
```

listener.timezone

This parameter specifies the timezone of the listener java JVM. This will override any system or user configured default timezone. The timezone property affects the timezone of date values that are used outside of BSON documents.

 **Important:** It is recommended that the listener timezone be set to UTC (or GMT). You should change this property only if you are using the listener to interact with relational tables that store dates in a timezone other than UTC/GMT.

Possible values: UTC, GMT, GMT+1, GMT+2, GMT-1, GMT-2, EST, CST, etc. Set this property to null to use the system's default timezone.

```
listener.timezone = { UTC | timezone }
```

listener.type

This optional parameter specifies the type of wire listener to start.

```
listener.type= { mongo | rest | mqtt }
```

mongo

Connect the wire listener to the MongoDB API. This is the default value.

rest

Connect the wire listener to the REST API.

mqtt

Connect the wire listener to the MQTT protocol.

response.documents.count.default

This optional parameter specifies the default number of documents in a single response to a query. The default value is 100.

```
response.documents.count.default = { 100 | default_docs }
```

response.documents.count.maximum

This optional parameter specifies the maximum number of documents in a single response to a query. The default value is 10000.

```
response.documents.count.maximum= { 10000 | max_docs }
```

response.documents.size.maximum

This optional parameter specifies the maximum size, in bytes, of all documents in a single response to a query. The default value is 1048576.

```
response.documents.size.maximum= { 1048576 | max_size }
```

sharding.enable

This optional parameter indicates whether to enable the use of commands and queries on sharded data.

```
sharding.enable= { false | true }
```

false

Do not enable the use of commands and queries on sharded data. This is the default value.

true

Enable the use of commands and queries on sharded data.

sharding.parallel.query.enable

This optional parameter indicates whether to enable the use of parallel sharded queries. Parallel sharded queries require that the SHARD_ID configuration parameter be set to unique IDs on all shard servers. The sharding.enable parameter must also be set to true.

```
sharding.parallel.query.enable = { false | true }
```

false

Do not enable parallel sharded queries. This is the default value.

true

Enable parallel sharded queries.

Command and operation configuration

These parameters provide configuration options for JSON commands and operations.

collection.informix.options

This optional parameter specifies which table options for shadow columns or auditing to use when creating a JSON collection.

```
collection.informix.options=[ { [ { | "audit" | "crcols" | "erkey" | "replcheck" | "vercols" } ] } ]
```

audit

Use the AUDIT option of the CREATE TABLE statement to create a table to be included in the set of tables that are audited at the row level if selective row-level is enabled.

crcols

Use the CRCOLS option of the CREATE TABLE statement to create two shadow columns that uses for conflict resolution.

erkey

Use the ERKEY option of the CREATE TABLE statement to create the ERKEY shadow columns that uses for a replication key.

replcheck

Use the REPLCHECK option of the CREATE TABLE statement to create the **ifx_replcheck** shadow column that uses for consistency checking.

vercols

Use the VERCOLS option of the CREATE TABLE statement to create two shadow columns that Informix® uses to support update operations on secondary servers.

command.listDatabases.sizeStrategy

This optional parameter specifies a strategy for calculating the size of your database when the MongoDB `listDatabases` command is run. The `listDatabases` command estimates the size of all collections and collection indexes for each database. However, relational tables and indexes are excluded from this size calculation.



Important: The MongoDB `listDatabases` command performs expensive and CPU-intensive computations on the size of each database in the database server instance. You can decrease the expense by using the **command.listDatabases.sizeStrategy** parameter.

```
command.listDatabases.sizeStrategy= { { estimate | { estimate:n } | compute | none | perDatabaseSpace } }
```

estimate

Estimate the size of the database by sampling documents in every collection. This is the default value. This strategy is the equivalent of `{estimate: 1000}`, which takes a sample size of 0.1% of the documents in every collection. This is the default value.

```
command.listDatabases.sizeStrategy=estimate
```

estimate: n

Estimate the size of the database by sampling one document for every *n* documents in every collection. The following example estimates the collection size by using sample size of 0.5% or 1/200th of the documents:

```
command.listDatabases.sizeStrategy={estimate:200}
```

compute

Compute the exact size of the database.

```
command.listDatabases.sizeStrategy=compute
```

none

List the databases but do not compute the size. The database size is listed as 0.

```
command.listDatabases.sizeStrategy=none
```

perDatabaseSpace

Calculate the size of a database by adding the sizes for all dbspaces, sbspaces, and blobspaces that are assigned to the tenant database.



Important: The **perDatabaseSpace** option applies only to tenant databases that are created by the multi-tenancy feature.

update.client.strategy

This optional parameter specifies the method that is used by the wire listener to send updates to the database server. When the wire listener does the update processing, it queries the server for the existing document and then updates the document.

```
update.client.strategy= { { updatableCursor | deleteInsert } }
```

updatableCursor

Updates are sent to the database server by using an updatable cursor. This is the default value.

deleteInsert

The original document is deleted when the updated document is inserted.



Important: If the collection is sharded, you must use this method.

update.mode

This optional parameter determines where document updates are processed. The default value is

```
update.mode=mixed.
```

```
update.mode= { { mixed | client } }
```

client

Use the wire listener to process updates. You must use this mode if you enable sharding and want to allow the updating of shard key field values.

mixed

Attempt to process updates on the database server first, then fallback to the wire listener. This is the default value.

Database resource management

These parameters provide database resource management options.

database.buffer.enable



Prerequisite: `database.log.enable=true`

This optional parameter indicates whether to enable buffered logging when you create a database by using the wire listener.

```
database.buffer.enable = { true | false }
```

true

Enable buffered logging. This is the default value.

false

Do not enable buffered logging.

database.create.enable

This optional parameter indicates whether to enable the automatic creation of a database, if a database does not exist.

```
database.create.enable = { true | false }
```

true

If a database does not exist, create a database. This is the default value.

false

If a database does not exist, do not create a database. With this option, you can access only existing databases.

database.dbospace



Prerequisite: `dbospace.strategy=fixed`

This optional parameter specifies the name of the dbospace databases that are created. The default value is `database.dbospace=rootdbs`.

```
database.dbospace = { rootdbs | dbospace_name }
```

database.locale.default

This optional parameter specifies the default locale to use when a database is created by using the wire listener. The default value is `en_US.utf8`.

```
database.locale.default = { en_US.utf8 | locale }
```

database.log.enable

This optional parameter indicates whether to create databases that are enabled for logging.

```
database.log.enable = { true | false }
```

true

Create databases that are enabled for logging. This is the default value. Use the **database.buffer.enable** parameter to enable buffered logging.

false

Do not create databases that are enabled for logging.

database.onException.errorCodes

A JSON document describing what actions to take on specific database error codes. Each action should be followed by an array of the database's integer error codes that should trigger the specified action.

closePools

Error codes that should trigger the listener to close the existing connection pools.

disposeOfConnections

Error codes that indicate the current connection is stale and should be disposed of.

removeCollectionFromCache

Error codes that indicate that the listener's currently cached information about the collection is stale and should be refreshed.

reprepareStatement

Error codes that indicate that the prepared statement should be re-prepared.

retryStatement

Error codes that indicate that an insert, update, delete or query statement should be retried once before the error/result is returned to the client.

These lists of error codes are the default values for each statement, and can be changed or added as you desire. For example:

```
database.onException.errorCodes={
  "closePools":[-79716, -79730, -79735],
  "disposeOfConnection":[-349, -79716, -79730, -79735],
  "removeCollectionFromCache":[-710, -206],
  "reprepareStatement":[-208, -267, -285, -79716],
  "retryStatement":[]
}
```

dbspace.strategy

This optional parameter specifies the strategy to use when determining the location of new databases, tables, and indexes.

```
dbspace.strategy = { autolocate | fixed }
```

autolocate

The database server automatically determines the dbspace for the new databases, tables, and indexes. This is the default value.

fixed

Use a specific dbspace, as specified by the **database.dbspace** property.

fragment.count

This optional parameter specifies the number of fragments to use when creating a collection. If you specify 0, the database server determines the number of fragments to create. If you specify a *fragment_num* greater than 0, that number of fragments are created when the collection is created. The default value is 0.

```
fragment.count = { 0 | fragment_num }
```

jdbc.afterNewConnectionCreation

This optional parameter specifies one or more SQL commands to run after a new connection to the database is created.

```
jdbc.afterNewConnectionCreation = [ " sql_command " ]
```

For example, to accelerate queries run through the wire listener by using :

```
jdbc.afterNewConnectionCreation=["SET ENVIRONMENT USE_DWA 'ACCELERATE ON'"]
```

MongoDB compatibility

These parameters provide options for MongoDB compatibility.

compatible.maxBsonObjectSize.enable

This optional parameter indicates whether the maximum BSON object size is compatible with MongoDB.



Tip: If you insert a BSON document by using an SQL operation, Informix® supports a maximum document size of 2 GB.

```
compatible.maxBsonObjectSize.enable = { false | true }
```

false

Use a maximum document size of 256 MB with the wire listener. This is the default value.

true

Use a maximum document size of 16 MB. The maximum document size for MongoDB is 16 MB.

mongo.api.version

This optional parameter specifies the MongoDB API version with which the wire listener is compatible. The version affects authentication methods as well as MongoDB commands.

```
mongo.api.version = { 2.4 | 2.6 | 3.0 | 3.2 | 3.4 | 3.6 | 4.0 | 4.2 }
```



Note: 2.4 is the default value.



Important: Do not set mongo.api.version=3.0 or higher if you want to use the REST API with MongoDB style authentication. See [User authentication with the wire listener on page 47](#).

update.one.enable

This optional parameter indicates whether to enable support for updating a single JSON document.



Important: The **update.one.enable** parameter applies to JSON collections only. For relational tables, the MongoDB multi-parameter is ignored and all documents that meet the query criteria are updated.

```
update.one.enable = { false | true }
```

false

All collection updates are treated as multiple JSON document updates. This is the default value.

With the `update.one.enable=false` setting, the MongoDB **db.collection.update** multi-parameter is ignored and all documents that meet the query criteria are updated.

true

Allow updates on collections to a single document or multiple documents.

With the `update.one.enable=true` setting, the MongoDB **db.collection.update** multi-parameter is accepted. The **db.collection.update** multi-parameter controls whether you can update a single document or multiple documents.

Performance

These parameters provide performance options for databases and collections.

delete.preparedStatement.cache.enable

This optional parameter indicates whether to cache prepared statements that delete documents for reuse.

```
delete.preparedStatement.cache.enable = { true | false }
```

true

Use a prepared statement cache for statements that delete documents. This is the default value.

false

Do not use a prepared statement cache for statements that delete documents. A new statement is prepared for each query.

insert.batch.enable

If multiple documents are sent as a part of a single INSERT statement, this optional parameter indicates whether to batch document inserts operations into collections.

```
insert.batch.enable = { true | false }
```

true

Batch document inserts into collections by using JDBC batch calls to perform the insert operations. This is the default value.

false

Do not batch document insert operations into collections.

insert.batch.queue.enable

This optional parameter indicates whether to queue INSERT statements into larger batches. You can improve insert performance by queuing INSERT statements, however, there is decreased durability.

This parameter batches all INSERT statements, even a single INSERT statement. These batched INSERT statements are flushed at the interval that is specified by the **insert.batch.queue.flush.interval** parameter, unless another operation arrives on the same collection. If another operation arrives on the same collection, the batch inserts are immediately flushed to the database server before proceeding with the next operation.

```
insert.batch.queue.enable = { false | true }
```

false

Do not queue INSERT statements. This is the default.

true

Queue INSERT statements into larger batches. Use the **insert.batch.queue.flush.interval** parameter to specify the amount of time between insert queue flushes.

insert.batch.queue.flush.interval



Prerequisite: `insert.batch.queue.enable=true`

This optional parameter specifies the number of milliseconds between flushes of the insert queue to the database server. The default value is `insert.batch.queue.flush.interval=100`.

```
insert.batch.queue.flush.interval = { 100 | flush_interval_time }
```

index.cache.enable

This optional parameter indicates whether to enable index caching on collections. To write the most efficient queries, the wire listener must be aware of the existing BSON indexes on your collections.

```
index.cache.enable = { true | false }
```

true

Cache indexes on collections. This is the default value.

false

Do not cache indexes on collections. The wire listener queries the database for indexes each time a collection query is translated to SQL.

index.cache.update.interval

This optional parameter specifies the amount of time, in seconds, between updates to the index cache on a collection table. The default value is `index.cache.update.interval=120`.

```
index.cache.update.interval = { 120 | cache_update_interval }
```

insert.preparedStatement.cache.enable

This optional parameter indicates whether to cache the prepared statements that are used to insert documents.

```
insert.preparedStatement.cache.enable = { true | false }
```

true

Cache the prepared statements that are used to insert documents. This is the default value.

false

Do not cache the prepared statements that are used to insert documents.

preparedStatement.cache.enable

This optional parameter indicates whether to cache prepared statements for reuse.

```
preparedStatement.cache.enable = { true | false }
```

true

Use a prepared statement cache. This is the default value.

false

Do not use a prepared statement cache. A new statement is prepared for each query.

preparedStatement.cache.size

This optional parameter specifies the size of the least-recently used (LRU) map that is used to cache prepared statements. The default value is `preparedStatement.cache.size=20`.

```
preparedStatement.cache.size = { 20 | LRU_size }
```

Security

The parameters provide security enablement options.

authentication.enable

This optional parameter indicates whether to enable user authentication.

You can choose to authenticate users through the wire listener or in the database server.

```
authentication.enable = { false | true }
```

false

Do not authenticate users. This is the default value.

true

Authenticate users. Use the **authentication.localhost.bypass.enable** parameter to control the type of authentication.

authentication.localhost.bypass.enable



Prerequisite: `authentication.enable=true`

If you connect from the localhost to the Informix® **admin** database, and the **admin** database contains no users, this optional parameter indicates whether to grant full administrative access.

The Informix® **admin** database is similar to the MongoDB admin database. The Informix® **authentication.localhost.bypass.enable** parameter is similar to the MongoDB **enableLocalhostAuthBypass** parameter.

```
authentication.localhost.bypass.enable= { true | false }
```

true

Grant full administrative access to the user. This is the default value.

false

Do not grant full administrative access to the user.

command.denylist

This optional parameter lists commands that are removed from the command registry and can not be called. By default, the deny list is empty.

```
command.denylist = [ command ]
```

db.authentication

This optional parameter specifies the user authentication method. See [User authentication with the wire listener on page 47](#).

```
db.authentication = { mongodb-cr | informix-password | informix-mongodb-cr }
```

mongodb-cr

Authenticate through the wire listener with a MongoDB authentication method. The MongoDB authentication method depends on the setting of the `mongo.api.version` parameter. This is the default value when `listener.type` is set to "mongo".

informix-password

Authenticate through the database server with the username and password provided by the client connection. Informix password authentication is only supported on the REST and MQTT listeners. It is not supported on Mongo listeners. This is the default value when `listener.type` is set to "rest" or "mqtt".

informix-mongodb-cr

Authenticate through the database server with a pluggable authentication module.

listener.admin.ipAddress

This optional parameter specifies the IP address for the administrative host. Must be a loopback IP address. The default value is 127.0.0.1.



Important: If you specify an address that is not a loopback IP address, an attacker might perform a remote privilege escalation and obtain administrative privileges without knowing a user password.

```
listener.admin.ipAddress = ip_address
```

listener.authentication.timeout

This optional parameter specifies the number of milliseconds that the wire listener waits for a client connection to authenticate. The default value is 0, which indicates that the wire listener waits indefinitely for client connections to authenticate.

```
listener.authentication.timeout = milliseconds
```

listener.http.accessControlAllowCredentials

This optional parameter indicates whether to display the response to the request when the omit credentials flag is not set. When this parameter is part of the response to a preflight request, it indicates that the actual request can include user credentials.

```
listener.http.accessControlAllowCredentials= { true | false }
```

true

Display the response to the request. This is the default value.

false

Do not display the response to the request.

listener.http.accessControlAllowHeaders

This optional parameter, which is part of the response to a preflight request, specifies the header field names that are used during the actual request. You must specify the value by using a JSON array of strings. Each string in the array is the case-insensitive header field name. The default value is

```
listener.http.accessControlAllowHeaders=[ "accept" , "cursorId" , "content-type" ].
```

```
listener.http.accessControlAllowHeaders= [ { "accept", "cursorId", "content-type" | "header_field_name" } ]
```

For example, to allow the headers `foo` and `bar` in a request:

```
listener.http.accessControlAllowHeaders=["foo","bar"]
```

listener.http.accessControlAllowMethods

This optional parameter, which is part of the response to a preflight request, specifies the REST methods that are used during the actual request. You must specify the value by using a JSON array of strings. Each string in the array is the name of an HTTP method that is allowed. The default value is

```
listener.http.accessControlAllowMethods=["GET", "PUT", "POST", "DELETE", "OPTIONS"].
```

```
listener.http.accessControlAllowMethods= [ { "GET", "PUT", "POST", "DELETE", "OPTIONS" | "rest_method" } ]
```

listener.http.accessControlAllowOrigin

This optional parameter specifies which uniform resource identifiers (URI) are authorized to receive responses from the REST listener when processing cross-origin resource sharing (CORS) requests. You must specify the value by using a JSON array of strings, with a separate string in the array for each value for the HTTP Origin header in a request. The values that are specified in this parameter are validated to ensure that they are identical to the Origin header.

HTTP requests include an Origin header that specifies the URI that served the resource that processes the request. When a resource from a different origin is accessed, the resource is validated to determine whether sharing is allowed.

The default value, `listener.http.accessControlAllowOrigin={"$regex":".*"}`, means that any origin is allowed to perform a CORS request.

```
listener.http.accessControlAllowOrigin= [ { "$regex":".*" | authorized_URI } ]
```

Here are some usage examples:

- Grant access to the localhost:

```
listener.http.accessControlAllowOrigin="http://localhost"
```

- Grant access to all hosts in the subnet 10.168.8.0/24. The first 3 segments are validated as 10, 168, and 8, and the fourth segment is validated as a value 1 - 255:

```
listener.http.accessControlAllowOrigin={"$regex":"^http://10\\\\\\\\.168\\\\\\\\.8\\\\\\\\.([01]?\\\\\\\\d\\\\\\\\\\\\d?|2[0-4]\\\\\\\\\\\\d|25[0-5])$" }
```

- Grant access to all hosts in the subnet 10.168.8.0/24. The first 3 segments are validated as 10, 168, and 8, and the fourth segment must contain one or more digits:

```
listener.http.accessControlAllowOrigin={"$regex":  
"^http://10\\\\\\\\.168\\\\\\\\.8\\\\\\\\.\\\\\\\\d+$" }
```

listener.http.accessControlExposeHeaders

This optional parameter specifies which headers of a CORS request to expose to the API. You must specify the value by using a JSON array of strings. Each string in the array is the case-insensitive name of a header to be exposed. The default value is `listener.http.accessControlExposeHeaders=["cursorId"]`.

```
listener.http.accessControlExposeHeaders= { ["cursorId"] | CORS_headers }
```

For example, to expose the headers `foo` and `bar` to a client:

```
listener.http.accessControlExposeHeaders=["foo","bar"]
```

listener.http.accessControlMaxAge

This optional parameter specifies the amount of time, in seconds, that the result of a preflight request is cached in a preflight result cache. A value of 0 indicates that the Access-Control-Max-Age header is not included in the response to a preflight request. A value greater than 0 indicates that the Access-Control-Max-Age header is included in the response to a preflight request.

The default value is `listener.http.accessControlMaxAge=0`.

```
listener.http.accessControlMaxAge= { 0 | preflight_result_cache_time }
```

listener.http.headers

This optional parameter specifies the information to include in the HTTP headers of responses, as a JSON document. The default value is no additional information in the HTTP headers.

```
listener.http.headers = JSON_document
```

For example, you set this parameter to the following value:

```
listener.http.headers={ "Access-Control-Allow-Origin" : "http://192.168.0.1",  
"Access-Control-Allow-Credentials" : "true" }
```

Then the HTTP headers for all responses look like this:

```
Access-Control-Allow-Origin : http://192.168.0.1  
Access-Control-Allow-Credentials : true
```

listener.http.headers.size.maximum

This optional parameter specifies the maximum size of headers in incoming HTTP requests. The default is 8192 bytes.

```
listener.http.headers.size.maximum = { 8192 | size }
```

listener.rest.cookie.domain

This optional parameter specifies the name of the cookie that is created by the REST wire listener. If not specified, the domain is the default value as determined by the Apache Tomcat web server.

```
listener.rest.cookie.domain= { [ rest_cookie_name ] }
```

listener.rest.cookie.httpOnly

This optional parameter indicates whether to set the HTTP-only flag.

```
listener.rest.cookie.httpOnly= { true | false }
```

true

Set the HTTP-only flag. This flag helps to prevent cross-site scripting attacks. This is the default value.

false

Do not set the HTTP-only flag.

listener.rest.cookie.length

This optional parameter specifies the length, in bytes, of the cookie value that is created by the REST wire listener, before Base64 encoding. The default value is `listener.rest.cookie.length=64`.

```
listener.rest.cookie.length= { 64 | rest_cookie_length }
```

listener.rest.cookie.name

This optional parameter specifies the name of the cookie that is created by the REST wire listener to identify a session. The default value is `listener.rest.cookie.name=informixRestListener.sessionId`.

```
listener.rest.cookie.name= { informixRestListener.sessionId | rest_cookie_name }
```

listener.rest.cookie.path

This optional parameter specifies the path of the cookie that is created by the REST wire listener. The default value is `listener.rest.cookie.path=.`

```
listener.rest.cookie.path= { / | rest_cookie_path }
```

listener.rest.cookie.secure

This optional parameter indicates whether the cookies that are created by the REST wire listener have the secure flag on. The secure flag prevents the cookies from being used over an unsecure connection.

```
listener.rest.cookie.secure= { false | true }
```

false

Turn off the secure flag. This is the default value.

true

Turn on the secure flag.

listener.ssl.algorithm

This optional parameter specifies the Service Provider Interface (SPI) for the KeyManagerFactory that is used to access the network encryption keystore. On an Oracle Java Virtual Machine (JVM), this value is typically SunX509. On an IBM® JVM, this value is typically IbmX509. The default value is no SPI.



Important: Do not set this property if you are not familiar with Java Cryptography Extension (JCE).

```
listener.ssl.algorithm = SPI
```

listener.ssl.ciphers

This optional parameter specifies a list of Secure Sockets Layer (SSL) or Transport Layer Security (TLS) ciphers to use with network encryption. The default value is no ciphers, which means that the default list of enabled ciphers for the JVM are used.



Important: Do not set this property if you are not familiar with Java Cryptography Extension (JCE) and the implications of using multiple ciphers. Consult a security expert for advice.

```
listener.ssl.ciphers = cipher
```

You can include spaces between ciphers.

For example, you can set the following ciphers:

```
listener.ssl.ciphers=TLS_RSA_WITH_AES_128_CBC_SHA, TLS_RSA_WITH_AES_256_CBC_SHA,
TLS_EMPTY_RENEGOTIATION_INFO_SCSV
```

listener.ssl.enable

This optional parameter enables SSL or TLS network encryption on the socket for client connections. See [Configuring SSL connections between the wire listener and client applications on page 53](#).

```
listener.ssl.enable = { false | true }
```

false

Disable network encryption. This is the default.

true

Allow network encryption.

listener.ssl.key.alias

This optional parameter specifies the alias, or identifier, of the entry into the keystore. The default value is no alias, which indicates that the keystore contains one entry. If the keystore contain more than one entry and a key password is needed to unlock the keystore, set this parameter to the alias of the entry that unlocks the keystore.

```
listener.ssl.key.alias = alias
```

This parameter is effective when the listener.ssl.enable parameter is set to true.

listener.ssl.key.password

This optional parameter specifies the password to unlock the entry into the keystore, which is identified by the listener.ssl.key.alias parameter. The default value is no password, which means to use the keystore password. If the entry into the keystore requires a password that is different from the keystore password, set this parameter to the entry password.

```
listener.ssl.key.password = password
```

This parameter is effective when the listener.ssl.enable parameter is set to true.

listener.ssl.keyStore.file

This optional parameter specifies the fully-qualified path and file name of the Java keystore file to use for network encryption. The default value is no file.

```
listener.ssl.keyStore.file = file_path
```

This parameter is effective when the listener.ssl.enable parameter is set to true.

listener.ssl.keyStore.password

This optional parameter specifies the password to unlock the Java keystore file for network encryption. The default value is no password.

```
listener.ssl.keyStore.password = password
```

This parameter is effective when the listener.ssl.enable parameter is set to true.

listener.ssl.keyStore.type

This optional property specifies the provider identifier for the network encryption keystore SPI. The default value is JKS.



Important: Do not set this property if you are not familiar with Java Cryptography Extension (JCE).

```
listener.ssl.keyStore.type = SPI
```

This parameter is effective when the listener.ssl.enable parameter is set to true.

listener.ssl.protocol

This optional parameter specifies the SSL or TLS protocols. The default value is TLS.

```
listener.ssl.protocol = protocol
```


This parameter is effective when the `listener.ssl.enable` parameter is set to `true`.

security.sql.passthrough

This optional parameter indicates whether to enable support for issuing SQL statements by using JSON documents.

```
security.sql.passthrough = { false | true }
```

false

Disable the ability to issue SQL statements by using the MongoDB API. This is the default.

true

Allow SQL statements to be issued by using the MongoDB API.

Wire listener resource management

These parameters provide wire listener resource management options.

cursor.idle.timeout

This optional parameter specifies the number of milliseconds that a cursor can be idle before it is closed. The default value is 30000. A positive integer value for *time* specifies the number of milliseconds before an idle timeout.

```
cursor.idle.timeout = { 30000 | time }
```

listener.connectionPool.closeDelay.time

This optional parameter specifies the amount of time to keep a connection pool open after the last client disconnects. When the existing connection pool is open, the next connection can connect faster by reusing the existing pool instead of creating a new connection pool. The default value is 0, which indicates that the connection pool is closed immediately after the last client disconnects. A positive integer value for *time* specifies the number of time units to keep the connection pool open. The unit of time is set by the **listener.connectionPool.closeDelay.timeUnit** parameter.

```
listener.connectionPool.closeDelay.time = { 0 | time }
```

listener.connectionPool.closeDelay.timeUnit

This optional parameter specifies the time unit for the **listener.connectionPool.closeDelay.time** parameter. The *unit* can be NANoseconds, MICROseconds, MILLIseconds, SECONDS, MINUTES, HOURS, or DAYS. The default value is SECONDS.

```
listener.connectionPool.closeDelay.timeUnit = { SECONDS | unit }
```

listener.idle.timeout

This optional parameter specifies the amount of time, in milliseconds, that a client connection to the wire listener can idle before it is forcibly closed. You can use this parameter to close connections and free

associated resources when clients are idle. The default value is 300000 milliseconds. The value of 0 indicates that client connections are never timed out.



Important: When set to a nonzero value, the wire listener socket that is used to communicate with a MongoDB client is forcibly closed after the specified time. To the client, the forcible closure appears as an unexpected disconnection from the server the next time there is an attempt to write to the socket.

```
listener.idle.timeout = { 300000 | idle_time }
```

listener.idle.timeout.minimum

This optional parameter specifies the lower threshold, in milliseconds, of the listener idle timeout, which is set by the low memory monitor. The default value is 10000 milliseconds. This property has no effect when the heap size is sufficiently large to not need a reduction in idle timeout.

```
listener.idle.timeout.minimum = { 10000 | idle_time }
```

This parameter is effective when the `listener.memoryMonitor.enable` parameter is set to `true`.

listener.input.buffer.size

This optional parameter specifies the size, in MB, of the input buffer for each wire listener socket. The default value is 8192 bytes.

```
listener.input.buffer.size = { 8192 | input_buffer_size }
```

listener.memoryMonitor.enable

This optional parameter enables the wire listener memory monitor. When memory usage for the wire listener is high, the memory monitor attempts to reduce resources, such as removing cached JDBC prepared statements, removing idle JDBC connections from the connection pools, and reducing the maximum size of responses.

```
listener.memoryMonitor.enable = { true | false }
```

true

Enable the memory monitor. This is the default.

false

Disable the memory monitor.

listener.memoryMonitor.allPoint

This optional parameter specifies the maximum percentage of heap usage before the memory monitor reduces resources. The default value is 80.

```
listener.memoryMonitor.allPoint = percentage
```

This parameter is effective when the `listener.memoryMonitor.enable` parameter is set to `true`.

listener.memoryMonitor.diagnosticPoint

This optional parameter specifies the percentage of heap usage before diagnostic information about memory usage is logged. The default value is 99.

```
listener.memoryMonitor.diagnosticPoint = percentage
```

This parameter is effective when the listener.memoryMonitor.enable parameter is set to true.

listener.memoryMonitor.zeroPoint

This optional parameter specifies the percentage of heap usage before the memory manager reduces resource usage to the lowest possible levels. The default value is 95.

```
listener.memoryMonitor.zeroPoint = percentage
```

This parameter is effective when the listener.memoryMonitor.enable parameter is set to true.

listener.output.buffer.size

This optional parameter specifies the size, in MB, of the output buffer for each listener socket. The default value is 8192 bytes.

```
listener.output.buffer.size = { 8192 | output_buffer_size }
```

listener.pool.admin.enable

This optional parameter enables a separate thread pool for connections from the administrative IP address, which is set by the listener.admin.ipAddress parameter. The default value is false. A separate thread pool ensures that administrative connections succeed even if the listener thread pool lacks available resources.

```
listener.pool.admin.enable = { false | true }
```

false

Prevents a separate thread pool. This is the default.

true

Creates a separate thread pool for administrative connections.

listener.pool.keepAliveTime

This optional parameter specifies the amount of time, in seconds, that threads above the core pool size are allowed to idle before they are removed from the wire listener JDBC connection pool. The default value is 60 seconds.

```
listener.pool.keepAliveTime = { 60 | thread_idle }
```

listener.pool.queue.size

This optional parameter specifies the number of requests to queue above the core wire listener pool size before expanding the pool size up to the maximum. A positive integer specifies the queue size to use before expanding the pool size up to the maximum.

```
listener.pool.queue.size = { 0 | -1 }
```

0

Do not allocate a queue size for tasks. All new sessions are either run on an available or new thread up to the maximum pool size, or are rejected if the maximum pool size is reached. This is the default value.

-1

Allocate an unlimited queue size for tasks.

listener.pool.size.core

This optional parameter specifies the maximum sustained size of the thread pool that listens for incoming connections from clients. The default value is 128.

```
listener.pool.size.core = { 128 | max_thread_size }
```

listener.pool.size.maximum

This optional parameter specifies the maximum peak size of the thread pool that listens for incoming connections from clients. The default value is 1024.

```
listener.pool.size.maximum = { 1024 | max_peak_thread_size }
```

listener.socket.accept.timeout

This optional parameter specifies the number of milliseconds that a server socket waits for an accept() function. The default value is 1024. The value of 0 indicates to wait indefinitely. The value of this parameter can affect how quickly the wire listener shuts down.

```
listener.socket.accept.timeout = milliseconds
```

listener.socket.read.timeout

This optional parameter specifies the number of milliseconds to block when calling a read() function on the socket input stream. The default value is 1024. A value of 0 might prevent the wire listener from shutting down because the threads that poll the socket might never unblock.

```
listener.socket.read.timeout = milliseconds
```

pool.connections.maximum

This optional parameter specifies the maximum number of active connections to a database. The default value is 50.

```
pool.connections.maximum = { 50 | max_active_connect }
```

pool.idle.timeout

This optional parameter specifies the minimum amount of time that an idle connection is in the idle pool before it is closed. The default value is 60 and the default time unit is seconds.



Important: Set the unit of time in the **pool.idle.timeunit** parameter. The default value is seconds.

```
pool.idle.timeout = { 60 | min_idle_pool }
```

pool.idle.timeunit

Prerequisite: `pool.idle.timeout=time`

This optional parameter specifies the unit of time that is used to scale the **pool.idle.timeout** parameter.

```
pool.idle.timeunit = { SECONDS | NANoseconds | MICROseconds | MILLIseconds | MINUTES | HOURS | DAYS }
```

SECONDS

Use seconds as the unit of time. This is the default value.

NANoseconds

Use nanoseconds as the unit of time.

MICROseconds

Use microseconds as the unit of time.

MILLIseconds

Use milliseconds as the unit of time.

MINUTES

Use minutes as the unit of time.

HOURS

Use hours as the unit of time.

DAYS

Use days as the unit of time.

pool.lenient.return.enable

This optional parameter suppresses the following checks on a connection that is being returned that might throw exceptions:

- An attempt to return a pooled connection that is already returned.
- An attempt to return a pooled connection that is owned by another pool.
- An attempt to return a pooled connection that is an incorrect type.

```
pool.lenient.return.enable = { false | true }
```

false

Connection checks are enabled. This is the default.

true

Connection checks are disabled.

pool.lenient.dispose.enable

This optional parameter suppresses the checks on a connection that is being disposed of that might throw exceptions.

```
pool.lenient.dispose.enable = { false | true }
```

false

Connection checks are enabled. This is the default.

true

Connection checks are disabled.

pool.semaphore.timeout

This optional parameter specifies the amount of time to wait to acquire a permit for a database connection. The default value is 5 and the default time unit is seconds.



Important: Set the unit of time in the **pool.semaphore.timeunit** parameter.

```
pool.semaphore.timeout = { 5 | wait_time }
```

pool.semaphore.timeunit



Prerequisite: `pool.semaphore.timeout=wait_time`

This optional parameter specifies the unit of time that is used to scale the **pool.semaphore.timeout** parameter.

```
pool.semaphore.timeunit = { SECONDS | NANOSECONDS | MICROSECONDS | MILLISECONDS | MINUTES | HOURS | DAYS }
```

SECONDS

Use seconds as the unit of time. This is the default value.

NANOSECONDS

Use nanoseconds as the unit of time.

MICROSECONDS

Use microseconds as the unit of time.

MILLISECONDS

Use milliseconds as the unit of time.

MINUTES

Use minutes as the unit of time.

HOURS

Use hours as the unit of time.

DAYS

Use days as the unit of time.

pool.service.interval

This optional parameter specifies the amount of time to wait between scans of the idle connection pool. The idle connection pool is scanned for connections that can be closed because they have exceeded their maximum idle time. The default value is 30.



Important: Set the unit of time in the **pool.service.timeunit** parameter.

```
pool.service.interval = { 30 | wait_time }
```

pool.service.threads

This optional parameter specifies the number of threads to use for the maintenance of connection pools that share a common service thread pool. The default value is 1.

```
pool.service.threads = number
```

pool.service.timeunit

Prerequisite: `pool.service.interval=wait_time`

This optional parameter specifies the unit of time that is used to scale the **pool.service.interval** parameter.

```
pool.service.timeunit = { SECONDS | NANOSECONDS | MICROSECONDS | MILLISECONDS | MINUTES | HOURS | DAYS }
```

SECONDS

Use seconds as the unit of time. This is the default value.

NANOSECONDS

Use nanoseconds as the unit of time.

MICROSECONDS

Use microseconds as the unit of time.

MILLISECONDS

Use milliseconds as the unit of time.

MINUTES

Use minutes as the unit of time.

HOURS

Use hours as the unit of time.

DAYS

Use days as the unit of time.

pool.size.initial

This optional parameter specifies the initial size of the idle connection pool. The default value is 0.

```
pool.size.initial = { 0 | idle_pool_initial_size }
```

pool.size.minimum

This optional parameter specifies the minimum size of the idle connection pool. The default value is 0.

```
pool.size.minimum = { 0 | idle_pool_min_size }
```

pool.size.maximum

This optional parameter specifies the maximum size of the idle connection pool. The default value is 50.

```
pool.size.maximum = { 50 | idle_pool_max_size }
```

pool.type

This optional parameter specifies the type of pool to use for JDBC connections. The available pool types are:

```
pool.type = { basic | none | advanced | perThread }
```

basic

Thread pool maintenance of idle threads is run each time that a connection is returned. This is the default value.

none

No thread pooling occurs. Use this type for debugging purposes.

advanced

Thread pool maintenance is run by a separate thread.

perThread

Each thread is allocated a connection for its exclusive use.

pool.typeMap.strategy

This optional parameter specifies the strategy to use for distribution and synchronization of the JDBC type map for each connection in the pool.

```
pool.typeMap.strategy = { copy | clone | share }
```

copy

Copy the connection pool type map for each connection. This is the default value.

clone

Clone the connection pool type map for each connection.

share

Share a single type map between all connections. You must use this strategy with a thread-safe type map.

response.documents.size.minimum

This optional parameter specifies the number of bytes for the lower threshold for the maximum response size, which is set by the response.documents.size.maximum parameter. The memory manager can reduce the response size to this size when resources are low. The default value is 65536 bytes.

```
response.documents.size.minimum = bytes
```

This parameter is effective when the listener.memoryMonitor.enable parameter is set to true.

timeseries.loader.connections

This optional parameter specifies the number of connections between each time series table and the MQTT wire listener for loading time series data. The default value is 10 connections per table.

```
timeseries.loader.connections = { 10 | number }
```

Modifying the wire listener configuration file

You can modify the wire listener connection properties that are set in the configuration file.

About this task

The wire listener configuration file, named `%INFORMIXDIR%\etc\jsonListener.properties` by default, controls the wire listener and the connection between the client and database server.

To modify the wire listener configuration file:

- 1. Stop the wire listener.
- 2. Update the wire listener configuration file.
- 3. Start the wire listener.

Wire listener command line options


You can use command line options to control the wire listener.


Syntax

```
>>-java-- -jar pathToListener---- -config properties_file----->

>--+- -start-----+-----+-----+-----+-----+-----+-----+----->
|          '- -logfile--log_file-' |          .-error-. | |          .-27017------. | |
|          |          '- -loglevel--+-warn--+-' '- -port--+-port_number--+' |
|          |          +-info--+
|          |          +-debug--+
|          |          '-trace-'
|          '- -stop-----+-----+-----+-----+-----+-----+-----+-----'
|          |          .-10------. |
|          |          '- -wait--+-wait_time--+'

>-----+-----+-----+-----+-----+-----+-----><
'- -version-' '- -buildInformation-'
```

Argument	Purpose
-config <i>properties_file</i>	Specifies the name of the wire listener configuration file to run. This argument is required to start or stop the wire listener.
-start	Starts the wire listener. You must also specify the configuration file.
-stop	Stops the wire listener. You must also specify the configuration file. The stop command is similar to the MongoDB shutdown command.
-logfile <i>log_file</i>	Specifies the name of the log file that is used. If this option is not specified, the log messages are sent to <code>std.out</code> .  Important: If you have customized the Logback configuration or specified another logging framework, the settings for -loglevel and -logfile are ignored.
-loglevel	Specifies the logging level. error Errors are sent to the log file. This is the default value.

Argument	Purpose
	<p>warn</p> <p>Errors and warnings are sent to the log file.</p> <p>info</p> <p>Informational messages, warnings, and errors are sent to the log file.</p> <p>debug</p> <p>Debug, informational messages, warnings, and errors are sent to the log file.</p> <p>trace</p> <p>Trace, debug, informational messages, warnings, and errors are sent to the log file.</p> <p> Important: If you have customized the Logback configuration or specified another logging framework, the settings for -loglevel and -logfile are ignored.</p>
-port <i>port_number</i>	Specifies the port number. If a port is specified on the command line, it overrides the port properties set in the wire listener configuration file. The default port is 27017.
-wait <i>wait_time</i>	Specifies the amount of time, in seconds, to wait for any active sessions to complete before the wire listener is stopped. The default is 10 seconds. To force an immediate shutdown, set the <i>wait_time</i> to 0 seconds.
-version	Prints the wire listener version.
-buildInformation	Prints the wire listener build information.

Example

Examples

In this example, the wire listener is started and the log is specified as `$INFORMIXDIR/jsonListener.log`:

```
java -jar $INFORMIXDIR/bin/jsonListener.jar
-config $INFORMIXDIR/etc/jsonListener.properties
-logfile $INFORMIXDIR/jsonListener.log -start
```

In this example, the wire listener is started with the log level set to debug:

```
java -jar $INFORMIXDIR/bin/jsonListener.jar
-config $INFORMIXDIR/etc/jsonListener.properties
-loglevel debug -start
```

In this example, port 6388 is specified:

```
java -jar $INFORMIXDIR/bin/jsonListener.jar
-config $INFORMIXDIR/etc/jsonListener.properties
-port 6388 -start
```

In this example, the wire listener is paused 10 seconds before the wire listener is stopped:

```
java -jar $INFORMIXDIR/bin/jsonListener.jar
-config $INFORMIXDIR/etc/jsonListener.properties
-wait 10 -stop
```

In this example, the wire listener version is printed:

```
java -jar $INFORMIXDIR/bin/jsonListener.jar
-version
```

In this example, the wire listener build information is printed:

```
java -jar $INFORMIXDIR/bin/jsonListener.jar
-buildInformation
```

Starting the wire listener

You can start the wire listener for the MongoDB API, the REST API, or the MQTT protocol, by using the start command.

Before you begin

- Stop all wire listeners that are currently running. If you create a server instance during the installation process, the MongoDB API wire listener is started automatically and connected to the MongoDB API.
- If you plan to customize the Logback logger or another custom Simple Logging Facade for Java (SLF4J) logger, you must configure the logger before starting the wire listener.
- [Configuring the wire listener for the first time on page 8](#)
- [Requirements for JSON compatibility on page 5](#)

To start the wire listener, run the wire listener command with the -start option.

Example

For example:

```
java -jar $INFORMIXDIR/bin/jsonListener.jar
-config $INFORMIXDIR/etc/jsonListener.properties -start
```

The `listener.type` property in the configuration file that you specify defines whether to start the wire listener for the MongoDB API, the REST API, or the MQTT protocol.

Results

The wire listener starts.

Example

Examples

In the following example, the wire listener is started with the configuration file specified as

`jsonListener_mongo.properties`, the log file specified as `jsonListener_mongo.log`, and the log level specified as `info`:

```
java -jar $INFORMIXDIR/bin/jsonListener.jar
-config $INFORMIXDIR/etc/jsonListener_mongo.properties
```

```
-logfile $INFORMIXDIR/jsonListener_mongo.log
-loglevel info -start
```

Here is the output from starting the wire listener:

```
starting mongo listener on port 27017
```

In the following example, the wire listener is started with the configuration file specified as

`jsonListener_rest.properties`:

```
java -jar $INFORMIXDIR/bin/jsonListener.jar
-config $INFORMIXDIR/etc/jsonListener_rest.properties -start
```

Here is the output from starting the REST API wire listener:

```
starting rest listener on port 27017
```

Running multiple wire listeners

You can run multiple wire listeners.

About this task

By running multiple wire listeners, you can use a combination of the APIs that are supported by the wire listener: MongoDB, REST, and MQTT. Create a configuration file for each listener type that you want to run. For example, you can create a configuration file for the MongoDB API and a configuration file for the REST API or the MQTT protocol. You can start all wire listeners with the same start command by providing multiple `-config` arguments.

1. Create the individual configuration files in the `$INFORMIXDIR/etc` directory. You can use the `$INFORMIXDIR/etc/jsonListener-example.properties` file as a template.
2. Customize each configuration file and assign a unique name.



Important: The **url** parameter must be specified, either in each individual configuration file or in the file that is referenced by the **include** parameter.

- a. Specify the **include** parameter to reference an additional configuration file. The path can be relative or absolute. If you have multiple configuration files, you can avoid duplicating parameter settings in the multiple configuration files by specifying a subset of shared parameters in a single configuration file, and the unique parameters in the individual configuration files.
3. Start the wire listeners.

Example

Example: Running multiple wire listeners that share parameter settings

In this example, the same **url**, **authentication.enable**, and **security.sql.passthrough** parameters are used to run two wire listeners:

1. Create a configuration file named `shared.properties` that includes the following parameters:

```
url=jdbc:informix-sqli://localhost:9090/sysmaster:
INFORMIXSERVER=ol_informix1210;
```

```
authentication.enable=true
security.sql.passthrough=true
```

2. Create a configuration file for use with the MongoDB API that is named `mongo.properties`, with the parameter

`include=shared.properties` **set**:

```
include=shared.properties
listener.type=mongo
listener.port=27017
```

3. Create a configuration file for use with the REST API that is named `rest.properties`, with the parameter

`include=shared.properties` **set**:

```
include=shared.properties
listener.type=rest
listener.port=8080
```

4. From the command line, run the **start** command. Include separate `-config` arguments for each wire listener API type.

```
java -jar $INFORMIXDIR/bin/jsonListener.jar
-config json.properties
-config rest.properties -start
```

Stopping the wire listener

You can stop the wire listener by using the **stop** command.

About this task

You must stop the wire listener before you modify any configuration settings.

From the command line, run the **stop** command with the configuration file specified.

For example:

```
java -jar $INFORMIXDIR/bin/jsonListener.jar -config
$INFORMIXDIR/etc/jsonListener.properties -stop
```



Important: You must specify the `-config` argument to stop the wire listener from the command line.

Results


The wire listener is stopped.

Wire listener logging

The wire listener can output trace, debug, informational messages, warnings, and error information to a log.

The default logging mechanism for the wire listener is Logback. Logback is pre-configured and installed along with the JSON components. For more information on how to customize Logback, see <http://logback.qos.ch/>.

If you start the MongoDB API wire listener from the command line, you can specify the amount of detail, name, and location of your log file by using the `-loglevel` and `-logfile` command-line arguments.

 **Important:** If you have customized the Logback configuration or specified another logging framework, the settings for **-loglevel** and **-logfile** are ignored.

User authentication with the wire listener

You can authenticate users through the wire listener with MongoDB authentication or with the database server, through a pluggable authentication module (PAM).

You can use the following types of authentication methods with the wire listener:

MONGODB-CR challenge-response


The wire listener authenticates users with the MongoDB challenge-response authentication method outside of the database server environment. You create users with the MongoDB API create user commands. Clients connect to the wire listener as MongoDB users and the wire listener authenticates the users. The wire listener connects to the database server as the user that is specified by the **url** parameter in the wire listener configuration file. The database server cannot access MongoDB user account information.

For MongoDB version 2.4, user information and privileges are stored in the **system_users** collection in each database. For MongoDB version 2.6 and later, user information and privileges are stored in the **system.users** collection in the **admin** database. If you are upgrading your MongoDB version and you have existing users, you must upgrade your user schema.

SCRAM-SHA-1 two-step authentication

SCRAM-SHA-1 is only available when the **mongo.api.version=3.0** parameter is set in the wire listener configuration file.

The wire listener authenticates users with the SCRAM-SHA-1 authentication method outside of the database server environment. You create users with the MongoDB API create user commands. User information and privileges are stored in the **system.users** collection in the **admin** database. Clients connect to the wire listener as MongoDB users and the wire listener authenticates the users. The wire listener connects to the database server as the user that is specified by the **url** parameter in the wire listener configuration file. The database server cannot access MongoDB user account information.

 **Important:** You cannot use SCRAM authentication with the REST API or the MQTT protocol.

Database server authentication with a user and password

The wire listener connects to the database server using the user and password that is provided by clients and the database server authenticates the user. The database server controls all user accounts and privileges. You can audit user activities and configure fine-grained access control.



Important: You can use database server user password authentication only with the REST API and MQTT protocol.

Database server authentication with a PAM (UNIX, Linux)

The PAM implements the MONGODB-CR challenge-response method. The wire listener connects to the database server using the user and password that is provided by clients and the database server authenticates the user through PAM. The database server controls all user accounts and privileges. You can audit user activities and configure fine-grained access control.

Which types of authentication that you can use depend on the type of client and the version of MongoDB.

MongoDB clients

Table 1. Authentication types for the MongoDB API by version

Authentication type	Mong oDB 2.4	Mong oDB 2.6	MongoDB 3.0	Details
MONGODB-CR	Yes	Yes	No	Follow the instructions for configuring MongoDB authentication.
SCRAM-SHA-1	No	No	Yes	The user schema must be at MongoDB version 2.6 or later.
Informix user password	No	No	No	Database server authentication with a user and password is not supported for MongoDB clients because of the way MongoDB drivers hash the password.
PAM	Yes	Yes	No	Follow the instructions for configuring database server authentication with PAM.

REST API clients



Important: You cannot set the **mongo.api.version** parameter to 3.0 in the wire listener configuration file because the REST API does not support SCRAM authentication.

Table 2. Authentication types for the REST API by supported MongoDB versions

Authentication type	Mong oDB 2.4	Mong oDB 2.6	MongoDB 3.0	Details
MONGODB-CR	Yes	Yes	No	Follow the instructions for configuring MongoDB authentication. HTTP clients authenticate using the HTTP basic authentication method.

Table 2. Authentication types for the REST API by supported MongoDB versions (continued)

Authentication type	Mong oDB 2.4	Mong oDB 2.6	MongoDB 3.0	Details
SCRAM-SHA-1	No	No	No	SCRAM is not supported.
Informix user password	Yes	Yes	Yes	Set db.authentication=informix-password in your listener properties file. HTTP clients authenticate using the HTTP basic authentication method.
PAM	Yes	Yes	No	Follow the instructions for configuring database server authentication with PAM. HTTP clients authenticate using the HTTP basic authentication method.

MQTT clients



Important: You cannot set the **mongo.api.version** parameter to 3.0 in the wire listener configuration file because the MQTT protocol does not support SCRAM authentication.

Table 3. Authentication types for the MQTT protocol by supported MongoDB versions

Authentication type	Mong oDB 2.4	Mong oDB 2.6	MongoDB 3.0	Details
MONGODB-CR	Yes	Yes	Yes	Follow the instructions for configuring MongoDB authentication. The MQTT CONNECT packet must include the database name as a prefix of the user name, in the following format: <code>"database_name.user_name"</code> .
SCRAM-SHA-1	No	No	No	SCRAM is not supported.
Informix user password	Yes	Yes	Yes	Set db.authentication=informix-password in your listener properties file. The MQTT CONNECT packet must include the database name as a prefix of the user name, in the following format: <code>"database_name.user_name"</code> .
PAM	Yes	Yes	No	Follow the instructions for configuring database server authentication with PAM. The MQTT CONNECT packet must include the database name as a prefix of the user name, in the following format: <code>"database_name.user_name"</code> .

Configuring MongoDB authentication

You can configure the wire listener to use MongoDB authentication.

Before you begin

If you are upgrading your MongoDB version and you have existing MongoDB users, you must upgrade your user schema.

To configure MongoDB authentication:

1. Set the following parameters in the wire listener configuration file:
 - Enable authentication: Set `authentication.enable=true`.
 - Specify MongoDB authentication: Set `db.authentication=mongodb-cr`.
 - Set the MongoDB version: Set `mongo.api.version` to the version that you want.
 - Optional. Specify the authentication timeout period: Set the `listener.authentication.timeout` parameter to the number of milliseconds for authentication timeout.
2. Restart the wire listener.
3. If necessary, upgrade your user schema by running the `authSchemaUpgrade` command in the **admin** database.

Example

For example:

```
use admin
db.runCommand({authSchemaUpgrade : 1})
```

The `authSchemaUpgrade` command upgrades the user schema to the MongoDB version that is specified by the `mongo.api.version` parameter.

Adding users

To add authorized users:

1. Start the wire listener with authentication turned off: Set `authentication.enable=false` in the wire listener configuration file.
2. Add users:
Choose from:
 - For MongoDB version 2.4, run the `addUser` command for each user in each database.
 - For MongoDB version 2.6 and 3.0, run the `createUser` command for each user.
3. Turn on authentication: Set `authentication.enable=true` in the wire listener configuration file.
4. Restart the wire listener.

Configuring database server user password authentication

You can configure the database server to authenticate wire listener users based on the client user and password.

About this task

This is only supported for the REST and MQTT listener types. Mongo listeners do not support this type of authentication against the database server because of the way the Mongo client drivers hash the password before it is passed to the wire listener.

To configure database server user password authentication:

1. Set the following parameters in the wire listener configuration file:
 - Enable authentication: Set `authentication.enable=true`.
 - Specify Informix password authentication: Set `db.authentication=informix-password`.
2. Restart the wire listener.
3. Configure REST and MQTT clients to connect to the wire listener with a username and password that has privileges on the database server.

Configuring database server authentication with PAM (UNIX™, Linux™)

You can configure the database server to authenticate wire listener users with a pluggable authentication module (PAM).

About this task

You create a user for the wire listener for PAM connections. The wire listener uses the PAM user to look up system catalog-related information before sending client connection requests to the database server for authentication. The database server authenticates the client users through PAM.

To configure PAM authentication for MongoDB, REST, or MQTT clients:

1. Set the **IFMXMONGOAUTH** environment variable.

Example

For example:

```
setenv IFMXMONGOAUTH 1
```

2. Create a PAM service file that is named `/etc/pam.d/pam_mongo` and has the following contents:

```
auth    required $INFORMIXDIR/lib/pam_mongo.so file=mongohash
account required $INFORMIXDIR/lib/pam_mongo.so
```

Replace `$INFORMIXDIR` with the value of the **\$INFORMIXDIR** environment variable.

3. On IBM® AIX® 64-bit computers, create a symbolic link that is named `64` that points to the `lib` directory by running the following commands:

```
cd $INFORMIXDIR/lib
ln -s . 64
```

4. Edit the `sqlhosts` file to add a connection that uses PAM. Include the `s=4` option. Specify the PAM service `pam_mongo` with the `pam_serv` option. Specify the password authentication mode with the `pamauth` option.

For example:

```
ol_informix1210 onsoctcp myhost 40000 s=4,pam_serv=pam_mongo,pamauth=password
```

5. Enable connections from mapped users by setting the **USERMAPPING** configuration parameter to **BASIC** or **ADMIN** in the `onconfig` file.
6. Set up mapping to an operating system user that has no privileges.

Example

For example, on a typical Linux™ system, the user **nobody** is appropriate. Add the following line to the `/etc/informix/allowed.surrogates` file:

```
users:nobody
```

7. Restart the database server.
8. Create a PAM user for the wire listener. The user must be internally authenticated and map to the user **nobody**.

Example

For example, create a user that is named **mongo** by running the following SQL in the **sysmaster** database:

```
CREATE USER 'mongo' WITH PASSWORD 'aPassword'  
  PROPERTIES USER 'nobody';  
GRANT CONNECT TO 'mongo';
```

9. Verify the creation of the user by running the following statement:

Example

```
SELECT * FROM sysuser:sysmongousers  
  WHERE username='mongo';
```

The result of the query shows the user and hashed password:

```
username    mongo  
hashed_password  bbb8f9630d5c6e094b9aedd945893faf
```

10. Set the following parameters in the wire listener configuration file:
 - Enable authentication: Set authentication.enable=true.
 - Specify PAM authentication: Set db.authentication=informix-mongodb-cr.
 - Set the MongoDB version: Set mongo.api.version=2.6 or mongo.api.version=2.4. The PAM authentication method is not compatible with MongoDB version 3.0.
 - Optional. Specify the authentication timeout period: Set the listener.authentication.timeout parameter to the number of milliseconds for authentication timeout.
 - Specify the mapped user and password for connections and specify to encode and hash the password: Set the url parameter. Include the NONCE property set to any 16 character string that contains only the digits 0-9 and the lower-case characters a-f (extended grep: [0-9a-f]{16}). For example:

```
url=jdbc:informix-sqli://10.168.8.135:40000/sysmaster:USER=mongo;  
  PASSWORD=aPassword;NONCE=0123456789abcdef
```

11. Restart the wire listener.
12. Create users that the database server authenticates with PAM by running the SQL statement CREATE USER.
If you have existing MongoDB users, you must re-create those users in the database server.

Encryption for wire listener communications

You can use Secure Sockets Layer (SSL) protocol to encrypt communication for the wire listener.

You can encrypt wire listener communications in one or both of the following ways:

- Configure SSL connections between the wire listener and the database server.
- Configure SSL connections between the wire listener and all client applications.

If you configure SSL communication for both the database server and client applications, you can use the same or different keystore files on the wire listener for each type of connection.

Configuring SSL connections between the wire listener and the database server

You can encrypt the connections between the wire listener and the database server with the Secure Sockets Layer (SSL) protocol.

Before you begin

You must have SSL configured for the database server. See [Configuring a server instance for secure sockets layer connections on page](#) .

About this task

The wire listener must use the same public key certificate file as the database server.

To configure SSL connections between the wire listener and the database server:

1. Use the keytool utility that comes with your Java runtime environment to import a client-side keystore database and add the public key certificate to the keystore:

```
C:\work>keytool -importcert -file server_keystore_file -keystore client_keystore_name
```

The *server_keystore_file* is the name of the server key certificate file.

2. Edit the wire listener properties file to update the url property to use the SSL port that you configured for the database server and add the `SSLCONNECTION=true` property to the end of the URL.
3. Start the listener with the `javax.net.ssl.trustStore` and `javax.net.ssl.trustStorePassword` system properties set:

```
java -Djavax.net.ssl.trustStore="client_keystore_path"
-Djavax.net.ssl.trustStorePassword="password" -jar jsonListener.jar
-config jsonListener.properties -logfile jsonListener.log -start
```

The *client_keystore_path* is the full path and file name of the client keystore file. The *password* is the keystore password.

Configuring SSL connections between the wire listener and client applications

You can encrypt the connections between the wire listener and the client applications with the Secure Sockets Layer (SSL) protocol.

About this task

All client applications must use the same public key certificate file as the wire listener.

To configure SSL connections between the wire listener and client applications:

1. Create a keystore and certificate for the wire listener.
Use the method that best fits your type of client application and programming language. For example, you can use , OpenSSL, or the Java keytool tool.
2. Edit the wire listener properties file to configure the wire listener SSL properties and restart the listener.
Set the following SSL properties:

- Set the **listener.ssl.enable** parameter to `true` to enable SSL.
 - Set the **listener.ssl.keyStore.file** parameter to the path of the keystore file.
 - Set the **listener.ssl.keyStore.password** parameter to the password to unlock the keystore file.
 - Set the **listener.ssl.key.alias** parameter to the alias or identifier of the keystore entry. If the keystore contains only one entry, this parameter does not need to be set.
 - Set the **listener.ssl.key.password** parameter to the password to unlock the entry from the keystore. If this parameter is not set, the listener uses the **listener.ssl.keyStore.password** parameter.
 - Set **listener.ssl.keyStore.type** parameter if the keystore is not of type JKS (Java keystore).
3. For REST listeners (`listener.type=rest`), if you are using Java 8, you will need to include the Jetty ALPN boot jar file in the `-Xbootclasspath JVM` argument when starting the listener. See <https://www.eclipse.org/jetty/documentation/9.4.x/alpn-chapter.html> for more information.



Note: If you are running the REST listener with Java 9 or higher, ALPN is built into the JRE, so this step is not required.

4. Configure client applications to connect to the listener over SSL.

High availability support in the wire listener

The wire listener provides high availability support.

To provide high availability to client applications, use the appropriate method:

- For REST clients, you can use a reverse proxy for multiple wire listeners.
- For MongoDB clients, use a high-availability cluster configuration for your Informix® database servers. For each database server in the cluster, run a wire listener that is directly connected to that database server. Each wire listener must be on the same computer as the database server that it is connected to and all wire listeners must run on the port 27017. For more information, see <http://docs.mongodb.org/meta-driver/latest/legacy/connect-driver-to-replica-set/>.

To provide high availability between the wire listener and the Informix® database server, use one of the following methods:

- Route the connection between the wire listener and the database server through the Connection Manager.
- Configure the **url** parameter in the wire listener configuration file to use one of the methods of connecting to a high-availability cluster. For more information, see [Dynamically reading the Informix® sqlhosts file on page](#) or [Properties for connecting directly to an HDR pair of servers on page](#).

JSON data sharding

You can shard data with HCL Informix®. Documents from a collection or rows from a table can be sharded across a cluster of database servers, reducing the number of documents or rows and the size of the index for the database of each server. When you shard data across database servers, you also distribute performance across hardware. As your database grows in size, you can scale up by adding more shard servers to your shard cluster.

Documents or rows that are inserted on a shard server are distributed to the appropriate shard servers in a shard cluster based on the sharding schema. Queries on a sharded table automatically retrieve data from all relevant shard servers in a shard cluster. When data is sharded based on a field or column that specifies certain segmentation characteristics, queries can skip shard servers that do not contain relevant data.

A shard cluster of Informix® database servers is a special form of Enterprise Replication. You can create a shard cluster with Enterprise Replication commands or with MongoDB commands.

Informix® shard cluster architecture is very flexible:

- Shard servers can run on different hardware and operating systems.
- Shard servers can run different version of Informix®. For example, you can upgrade Informix® on shard servers individually.
- Shard servers can have high-availability secondary servers from which users can query the sharded table.

To start sharding data:

1. Prepare shard servers for sharding.
2. Create a shard cluster.
3. Define a schema for sharding data against an existing table.

Preparing shard servers

You must prepare shard servers before you can shard data.

To set up shard servers:

1. On each shard server, set the SHARD_ID configuration parameter to a positive integer value that is unique in the shard cluster by running the following command:

```
onmode -wvf SHARD_ID=unique_positive_integer
```

If the SHARD_ID configuration parameter is already set to a positive integer, you can change the value by editing the `onconfig` file and then restarting the database server. You can also set the SHARD_MEM configuration parameter to customize the number of memory pools that are used during shard queries.

2. Specify trusted hosts information for all shard servers.

On each shard server, run the SQL administration API `task()` or `admin()` function with the `cdr add trustedhost` argument and include the appropriate host values for all the other shard servers. You must be a Database Server Administrator (DBSA) to run these functions.

3. On each shard server, edit the wire listener configuration file:

- a. Set the **sharding.enable** parameter to `true`.
- b. Set the **sharding.query.parallel.enable** parameter to `true`.
- c. Set the **update.client.strategy** parameter to `deleteInsert`.
- d. If you want to allow shard key field values to be updated, set the **update.mode** parameter to `client`. If you do not want to allow the updating of shard key field values, you can leave the setting of the **update.mode** parameter as the default value of `mixed`.

- e. Set the **USER** attribute in the **url** parameter to a user who has the REPLICATION privilege. If you created a database server instance during installation, the **ifxjson** user, who has the REPLICATION privilege, is automatically set as the value of the **USER** attribute. Otherwise, see [Configuring the wire listener for the first time on page 8](#) for instructions.
4. On each shard server, restart the wire listener.

What to do next

When applications connect to shard servers, enable sharded queries to run against data across all shard servers by setting the USE_SHARDING session environment variable:

```
SET ENVIRONMENT USE_SHARDING ON;
```

Shard-cluster definitions for distributing data

A cluster of shard servers uses a definition to distribute data across shard servers.

You must create a shard-cluster definition to distribute data across the shard servers. The definition contains the following information:

- The Informix® Enterprise Replication group name of each participating shard server.
- The name of the database and collection or table that is distributed across the shard servers of a shard cluster.
- The field or column that is used as a shard key for distributing data. Shard key values determine which shard server a document or row is stored on.
- The sharding method by which documents or rows are distributed to specific shard servers. The sharding method is either a hash-based or expression-based.

Defining a sharding schema with a hash algorithm

The shardCollection command in the MongoDB shell creates a definition for distributing data across the database servers of a shard cluster.

To create a shard-cluster definition that uses a hash algorithm for distributing data across database servers:

1. Run the mongo command.

Result

The command starts the MongoDB shell.

2. Run the shardCollection command.

There are two ways to run the command:

Choose from:

- Run the sh.shardCollection MongoDB command. For example:

```
> sh.shardCollection("database1.collection1",  
  {customer_name:"hashed"})
```

- Run the db.runCommand from the MongoDB shell, with shardCollection command syntax. For example:

```
> db.runCommand({"shardCollection":"database2.collection_2",  
  key:{customer_name:"hashed"}})
```


The shardCollection command syntax for using a hash algorithm is shown in the following diagram:

```
db.runCommand ( { "shardCollection": " database. { collection | table } " , key: { { field | column } : "hashed" } } )
```

Element	Description	Restrictions
<i>database</i>	The name of the database that contains the collection that is distributed across database servers.	The database must exist.
<i>collection</i>	The name of the collection that is distributed across database servers.	The collection must exist.
<i>column</i>	The shard key that is used to distribute data across the database servers of a shard cluster.	The column must exist. Composite shard keys are not supported.
<i>field</i>	The shard key that is used to distribute data across the database servers of a shard cluster.	The field must exist. Composite shard keys are not supported.
<i>table</i>	The name of the table that is distributed across database servers.	The table must exist.

- For optimal query performance, connect to the wire listener and run the MongoDB ensureIndex command on the shard key of each of a cluster's shard servers. The ensureIndex command ensures that an index for the collection or table is created on the shard server.

Results

The name of a shard-cluster definition that is created by a shardCollection command that is run through the wire listener is:

```
sh_database_ { collection | table }
```

Example

Example

The following command defines a shard cluster that uses a hash algorithm on the shard key value **year** to distribute data across multiple database servers.

```
> sh.shardCollection("mydatabase.mytable",{year:"hashed"})
```

The name of the created shard-cluster definition is **sh_mydatabase_mytable**.

Defining a sharding schema with an expression

The MongoDB shell `db.runCommand` command with `shardCollection` command syntax creates a definition for distributing data across the database servers of a shard cluster.

To create a shard-cluster definition that uses an expression for distributing data across database servers:

1. Run the mongo command.

Result

The command starts the MongoDB shell.

2. Run the `db.runCommand` from the MongoDB shell, with `shardCollection` command syntax.

The `shardCollection` command syntax for using an expression is shown in the following diagram:

```
db.runCommand ( { "shardCollection": " database. { collection | table } ", key: { { column | field } :1 }, expressions : {  
"ER_group_name": expression " ER_group_name " : "remainder" } }
```

Element	Description	Restrictions
<i>collection</i>	The name of the collection that is distributed across database servers.	The collection must exist.
<i>column</i>	The shard key that is used to distribute data across the database servers of a shard cluster.	The column must exist. Composite shard keys are not supported.
<i>database</i>	The name of the database that contains the collection that is distributed across database servers.	The database must exist.
<i>ER_group_name</i>	The group name of a database server that receives copied data. The default group name for a database server is the database server's name prepended with <code>g_</code> . For example, the default group name for a database server that is named myserver is g_myserver .	None.
<i>expression</i>	The expression that is used to select documents by shard key value.	None.
<i>field</i>	The shard key that is used to distribute data across the database servers of a shard cluster.	The field must exist. Composite shard keys are not supported.
<i>remainder</i>	Specifies a database server that receives documents with shard key values that are not	

Element	Description	Restrictions
	selected by expressions. The remainder expression is required.	
<i>table</i>	The name of the table that is distributed across database servers.	The table must exist.

- For optimal query performance, connect to the wire listener and run the MongoDB `ensureIndex` command on the shard key of each of a cluster's shard servers. The `ensureIndex` command ensures that an index is created for the collection or table on the shard server.

Results

The name of a shard-cluster definition that is created by a `shardCollection` command that is run through the wire listener is:

```
sh_database_ { collection | table }
```

Example

Examples

Define a shard cluster that uses an expression to distribute data across multiple database servers

The following command defines a shard cluster that uses an expression on the field value **state** for distributing **collection1** across multiple database servers.

```
> db.runCommand({"shardCollection":"database1.collection1",
  key:{state:1},expressions:{"g_shard_server_1":"in ('KS','MO')",
  "g_shard_server_2":"in ('CA','WA')","g_shard_server_3":"remainder"}})
```

The name of the created shard-cluster definition is **sh_database1_collection1**.

- Inserted documents with **KS** and **MO** values in the **state** field are sent to **g_shard_server_1**.
- Inserted documents with **CA** and **WA** values in the **state** field are sent to **g_shard_server_2**.
- All inserted documents that do not have **KS**, **MO**, **CA**, or **WA** values in the **state** field are sent to **g_shard_server_3**.

Define a shard cluster that uses an expression to distribute data across multiple database servers

The following command defines a shard cluster that uses an expression on the column value **animal** for distributing **table2** across multiple database servers.

```
> db.runCommand({"shardCollection":"database1.table2",
  key:{animal:1},expressions:{"g_shard_server_1":"in ('dog','coyote')",
  "g_shard_server_2":"in ('cat')","g_shard_server_3":"in ('rat')",
  "g_shard_server_4":"remainder"}})
```

The name of the created shard-cluster definition is **sh_database2_table2**.

- Inserted rows with **dog** or **coyote** values in the **animal** column are sent to **g_shard_server_1**.
- Inserted rows with **cat** values in the **animal** column are sent to **g_shard_server_2**.

- Inserted rows with **rat data** values in the **animal** column are sent to **g_shard_server_3**.
- All inserted rows that do not have **dog, coyote, cat,** or **rat** values in the **animal** column are sent to **g_shard_server_4**.

Define a shard cluster that uses an expression to distribute collections across multiple database servers

The following command defines a shard cluster that uses an expression on the field value **year** for distributing **collection3** across multiple database servers.

```
> db.runCommand({"shardCollection":"database1.collection3",
  key:{year:1},expressions:{"g_shard_server_1":"between 1980 and 1989",
  "g_shard_server_2":"between 1990 and 1999",
  "g_shard_server_3":"between 2000 and 2009",
  "g_shard_server_4":"remainder"}})
```

The name of the created shard-cluster definition is **sh_database3_collection3**.

- Inserted documents with values of **1980** to **1989** in the **year** field are sent to **g_shard_server_1**.
- Inserted documents with values of **1990** to **1999** in the **year** field are sent to **g_shard_server_2**.
- Inserted documents with values of **1980** to **1989** in the **year** field are sent to **g_shard_server_3**.
- Inserted documents with values below **1980** or above **2009** in the **year** field are sent to **g_shard_server_4**.

Shard cluster management

You can display information about shard cluster participants and about the shard cache on each shard server. You can add or remove shard servers from a shard cluster.

To display information about shard cluster participants, run the `db.runCommand` from the MongoDB shell, with `listShard` command syntax.

To display information about shard caches, run the `onstat -g shard` command.

Add a shard server

To add a shard server to the shard cluster, prepare the new shard server and add it to the shard cluster with the `addShard` command. Make sure to add the trusted host information for the new shard server to the existing shard servers.

Remove a shard server

To remove a shard server, run the `db.runCommand` from the MongoDB shell, with `removeShard` command syntax.

Change the sharding definition

After you add or remove a shard server, you might need to update the sharding definition:

- A definition that uses a hash algorithm to shard data is modified automatically.
- You must modify a sharding definition that uses an expression by running the `changeShardCollection` command.

When you change the sharding definition, existing data on shard servers is redistributed to match the new definition.

Changing the definition for a shard cluster

The `db.runCommand` command with `changeShardCollection` command syntax changes the definition for a shard cluster.

Before you begin

If the shard cluster uses an expression for distributing data across multiple database servers, you must add database servers to a shard cluster and remove database servers from a shard cluster by running the `changeShardCollection` command. If the shard-cluster definition uses a hash algorithm, database servers are automatically added to the shard cluster when you run the `sh.addShard` MongoDB shell command.

If you change a shard-cluster definition to include a new shard server, that server must first be added to a shard cluster by running the `db.runCommand` command with `addShard` command syntax.

When a shard-cluster definition changes, existing data on shard servers is redistributed to match the new definition.

About this task

The following steps apply to changing the definition for shard cluster that uses an expression for distributing documents in a collection across multiple database servers.

To change the definition for a shard cluster:

1. Run the mongo command.

Result

The command starts the MongoDB shell.

2. Change the shard-cluster definition by running the `changeShardCollection` command. You must redefine all expressions for all shard servers, not just newly added or changed shard servers.

```
db.runCommand ( { "changeShardCollection": " database. { collection | table } " , expressions: { " ER_group_name": " expression" , " ER_group_name": " remainder" } } )
```

Element	Description	Restrictions
<i>collection</i>	The name of the collection that is distributed across database servers.	The collection must exist.
<i>database</i>	The name of the database that contains the collection that is distributed across database servers.	The database must exist.
<i>ER_group_name</i>	The group name of a database server that receives copied data. The default group name for a database server is the database server's name prepended with <code>g_</code> . For example, the default	None.

Element	Description	Restrictions
	group name for a database server that is named myserver is g_myserver .	
<i>expression</i>	The expression that is used to select documents by shard key value.	None.
remainder	The database server that receives documents with shard key values that are not selected by expressions.	
<i>table</i>	The name of the table that is distributed across database servers.	The table must exist.

- For optimal query performance, connect to the wire listener and run the MongoDB `ensureIndex` command on the shard key each of a cluster's shard servers. The `ensureIndex` command ensures that an index for the collection or table is created on the shard server.

Example

Example

You have a shard cluster that is composed of three database servers, and the shard cluster is defined by the following command:

```
> db.runCommand({"shardCollection":"database1.collection1",
  expressions:{"g_shard_server_1":"in ('KS','MO')",
    "g_shard_server_2":"in ('CA','WA')","g_shard_server_3":"remainder"})
```

To add **g_shard_server_4** and **g_shard_server_5** to the shard cluster and change where data is sent to, run the following command:

```
> db.runCommand({"changeShardCollection":"database1.collection1",
  expressions:{"g_shard_server_1":"in ('KS','MO')",
    "g_shard_server_2":"in ('TX','OK')","g_shard_server_3":"in ('CA','WA')",
    "g_shard_server_4":"in ('OR','ID')","g_shard_server_5":"remainder"})
```

The new shard cluster contains five database servers:

- Inserted documents with a **state** field value of **KS** or **MO** are sent to **g_shard_server_1**.
- Inserted documents with a **state** field value of **TX** or **OK** are sent to **g_shard_server_2**.
- Inserted documents with a **state** field value of **CA** or **WA** are sent to **g_shard_server_3**.
- Inserted documents with a **state** field value of **OR** or **ID** are sent to **g_shard_server_4**.
- Inserted documents with a **state** field value that is not in the expression are sent to **g_shard_server_5**.

To then remove **g_shard_server_2** and change where the data that was on **g_shard_server_2** is sent to, run the following command:

```
> db.runCommand({"changeShardCollection":"database1.collection1",
  expressions:{"g_shard_server_1":"in ('KS','MO')",
    "g_shard_server_3":"in ('TX','CA','WA')",
```

```
"g_shard_server_4": "in ('OK', 'OR', 'ID')",
"g_shard_server_5": "remainder"}}
```

The new shard cluster contains four database servers.

- Inserted documents with a **state** field value of `TX` are now sent to **g_shard_server_3**.
- Inserted documents with a **state** field value of `OK` are now sent to **g_shard_server_4**.

Existing data on shard servers is redistributed to match the new definition.

Viewing shard-cluster participants

Run the `db.runCommand` MongoDB shell command with `listShards` syntax to list the group names, hosts, and port numbers of all shard servers in a shard cluster.

1. Run the mongo command.

Result

The command starts the MongoDB shell.

2. Run the `listShards` command:

Example

```
db.runCommand({listShards:1})
```

Results

The `listShards` command produces output in the following structure:

```
{
  "serverUsed" : "server_host/IP_address",
  "shards" : [
    {
      "_id" : "ER_group_name_1",
      "host" : "host_1:port_1"
    },
    {
      "_id" : "ER_group_name_2",
      "host" : "host_2:port_2"
    },
    {
      "_id" : "ER_group_name_x",
      "host" : "host_x:port_x"
    }
  ],
  "ok" : 1
}
```

ER_group_name

The group name of a shard server.

host

The host for a shard-cluster participant. The host can be a localhost name or a full domain name.

IP_address

The IP address of the database server that the listener is connected to.

port

The port number that a shard-cluster participant uses to communicate with other shard-cluster participants.

server_host

The host for the database server that the listener is connected to. The host can be a localhost name or a full domain name.

Example

Example

For this example, you have a shard cluster defined by the following command:

```
prompt> db.runCommand({"addShard":["myhost1.ibm.com:9201",  
    "myhost2.ibm.com:9202","myhost3.ibm.com:9203",  
    "myhost4.ibm.com:9204","myhost5.ibm.com:9205"]})
```

The following example output is shown when the listShards command is run in the MongoDB shell, and the listener is connected to the database server at myhost1.ibm.com.

Figure 1. listShards command output for a shard cluster

```
{  
  "serverUsed" : "myhost1.ibm.com/192.0.2.0:9200",  
  "shards" : [  
    {  
      "_id" : "g_myserver1",  
      "host" : "myhost1.ibm.com:9200"  
    },  
    {  
      "_id" : "g_myserver2",  
      "host" : "myhost2.ibm.com:9202"  
    },  
    {  
      "_id" : "g_myserver3",  
      "host" : "myhost3.ibm.com:9203"  
    },  
    {  
      "_id" : "g_myserver4",  
      "host" : "myhost4.ibm.com:9204"  
    },  
    {  
      "_id" : "g_myserver5",  
      "host" : "myhost5.ibm.com:9205"  
    }  
  ],  
  "ok" : 1  
}
```


MongoDB API

The Informix® support for MongoDB application programming interfaces and commands are described here.

Getting Started with Informix®'s MongoDB Solution

This topic covers the basics of getting started with using MongoDB API with Informix®.

How does Informix® support the MongoDB API?

The Informix® wire listener implements the MongoDB wire protocol. This allows MongoDB applications to connect to the Informix® wire listener and its associated Informix® database server. These applications communicate with Informix® as if it was a MongoDB server, with the wire listener acting as a translation layer between the MongoDB wire protocol and the SQL understood by the Informix® database server.

What are the components of the Informix® MongoDB solution?

There are three main components: the OneDB® server, the wire listener, and a MongoDB client. The wire listener is a mid-tier gateway server that enables communication between the MongoDB client and the OneDB® server.

How are JSON collections different from relational tables?

A JSON collection holds BSON (binary JSON) data. BSON documents have a flexible schema and can be used with unstructured data, meaning the structure and contents of BSON documents can differ from one document to another. This differs from relational tables where all rows must follow the same predefined structure.

Informix® fully supports JSON collections, which can be created through the wire listener. Additionally, the wire listener also makes it possible to run MongoDB queries against your traditional relational tables, using the same MongoDB API that you would use with JSON collections.

How do MongoDB commands map to SQL features?

MongoDB collection methods Informix SQL statements

<code>find</code>	<code>SELECT</code>
<code>save/insert</code>	<code>INSERT</code>
<code>remove</code>	<code>DELETE</code>
<code>update</code>	<code>UPDATE</code>
<code>ensureIndex</code>	<code>CREATE INDEX</code>
<code>sort</code>	<code>ORDER BY</code>
<code>limit</code>	<code>LIMIT/FIRST</code>

Commonly customizable wire listener properties

The properties that control the wire listener and the connection between the client and database server are set in the [wire listener configuration on page 9](#). The url parameter is required, but all other parameters are optional. Here are the commonly customized parameters.

`url`

This required parameter specifies the host name, database server, user ID, and password that are used in connections to the Informix® database server.

`authentication.enable`

This optional parameter indicates whether to enable user authentication. The default value is false.

`listener.port`

This optional parameter specifies the port number to listen on for incoming connections from MongoDB clients. The default value is 27017.

`security.sql.passthrough`

This optional parameter indicates whether to enable support for issuing SQL statements through the MongoDB API. The default value is false.

`sharding.enable`

This optional parameter indicates whether to enable the use of commands and queries on sharded data. The default value is false.

Starting the wire listener from the command line

You can [start the wire listener on page 44](#) by using a system command. For example:

```
java -jar onedb-wire-listener.jar
  -config onedb-wire-listener.properties
  -logFile onedb-wire-listener.log -loglevel info -start
```

MongoDB Create, read, update, and delete (CRUD) operations on collections and tables

These standard MongoDB CRUD operations are supported by Informix®:

- `insert`
- `find`
- `update`
- `remove`

This table shows an example of MongoDB operations and comparable SQL statements against relational tables. In the example, the retirement age of a customer is queried:

MongoDB operation

```
db.customer.insert( { name: "John", age: 65 } )

db.customer.find()

db.customer.find( {age: { $gt:65 } } )

db.customer.drop()

db.customer.ensureIndex( { name : 1, age : -1 } )

db.customer.remove( {age: { $gt:65 } } )

db.customer.update( { age: { $gt: 64 } }, { $set: { status:
"Retire" } }, { multi: true } )
```

Informix SQL statement

```
INSERT INTO customer (name, age) VALUES
("John", 65)

SELECT * FROM customer

SELECT * FROM customer WHERE age > 65

DROP TABLE customer

CREATE INDEX idx_1 on customer(name, age
DESC)

DELETE FROM customer where age > 65

UPDATE customer SET status = "Retire"
WHERE age > 64
```

Implicit operations for JSON collections and databases

If you insert into a non-existent JSON collection, a collection is implicitly created.

If you create a JSON collection in a non-existent database, a database is implicitly created.

Creating and listing indexes

You can use the MongoDB ensureIndex syntax to create an index that works for all data types. For example:

```
db.collection.ensureIndex( { zipcode: 1 } )

db.collection.ensureIndex( { state: 1, zipcode: -1} )
```

You can use the Informix® ensureIndex syntax to create an index for a specific data type. For example:

```
db.collection.ensureIndex( { zipcode : [1, "$int"] } )

db.collection.ensureIndex( { state: [1, "$string"], zipcode: [-1, "$int"] } )
```

You can list indexes by running the MongoDB `getIndexes` command.

Accessing multiple databases per connection

In standard OneDB® JDBC connections, you must specify the database name on the connection string and you must create one connection per database. In MongoDB, all messages include a fully qualified namespace that includes the database name and the collection. MongoDB connections are not associated with a particular database and each individual message or command specifies the intended database. A single MongoDB connection can switch between databases.

Moving data to and from collections and tables

You can run the MongoDB mongodump and mongoexport utilities to export data from MongoDB to OneDB®.

You can run the MongoDB `mongoexport` and `mongoimport` utilities to import data from MongoDB to OneDB®.

Viewing usage statistics

You can run the MongoDB `serverStatus` command to get the wire listener status information, including:

- Uptime
- Number of active and available connections
- Number of open cursors
- Total number of requests
- Counters for the number operations (queries, inserts, updates, deletes, commands, etc)

MongoDB to Informix® term mapping

The commonly used MongoDB terminology and concepts are mapped to the equivalent Informix® terminology and concepts.

The following table provides a summary of commonly used MongoDB terms and their Informix® conceptual equivalents.

Table 4. MongoDB concepts mapped to one or more Informix® concepts.

MongoDB concept	Informix® concept	Description
collection	table	This is the same concept. In Informix® this type of collection is sometimes referred to as a JSON collection. A JSON collection is similar to a relational database table, except it does not enforce a schema.
document	record	This is the same concept. In Informix®, this type of document is sometimes referred to as a JSON document.
field	column	This is the same concept.
master / slave	primary server / secondary server	This is the same concept. However, Informix® secondary servers have additional capabilities. For example, data on a secondary server can be updated and propagated to primary servers.
replica set	high-availability cluster	This is the same concept. However, when the replica set is updated, it is then sent to all servers, not only to the primary server.
sharded cluster	shard cluster	This is the same concept. In Informix®, a shard cluster is a group of servers (sometimes called shard servers) that contain sharded data.
shard key	shard key	This is the same concept.

Language drivers

The wire listener parses messages that are based on the MongoDB Wire Protocol.

You can use the MongoDB community drivers to store, update, and query JSON documents with Informix® as a JSON data store. These drivers can include Java™, C/C++, Ruby, PHP, PyMongo, and so on.

Download the MongoDB drivers for the programming languages at <http://docs.mongodb.org/ecosystem/drivers/>.

Command utilities and tools

You can use the MongoDB shell and any of the standard MongoDB command utilities and tools.

You can use the MongoDB shell to run interactive queries and operations against Informix. You can use any version of the MongoDB shell that supports the [mongo.api.version on page 22](#) configured for the wire listener.

You can run the MongoDB mongoexport, mongoimport, mongodump, and mongorestore utilities to import and export data to or from Informix®.

Collection methods

Informix® supports a subset of the MongoDB collection methods.

The collection methods are run on a JSON collection or a relational table. The syntax for collection methods in the **mongo** shell is `db.collection_name.collection_method()`, where `db` refers to the current database, `collection_name` is the name of the JSON collection or relational table, `collection_method` is the MongoDB collection method. For example, `db.cartype.count()` determines the number of documents that are contained in the **cartype** collection.

Table 5. Supported collection methods

Collection method	JSON collections	Relational tables	Details
aggregate	No	No	
count	Yes	Yes	
createIndex	Yes	Yes	For more information, see Index creation on page 71 .
dataSize	Yes	No	
distinct	Yes	Yes	
drop	Yes	Yes	
dropIndex	Yes	Yes	
dropIndexes	Yes	No	
ensureIndex	Yes	Yes	For more information, see Index creation on page 71 .

Table 5. Supported collection methods (continued)

Collection method	JSON collections	Relational tables	Details
find	Yes	Yes	You can use the Informix® \$nativeCursor query modifier with the addSpecial function.
findAndModify	Yes	Yes	For relational tables, findAndModify is supported only for tables that have a primary key. This method is not support sharded data.
findOne	Yes	Yes	
getIndexes	Yes	No	
getShardDistribution	No	No	
getShardVersion	No	No	
getIndexStats	No	No	
group	No	No	
indexStats	No	No	
insert	Yes	Yes	
isCapped	Yes	Yes	This command returns false because capped collections are not supported in Informix®.
mapReduce	No	No	
reIndex	No	No	
remove	Yes	Yes	The justOne option is not supported. This command deletes all documents that match the query criteria.
renameCollection	No	No	
save	Yes	No	
stats	Yes	No	
storageSize	Yes	No	
totalSize	Yes	No	
update	Yes	Yes	The multi option is supported for JSON collections if <code>update.one.enable=true</code> in the wire listener properties file. For relational tables, the multi-parameter is ignored and all documents that meet the query criteria are updated. If <code>update.one.enable=false</code> , all documents that match the query criteria are updated.
validate	No	No	

For more information about the MongoDB features, see <http://docs.mongodb.org/manual/reference/>.

Index creation

Informix® supports the creation of indexes on collections and relational tables by using the MongoDB API and the wire listener.

- [Index creation by using the MongoDB syntax on page 71](#)
- [Index creation for a specific data type by using the Informix extended syntax on page 71](#)
- [Index creation for arrays using the Informix extended syntax on page 72](#)
- [Index creation for text, geospatial, and hashed on page 73](#)

Index creation by using the MongoDB syntax

For JSON collections and relational tables, you can use the MongoDB `createIndex` and `ensureIndex` syntax to create an index that works for all data types. For example:

```
db.collection.createIndex( { zipcode: 1 } )
db.collection.createIndex( { state: 1, zipcode: -1} )
```



Tip: If you are creating an index for a JSON collection on a field that has a fixed data type, you can get the best query performance by using the Informix® extended syntax.

The following options are supported:

- name
- unique

The following options are not supported:

- background
- default_language
- dropDups
- expireAfterSeconds
- language_override
- sparse
- v
- weights

Index creation for a specific data type by using the Informix® extended syntax

You can use the Informix® `createIndex` or `ensureIndex` syntax on collections to create an index for a specific data type. For example:

```
db.collection.createIndex( { zipcode : [1, $int] } )
db.collection.createIndex( { state: [1, $string], zipcode: [-1, $int] } )
```

This syntax is supported for collections only. It not supported for relational tables.



Tip: If you are creating an index on a field that has a fixed data type, you can get better query performance by using the Informix® `createIndex` or `ensureIndex` syntax.

The following data types are supported:

- `$bigint`
- `$binary`
- `$boolean`
- `$date`
- `$double` [2 on page 72](#)
- `$int` [3 on page 72](#)
- `$integer` [3 on page 72](#)
- `$lvarchar` [1 on page 72](#)
- `$number` [2 on page 72](#)
- `$string` [1 on page 72](#)
- `$timestamp`
- `$varchar`



Notes:

1. `$string` and `$lvarchar` are aliases and create `lvarchar` indexes.
2. `$number` and `$double` are aliases and create double indexes.
3. `$int` and `$integer` are aliases.

Index creation for arrays using the Informix extended syntax

You can use the Informix® `createIndex` or `ensureIndex` syntax on collections to create an index for arrays. For example:

```
db.collection.createIndex( { "my_array" : [ 1, "$array", "$int" ] } )
```

which creates an integer array index on the field named “my_array”.

This syntax is similar to the Informix extended typed syntax. Specify the type of the index as “\$array?” and then provide a third argument specifying the data type stored in the array.



Note: This syntax is supported for collections only. It is not supported for relational tables.

The following data types are supported with array indexes:

- `$bigint`
- `$date`
- `$double` [1 on page 72](#)

- \$int^{2 on page 72}
- \$integer^{2 on page 72}
- \$number^{1 on page 72}
- \$varchar



Notes:

1. \$number and \$double are aliases and create double indexes.
2. \$int and \$integer are aliases.

Index creation for text, geospatial, and hashed

Text indexes

Text indexes are supported. You can search string content by using text search in documents of a collection.

You can create text indexes by using the MongoDB or Informix® syntax. For example, here is the MongoDB syntax:

```
db.articles.ensureIndex( { abstract: "text" } )
```

The Informix® syntax provides additional support for the Informix® basic text search functionality. For more information, see [createTextIndex on page 82](#).

Geospatial indexes

2dsphere indexes are supported by using the GeoJSON objects, but not the MongoDB legacy coordinate pairs.

2d indexes are not supported.

Hashed indexes

Hashed indexes are not supported. If a hashed index is specified, a regular untyped index is created.

For more information about the MongoDB features, see <http://docs.mongodb.org/manual/reference/>.

Database commands

Informix® supports a subset of the MongoDB database commands.

The basic syntax for database commands in the **mongo** shell is `db.command()`, where *db* refers to the current database, and *command* is the database command. You can use the **mongo** shell helper method **db.runCommand()** to run database commands on the current database.

- [User commands on page 74](#)
- [Database operations on page 75](#)

User commands

Aggregation commands

Table 6. Aggregation commands

MongoDB command	JSON collecti ons	Relational tables	Details
aggregate	Yes	Yes	The wire listener supports version 2.4 of the MongoDB aggregate command, which returns a command result. For more information, see Aggregation framework operators on page 101 .
count	Yes	Yes	
distinct	Yes	Yes	
group	No	No	
mapReduce	No	No	

Geospatial commands

Table 7. Geospatial commands

MongoDB command	JSON collecti ons	Relational tables	Details
geoNear	Yes	No	Supported by using the GeoJSON format. The MongoDB legacy coordinate pairs are not supported.
geoSearch	No	No	
geoWalk	No	No	

Query and write operation commands

Table 8. Query and write operation commands

MongoDB command	JSON collections	Relational tables	Details
delete	Yes	Yes	
eval	No	No	
findAndModify	Yes	Yes	For relational tables, the findAndModify command is supported only for tables that have a primary key. This command does not support sharded data.

Table 8. Query and write operation commands (continued)

MongoDB command	JSON collections	Relational tables	Details
getLastError	Yes	Yes	
getPrevError	No	No	
insert	Yes	Yes	
resetError	No	No	
text	No	No	Text queries are supported by using the \$text or \$ifxtext query operators, not through the text command.
update	Yes	Yes	

Database operations

Authentication commands

Table 9. Authentication commands

Name	Supported	Details
authenticate	Yes	
authSchemaUpgrade	Yes	This command upgrades user data to MongoDB API version 2.6 or higher.
logout	Yes	
getnonce	Yes	

User management commands

Table 10. User management commands

Name	Supported	Details
createUser	Yes	Supported for MongoDB API version 2.6 or higher.
dropAllUsersFromDatabase	Yes	Supported for MongoDB API version 2.6 or higher.
dropUser	Yes	Supported for MongoDB API version 2.6 or higher.
grantRolesToUser	Yes	Supported for MongoDB API version 2.6 or higher.
revokeRolesFromUser	Yes	Supported for MongoDB API version 2.6 or higher.
updateUser	Yes	Supported for MongoDB API version 2.6 or higher.
usersInfo	Yes	Supported for MongoDB API version 2.6 or higher.

Role management commands

Table 11. Role management commands

Name	Supported	Details
createRole	Yes	Supported for MongoDB API version 2.6 or higher.
dropAllRolesFromDatabase	Yes	Supported for MongoDB API version 2.6 or higher.
dropRole	Yes	Supported for MongoDB API version 2.6 or higher.
grantPrivilegesToRole	Yes	Supported for MongoDB API version 2.6 or higher.
grantRolesToRole	Yes	Supported for MongoDB API version 2.6 or higher.
invalidateUserCache	No	
rolesInfo	Yes	Supported for MongoDB API version 2.6 or higher.
revokePrivilegesFromRole	Yes	Supported for MongoDB API version 2.6 or higher.
revokeRolesFromRole	Yes	Supported for MongoDB API version 2.6 or higher.
updateRole	Yes	Supported for MongoDB API version 2.6 or higher.

Diagnostic commands

Table 12. Diagnostic commands

Name	Supported	Details
buildInfo	Yes	Whenever possible, the Informix® output fields are identical to MongoDB. There are additional fields that are unique to Informix®.
collStats	Yes	The value of any field that is based on the collection size is an estimate, not an exact value. For example, the value of the field 'size' is an estimate.
connPoolStats	No	
cursorInfo	No	
dbStats	Yes	The value of any field that is based on the collection size is an estimate, not an exact value. For example, the value of the field 'dataSize' is an estimate.
features	Yes	
getCmdLineOpts	Yes	

Table 12. Diagnostic commands (continued)

Name	Supported	Details
getLog	No	
hostInfo	Yes	The <code>memSizeMB</code> , <code>totalMemory</code> , and <code>freeMemory</code> fields indicate the amount of memory that is available to the Java™ virtual machine (JVM) that is running, not the operating system values.
indexStats	No	
listCommands	Yes	
listDatabases	Yes	

The value of any field that is based on the collection size is an estimate, not an exact value. For example, the value of the field 'sizeOnDisk' is an estimate.

The `listDatabases` command estimates the size of all collections and collection indexes for each database. However, relational tables and indexes are excluded from this size calculation.



Important: The `listDatabases` command performs expensive and CPU-intensive computations on the size of each database in the Informix® instance. You can decrease the expense by using the `sizeStrategy` option.

sizeStrategy

You can use this option to configure the strategy for calculating database size when the `listDatabases` command is run.

```
sizeStrategy: { { estimate | { estimate:n } | compute | none |
perDatabaseSpace } }
```


estimate

Estimate the size of the documents in the collection by using 1000 (or 0.1%) of the documents. This is the default value.

The following example estimates the collection size by using the default of 1000 (or 0.1%) of the documents:

```
db.runCommand({listDatabases:1,
sizeStrategy:"estimate"})
```

Table 12. Diagnostic commands (continued)

Name	Supported	Details
		<p>estimate: <i>n</i></p> <p>Estimate the size of the documents in a collection by sampling one document for every <i>n</i> documents in the collection.</p> <p>The following example estimates the collection size by using sample size of 0.5% or 1/200th of the documents:</p> <pre>db.runCommand({listDatabases:1, sizeStrategy:{estimate:200}})</pre> <p>compute</p> <p>Compute the exact size of each database.</p> <pre>db.runCommand({listDatabases:1, sizeStrategy:"compute"})</pre> <p>none</p> <p>List the databases but do not compute the size. The database size is listed as 0.</p> <pre>db.runCommand({listDatabases:1, sizeStrategy:"none"})</pre> <p>perDatabaseSpace</p> <p>Calculate the size of a database by adding the sizes for all dbspaces, sbspaces, and blobspaces that are assigned to the tenant database.</p> <p> Important: The perDatabaseSpace option applies only to tenant databases that are created by the multi-tenancy feature.</p> <pre>db.runCommand({listDatabases:1 , sizeStrategy:"perDatabaseSpace"})</pre>
ping	Yes	
serverStatus	Yes	
top	No	
whatsmyuri	Yes	

Instance administration commands

Table 13. Instance administration commands

Name	JSON collections	Relational tables	Details
clone	No	No	
cloneCollection	No	No	
cloneCollectionAsCapped	No	No	
collMod	No	No	
compact	No	No	
convertToCapped	No	No	
copydb	No	No	
create	Yes	No	Informix® does not support the following flags: <ul style="list-style-type: none"> • capped • autoIndexID • size • max
createIndexes	Yes	Yes	
drop	Yes	Yes	Informix® does not lock the database to block concurrent activity.
dropDatabase	Yes	Yes	
dropIndexes	Yes	No	The MongoDB deleteIndexes command is equivalent.
filemd5	Yes	Yes	
fsync	No	No	
getParameter	No	No	
listCollections	Yes	Yes	The includeRelational and includeSystem flags are supported to include or exclude relational or system tables in the results.

Table 13. Instance administration commands (continued)

Name	JSON collections	Relational tables	Details
			Default is includeRelational=true and includeSystem=false.
listIndexes	Yes	Yes	
logRotate	No	No	
reIndex	No	No	
renameCollection	No	No	
repairDatabase	No	No	
setParameter	No	No	
shutdown	Yes	Yes	<p>The timeoutSecs flag is supported. In the Informix®, the timeoutSecs flag determines the number of seconds that the wire listener waits for a busy client to stop working before forcibly terminating the session.</p> <p>The force flag is not supported.</p>
touch	No	No	

Replication commands**Table 14. Replication commands**

Name	Supported
isMaster	Yes
replSetFreeze	No
replSetGetStatus	No
replSetInitiate	No
replSetMaintenance	No
replSetReconfig	No
replSetStepDown	No
replSetSyncFrom	No

Table 14. Replication commands (continued)

Name	Supported
Resync	No

Sharding commands**Table 15. Replication commands**

Name	JSON collections	Relational tables	Details
addShard	Yes	Yes	<p>The MongoDB maxSize and name options are not supported.</p> <p>In addition to the MongoDB command syntax for adding a single shard server, you can use the Informix® specific syntax to add multiple shard servers in one command by sending the list of shard servers as an array. For more information, see Creating a shard cluster with MongoDB commands.</p>
enableSharding	Yes	Yes	This action is not required for Informix® and therefore this command has no affect for Informix®.
flushRouterConfig	No	No	
isdbgrid	Yes	Yes	
listShards	Yes	Yes	The equivalent Informix® command is cdr list server .
movePrimary	No	No	
removeShard	No	No	
shardCollection	Yes	Yes	<p>The equivalent Informix® command is cdr define shardCollection.</p> <p>The MongoDB unique and numInitialChunks options are not supported.</p>
shardingState	No	No	
split	No	No	

For more information about the MongoDB features, see <http://docs.mongodb.org/manual/reference/>.

Informix® JSON commands

The Informix® JSON commands are available in addition to the supported MongoDB commands. These commands enable functionality that is supported by Informix® and they are run by using the MongoDB API.

The syntax for using Informix® commands in the MongoDB shell is:

```
db.runCommand({command_document})
```

The *command_document* contains the Informix® command and any parameters.

- [createTextIndex on page 82](#)
- [exportCollection on page 83](#)
- [importCollection on page 85](#)
- [killCursors on page 85](#)
- [lockAccounts on page 86](#)
- [runFunction on page 87](#)
- [runProcedure on page 87](#)
- [transaction on page 88](#)
- [unlockAccounts on page 90](#)

createTextIndex

Create Informix® **bts** indexes.



Important: If you create text indexes by using the Informix® `createTextIndex` command, you must query them by using the Informix® `$ifxtext` query operator. If you create text indexes by using the MongoDB syntax for text indexes, you must query them by using the MongoDB `$text` query operator.

```
createTextIndex : " collection_name " , name : " indexName " [ , key : { " column " } ] , options : { [ <btX index  
parameters> (explicit id ) ] }
```

createTextIndex

This required parameter specifies the name of the collection or relational table where the **bts** index is created.

name

This required parameter specifies the name of the **bts** index.

options

This required parameter specifies the name-value pairs for the **bts** parameters that are used when creating the index. If no parameter values are required, you can specify an empty document.

Use **bts** index parameters to customize the behavior of the index and how text is indexed. Include JSON index parameters to control how JSON and BSON documents are indexed. For example, you can index the documents as field name-value pairs instead of as unstructured text so that you can search for text by field. The name and values of the **bts** index parameters in the options parameter are the same as the syntax for

creating a **bts** access method with the SQL CREATE INDEX statement. The following example creates an index named `articlesIdx` on the `articles` collection by using the **bts** parameter `all_json_names="yes"`:

```
db.runCommand({
  createTextIndex:"articles",
  name:"articlesIdx",
  options:{all_json_names:"yes"}})
```

key

This parameter is required if you are indexing relational tables, but optional if you are indexing collections. This parameter specifies which columns to index for relational tables.

The following example creates an index named `myidx` in the `mytab` relational table on the `title` and `abstract` columns:

```
db.runCommand({
  createTextIndex:"mytab",
  name:"myidx",
  key:{"title":"text","abstract":"text"},
  options:{}})
```

exportCollection

Export JSON collections from the wire listener to a file.

```
exportCollection : " collection_name " , file : " filepath " , format : { { " json " | " jsonArray " } [ , fields : { " filter " } ] | " csv " , fields : { " filter " } } [ , query : { " query_document " } ]
```

exportCollection

This required parameter specifies the collection name to export.

file

This required parameter specifies the output file path on the host machine where the wire listener is running.

For example:

- UNIX™ is `file:"/tmp/export.out"`
- Windows™ is `file:"C:/temp/export.out"`

format

This required parameter specifies the exported file format.

json

Default. The `.json` file format. One JSON-serialized document per line is exported.

The following command exports all documents from the collection that is named `c` by using the `json` format:

```
> db.runCommand({exportCollection:"c",file:"/tmp/export.out",
  ,format:"json"})
{
```

```

    "ok":1,
    "n":1000,
    "millis":NumberLong(119),
    "rate":8403.361344537816
  }

```

Where `"n"` is the number of documents that are exported, `"millis"` is the number of milliseconds it took to export, and `"rate"` is the number of documents per second that are exported.

jsonArray

The `.jsonArray` file format. This format exports an array of JSON-serialized documents with no line breaks. The array format is JSON-standard.

The following command exports all documents from the collection `c` by using the `jsonArray` format:

```

> db.runCommand({exportCollection:"c",file:"/tmp/export.out"
, format:"jsonArray"})
{
  "ok":1,
  "n":1000,
  "millis":NumberLong(81),
  "rate":12345.67901234568
}

```

Where `"n"` is the number of documents that are exported, `"millis"` is the number of milliseconds it took to export, and `"rate"` is the number of documents per second that are exported.

csv

The `.csv` file format. Comma-separated values are exported. You must specify which fields to export from each document. The first line of the `.csv` file contains the fields and all subsequent lines contain the comma-separated document values.

fields

This parameter specifies which fields are included in the output file. This parameter is required for the `csv` format, but optional for the `json` and `jsonArray` formats.

The following command exports all documents from the collection that is named `c` by using the `csv` format, only output the `"_id"` and `"name"` fields:

```

> db.runCommand({exportCollection:"c",file:"/tmp/export.out"
,format:"csv",fields:{"_id":1,"name":1}})
{
  "ok":1,
  "n":1000,
  "millis":NumberLong(57),
  "rate":17543.859649122805
}

```

Where `"n"` is the number of documents that are exported, `"millis"` is the number of milliseconds it took to export, and `"rate"` is the number of documents per second that are exported.

query

This optional parameter specifies a query document that identifies which documents are exported. The following example exports all documents from the collection that is named `c` that have a `qty` field that is less than 100:

```
> db.runCommand({exportCollection:"c",file:"/tmp/export.out",
,format:"json",query:{"qty":{"$lt":100}}})
{"ok":1,"n":100,"millis":NumberLong(5),"rate":20000}
```

importCollection

Import JSON collections from the wire listener to a file.

```
importCollection : " collection_name " , file : " filepath " , format : " { json | jsonArray | csv } "
```

importCollection

The required parameter specifies the collection name to import.

file

This required parameter specifies the input file path. For example, `file: "/tmp/import.json"`.



Important: The input file must be on the same host machine where the wire listener is running.

format

This required parameter specifies the imported file format.

json

Default. The `.json` file format.

The following example imports documents from the collection that is named `c` by using the `json` format:

```
> db.runCommand({importCollection:"c",file:"/tmp/import.out",
,format:"json"})
```

jsonArray

The `.jsonArray` file format.

The following example imports documents from the collection `c` by using the `jsonArray` format:

```
> db.runCommand({exportCollection:"c",file:"/tmp/import.out",
,format:"jsonArray"})
```

csv

The `.csv` file format.

killCursors

Close native cursors that were created with the `&nativeCursor` query modifier in a REST API query.

```
killCursors : 1 cursorIds : [ cursorID ]
```

killCursors

This required parameter closes native cursors.

cursorIds

This required parameter lists the native cursor IDs to close.

The following command closes the cursor with the ID of 22:

```
GET /dbname/$cmd?query={killCursors:1, cursorIds:[22]}
```

lockAccounts

Lock a database or user account.



Important:

- To run this command, you must be the instance administrator.
- If you specify the `lockAccounts:1` command without specifying a **db** or **user** argument, all accounts in all databases are locked.

```
lockAccounts : { 1 [ { , db : { " database_name " | [ " database_name " ] | { " $regex " : " json_document " } | { { | " include " : { " database_name " | [ " database_name " ] | { " $regex " : " json_document " } } | " exclude " : { " database_name " | [ " database_name " ] | { " $regex " : " json_document " } } } } " | , user : { " user_name " | " json_document " } } ] }
```

lockAccounts:1

This required parameter locks a database or user account.

db

This optional parameter specifies the database name of an account to lock. For example, to lock all accounts in database that is named `foo`:

```
> db.runCommand({lockAccounts:1,db:"foo"})
```

exclude

This optional parameter specifies the databases to exclude. For example, to lock all accounts on the system except the accounts that are in the databases named `alpha` and `beta`:

```
> db.runCommand({lockAccounts:1,db:{'exclude':['alpha','beta']}})
```

include

This optional parameter specifies the databases to include. For example, to lock all accounts in the databases named `delta` and `gamma`:

```
> db.runCommand({lockAccounts:1,db:{'include':['delta','gamma']}})
```

\$regex

This optional MongoDB evaluation query operator selects values from a specified JSON document. For example, to lock accounts for databases that begin with the character `a`, and end in `e`:

```
> db.runCommand({lockAccounts:1,db:{"$regex":"a.*e"}})
```

user

This optional parameter specifies the user accounts to lock. For example, to lock the account of all users that are not named `alice`:

```
> db.runCommand({lockAccounts:1,user:{$ne:"alice"}});
```

runFunction

Run an SQL function through the wire listener. This command is equivalent to the SQL statement EXECUTE FUNCTION.

```
runFunction : " function_name " [ , " arguments " : [ argument ] ]
```

runFunction

This required parameter specifies the name of the SQL function to run. For example, a **current** function returns the current date and time:

```
> db.runCommand({runFunction:"current"})
{"returnValue": 2016-04-05 12:09:00, "ok":1}
```

arguments

This parameter specifies an array of argument values to the function. You must provide as many arguments as the function requires. For example, an **add_values** function requires two arguments to add together:

```
> db.runCommand({runFunction:"add_values", "arguments":[3,6]})
{"returnValue": 9, "ok":1}
```

The following example returns multiple values from a **func_return3** function:

```
> db.runCommand({runFunction:"func_return3", "arguments":[101]})
{"returnValue": {"serial_num":1103, "name":"Newton", "points":100}, "ok":1}
```

runProcedure

Run an SQL stored procedure through the wire listener. This command is equivalent to the SQL statement EXECUTE PROCEDURE.

```
runProcedure : " procedure_name " [ , " arguments " : [ argument ] ]
```

runProcedure

This required parameter specifies the name of the SQL procedure to run. For example, a **colors_list** stored procedure, which uses a WITH RESUME clause in its RETURN statement, returns multiple rows about colors:

```
> db.runCommand({runProcedure:"colors_list"})
{"returnValue": [
  {"color" : "Red", "hex" : "FF0000"},
```

```
{ "color" : "Blue", "hex" : "0000A0" },
{ "color" : "White", "hex" : "FFFFFF" }
], "ok" : 1}
```

arguments

This parameter specifies an array of argument values to the procedure. You must provide as many arguments as the procedure requires. For example, an **increase_price** procedure requires two arguments to identify the original price and the amount of increase:

```
> db.runCommand({runProcedure:"increase_price", "arguments":[101, 10]})
{"ok":1}
```

transaction

Enable or disable transaction support for a session, run a batch transaction, or, when transaction support is enabled, commit or rollback transactions. This command binds or unbinds a connection to the current MongoDB session in a database. The relationship between a MongoDB session and the Informix® JDBC connection is not static.



Important: This command is not supported for queries that are run on shard servers.

```
transaction : { " enable " | " disable " | " commit " | " rollback " | " execute " , " commands " : [ command_docs ] [ , " finally "
: [ command_docs ] ] | " status " }
```

enable

This optional parameter enables transaction mode for the current session in the current database. The following example shows how to enable transaction mode:

```
> db.runCommand({transaction:"enable"})
{"ok":1}
```

disable

This optional parameter disables transaction mode for the current session in the current database. The following example shows how to disable for transaction mode:

```
> db.c.find()
{"_id":ObjectId("52a8f9c477a0364542887ed4"),"a":1}
> db.runCommand({transaction:"disable"})
{"ok":1}
```

commit

If transactions are enabled, this optional parameter commits the current transaction. If transactions are disabled, an error is shown. The following example shows how to commit the current transaction:

```
> db.c.insert({"a":1})
> db.runCommand({transaction:"commit"})
{"ok":1}
```

rollback

If transactions are enabled, this optional parameter rolls back the current transaction. If transactions are disabled, an error is shown. The following example shows how to roll back the current transaction:


```

> db.c.insert({"a":2})
> db.c.find()
{"_id":ObjectId("52a8f9c477a0364542887ed4"),"a":1}
{"_id":ObjectId("52a8f9e877a0364542887ed5"),"a":2}
> db.runCommand({transaction:"rollback"})
{"ok":1}

```

execute

This optional parameter runs a batch of commands as a single transaction. If transaction mode is not enabled for the session, this parameter enables transaction mode for the duration of the transaction.

The list of command documents can include insert, update, delete, findAndModify, and find command documents. In insert, update, and delete command documents, you cannot set the **ordered** property to false. You can use a find command document to run queries, including SQL queries, but not commands. A find command document can include the **\$orderby**, **limit**, **skip**, and **sort** operators. The following example deletes a document from the inventory collection and inserts documents into the archive collection:

```

> db.runCommand({"transaction" : "execute",
  "commands" : [
    {"delete":"inventory", "deletes" : [ { "q" : { "_id" : 432432 } } ] },
    {"insert" : "archive",
      "documents" : [ { "_id": 432432, "name" : "apollo", "last_status" : 9} ]
    }
  ]
})

```

Include the optional **finally** argument if you have a set of command documents to run at the end of the transaction regardless of whether the transaction is successful. The following example runs a query with the . The command document for the **finally** argument unsets the USE_DWA environment variable regardless of whether the previous query succeeds.

```

> db.runCommand({"transaction" : "execute",
  "commands" : [
    {"find" : "system.sql", "filter" : {"$sql" :
      "SET ENVIRONMENT USE_DWA 'ACCELERATE ON'" } },
    {"find" : "system.sql", "filter" : {"$sql" :
      "SELECT SUM(s.amount) as sum FROM sales AS s
      WHERE s.prid = 100 GROUP BY s.zip" } }
  ],
  "finally" : [{"find":"system.sql", "filter" : {"$sql" :
    "SET ENVIRONMENT USE_DWA 'ACCELERATE OFF'" } } ]
})

```

status

This optional parameter prints status information to indicate whether transaction mode is enabled, and if transactions are supported by the current database. The following example shows how to print status information:

```

> db.runCommand({transaction:"status"})
{"enabled":true,"supports":true,"ok":1}

```

unlockAccounts

Unlock a database or user account.



Important:

- To run this command, you must be the instance administrator.
- If you specify the `unlockAccounts:1` command without specifying a **db** or **user** argument, all accounts in all databases are unlocked.

```
unlockAccounts : { { 1 [ { , db : { " database_name " | [ " database_name " ] | { " $regex " : " json_document " } | { |  
" include " : { " database_name " | [ " database_name " ] | { " $regex " : " json_document " } } | " exclude " : { "  
database_name " | [ " database_name " ] | { " $regex " : " json_document " } } } } " | , user : { " user_name " | "  
json_document " } } ] } }
```

unlockAccounts:1

This required parameter unlocks a database or user account.

db

This optional parameter specifies the database name of an account to unlock. For example, to unlock all accounts in database that is named `foo`:

```
> db.runCommand({unlockAccounts:1,db:"foo"})
```

exclude

This optional parameter specifies the databases to exclude. For example, to unlock all accounts on the system except the accounts that are in the databases named `alpha` and `beta`:

```
> db.runCommand({unlockAccounts:1,db:{"exclude":["alpha","beta"]}})
```

include

This optional parameter specifies the databases to include. For example, to unlock all accounts in the databases named `delta` and `gamma`:

```
> db.runCommand({unlockAccounts:1,db:{"include":["delta","gamma"]}})
```

\$regex

This optional MongoDB evaluation query operator selects values from a specified JSON document. For example, to unlock accounts for databases that begin with the character `a`, and end in `e`:

```
> db.runCommand({unlockAccounts:1,db:{ "$regex": "a.*e" }})
```

user

This optional parameter specifies the user accounts to unlock. For example, to unlock the account of all users that are not named `alice`:

```
> db.runCommand({unlockAccounts:1,user:{$ne:"alice"}});
```

Running Informix queries through the MongoDB API

You can use MongoDB API commands through the wire listener to query collections and relational tables, run SQL commands, and run queries that join collections and relational tables.

Running SQL commands by using the MongoDB API

You can run SQL statements by using the MongoDB API and retrieve results back. The results of the SQL statements are treated like they are documents in a JSON collection.

Before you begin

You must enable SQL operations by setting **security.sql.passthrough=true** in the wire listener properties file.

From the MongoDB shell command, use the abstract system collection `system.sql` as the collection name and `$sql` as the query operator, followed by the SQL statement.

For example:

```
> db.getCollection("system.sql").find({ "$sql": "sql_statement" })
```

To use host variables, include question marks in the SQL statement, and include the `$bindings` operator with an array that contains a value for each host variable in order of appearance. For example:

```
> db.getCollection("system.sql").find({ "$sql": "sql_statement",
"$bindings": [values] })
```

Example

Examples

Create an SQL table

In this example, an SQL table is created by running the Informix® CREATE TABLE command by using the MongoDB API:

```
> db.getCollection("system.sql").find({ "$sql": "create table foo
(c1 int)" })
```

Drop an SQL table

In this example, an SQL table is dropped by running the Informix® DROP TABLE command by using the MongoDB API:

```
> db.getCollection("system.sql").find({ "$sql": "drop table foo" })
```

Delete SQL customer call records that are more than 5 years old

In this example, customer call records stored in SQL tables are deleted by running the Informix® DELETE command by using the MongoDB API:

```
> db.getCollection("system.sql").findOne({ "$sql": "delete from
cust_calls where (call_dtime + interval(5) year to year) < current" })
```

Result: 7 rows were deleted.

```
{ "n" : 7 }
```

Join JSON collections

In this example, a query counts the number of orders customers placed by using an outer join to include the customers who did not place orders.

```
> db.getCollection("system.sql").find({ "$sql": "select
c.customer_num,o.customer_num as order_cust,count(order_num) as
order_count from customer c left outer join orders o on
c.customer_num = o.customer_num group by 1, 2 order by 2" })
```

Result:

```
{ "customer_num" : 113, "order_cust" : null, "order_count" : 0 }
{ "customer_num" : 114, "order_cust" : null, "order_count" : 0 }
{ "customer_num" : 101, "order_cust" : 101, "order_count" : 1 }
{ "customer_num" : 104, "order_cust" : 104, "order_count" : 4 }
{ "customer_num" : 106, "order_cust" : 106, "order_count" : 2 }
```

Delete rows based on a host variable

In this example, the statement includes a host variable that specifies to delete the rows that have the name "john".

```
> db.getCollection("system.sql").find({ "$sql": "delete from mytab
where name = ?", "$bindings" : ["john"] })
```

Run a user-defined function with host variables

In this example, the statement runs a user-defined routine with two host variables to raise prices.

```
> db.getCollection("system.sql").find({ "$sql": "execute function
raise_price(?, ?)", "$bindings" : [101, 0.10] })
```

Running MongoDB operations on relational tables

You can run MongoDB operations on relational tables by using the MongoDB API.

About this task

Use the MongoDB database methods to run read and write operations on a relational table as if the table were a collection. The wire listener examines the database and if the accessed entity is a relational table, it converts the basic operations on that table to SQL and converts the returned values into a JSON document. At the first access to an entity, the wire listener caches the name and type of that entity. The first access results in an extra call to the Informix® server, but subsequent operations do not.

From the MongoDB API, enter the relational table name as the collection name in the MongoDB collection method. For example:

```
>db.getCollection("tablename");
```

Example

Examples

The following examples use the **customer** table in the **stores_demo** sample database. All of the tables in the **stores_demo** database are relational tables, but you can use the same MongoDB collection methods to access and modify the tables, as if they were collections.

Get the customer count

In this example, the number of customers is returned.

```
> db.customer.count()
28
```

Query for a particular customer

In this example, a specific customer record is retrieved.

```
> db.customer.find({customer_num:101})
{ "customer_num" : 101, "fname" : "Ludwig", "lname" : "Pauli", "company" :
  "All Sports Supplies", "address1" : "213 Erstwilt Court", "address2" :
  null, "city" : "Sunnyvale", "state" : "CA", "zipcode" : "94086",
  "phone" : "408-555-8075" }
```

Update a customer phone number

In this example, the customer phone number is updated.

```
> db.customer.update({"customer_id":101}, {"$set":{"phone":"408-555-1234"}})
```

Running join queries by using the wire listener

You can use the wire listener to run join queries on JSON and relational data. The syntax supports collection-to-collection joins, relational-to-relational joins, and collection-to-relational joins. Join queries are supported in sharded environments when parallel sharded queries are enabled.

About this task

Join queries in the wire listener are done by submitting a join query document to the **system.join** pseudo table.

- Wire listener join queries support the sort, limit, skip, and explain options that you can set on a MongoDB cursor.
- Fields that are specified in the sort clause must also be included in the projection clause.
- The **\$hint** operator is not supported.

1. Create a join query document.

Example

The join query document has the following syntax:

```
{ "$collections" : { "table_or_collection_name" : { "$project" : { specifications } [ , "$where" : { filter } ] } , "$condition" :
{ { "tabName1.column" : "tabName2.column" | "tabName1.column" : [ "tabName2.column" ] } } }
```

\$collections

This required Informix® JSON operator defines the two or more collections or relational tables that are included in the join.

\$project

This required MongoDB JSON operator applies a projection clause to the *table_or_collection_name* that is specified.

\$where

This optional MongoDB JSON operator applies a query filter to the table or relational table. You can use any of the supported query operators that are listed here: [Query and projection operators on page 95](#).

\$condition

This required Informix® JSON operator defines how the specified collections or tables are joined. You can specify a condition by mapping a single table column to another single table column, or a single table column to multiple other table columns.

2. Run a find query against a pseudo table that is named **system.join** with the join query document specified.

Example

For example, in the MongoDB shell:

```
> db.system.join.find({join_query_document})
```

Results

The query results are returned.

Example

Examples of join query document syntax

This example retrieves customer orders that total more than \$100. The join query document joins the **customer** and **orders** tables, on the **customer_num** field where the order total is greater than 100. The same query document works if the customers and orders tables are collections, relational tables, or a combination of the two.

```
{ "$collections":
  {
    "customers":
      { "$project": { customer_num: 1, name: 1, phone: 1 } },
    "orders":
      { "$project": { order_num: 1, nitems: 1, total: 1, _id: 0 },
        "$where": { total: { "$gt": 100 } } }
  },
  "$condition":
    { "customers.customer_num": "orders.customer_num" }
}
```

This example retrieves the order, shipping, and payment information for order number 1093. The array syntax is used in the **\$condition** syntax of the join query document.

```
{ "$collections":
  {
    "orders":
      { "$project": { order_num: 1, nitems: 1, total: 1, _id: 0 },
        "$where": { order_num: 1093 } },
  }
}
```

```

        "shipments":
            {"$project":{shipment_date:1,arrival_date:1}},
        "payments":
            {"$project":{payment_method:1,payment_date:1}}
    },
    "$condition":
        {"orders.order_num":["shipments.order_num","payments.order_num"]}
}

```

This example retrieves the order and customer information for orders that total more than \$1000 and that are shipped to the postal code 10112.

```

{"$collections":
  {
    "orders":
      {"$project":{order_num:1,nitems:1,total:1,_id:0},
       "$where":{total:{$gt:1000}}},
    "shipments":
      {"$project":{shipment_date:1,arrival_date:1,_id:0},
       "$where":{address.zipcode:10112}},
    "customer":
      {"$project":{customer_num:1,name:1,company:1,_id:0}}
  },
  "$condition":
    {
      "orders.order_num":"shipments.order_num",
      "orders.customer_num":"customer.customer_num",
    }
}

```

Operators

The MongoDB operators that are supported by Informix® are sorted into logical areas.

MongoDB read and write operations on existing relational tables are run as if the table were a collection. The wire listener determines whether the accessed entity is a relational table and converts the basic MongoDB operations on that table to SQL, and then converts the returned values back into a JSON document. The initial access to an entity results in an extra call to the Informix® server. However, the wire listener caches the name and type of an entity so that subsequent operations do not require an extra call.

MongoDB operators are supported on both JSON collections and relational tables, unless explicitly stated otherwise.

Query and projection operators

Informix® supports a subset of the MongoDB query and projection operators.

You can refine your queries with the MongoDB query and projection operators. For example, in the **mongo** shell, to find members of the **cartype** collection with an age greater than 10, you can use the **\$gt** operator: `db.cartype.find({"age":{"$gt":10.0}})`.

The JSON wire listener supports the skip, limit, and sort query options. You can set these options by using the **mongo** shell or MongoDB drivers.

- [Query selectors on page 96](#)
- [Projection operators on page 98](#)

Query selectors

Use query selectors to select specific data from queries.

Array query operators

Table 16. Array query operators

MongoDB command	JSON collections	Relational tables	Details
\$elemMatch	Yes	No	
\$size	Yes	No	

Comparison query operators

Table 17. Comparison query operators

MongoDB command	JSON collections	Relational tables	Details
\$all	Yes	Yes	Supported for primitive values and simple queries only. The operator is only supported when it is the only condition in the query document.
\$eq	Yes	Yes	
\$gt	Yes	Yes	
\$gte	Yes	Yes	
\$in	Yes	Yes	
\$lt	Yes	Yes	
\$lte	Yes	Yes	
\$ne	Yes	Yes	
\$nin	Yes	Yes	
\$query	Yes	Yes	

Element query operators

Table 18. Element query operators

MongoDB command	JSON collections	Relational tables	Details
\$exists	Yes	No	
\$type	Yes	No	

Evaluation

Table 19. Evaluation query operators

MongoDB command	JSON collections	Relational tables	Details
\$mod	Yes	Yes	
\$regex	Yes	Yes	The only supported value for the \$options flag is i, which specifies a case-insensitive search.
\$text	Yes	Yes	<p>The \$text query operator support is based on MongoDB version 2.6.</p> <p>You can customize your text index and take advantage of additional text query options by creating a basic text search index with the createTextIndex command. For more information, see Informix JSON commands on page 82.</p>
\$where	No	No	

Geospatial query operators

Geospatial queries are supported by using the GeoJSON format. The legacy coordinate pairs are not supported.

Table 20. Geospatial query operators

MongoDB command	JSON collections	Relational tables	Details
\$geoWithin	Yes	No	
\$geoIntersects	Yes	No	
\$near	Yes	No	
\$nearSphere	Yes	No	

JavaScript™ query operators

The JavaScript™ query operators are not supported.

Logical query operators

Table 21. Logical query operators

MongoDB command	JSON collections	Relational tables	Details
\$and	Yes	Yes	
\$or	Yes	Yes	
\$not	Yes	Yes	
\$nor	Yes	Yes	

Projection operators

Use projection operators to select specific data from a document.

Projection operators

Table 22. Projection operators

MongoDB command	JSON collections	Relational tables	Details
\$	No	No	
\$elemMatch	Yes	No	
\$meta	Yes	Yes	
\$slice	No	No	

Query modifiers

Table 23. Query modifiers

MongoDB command	JSON collections	Relational tables	Details
\$comment	No	No	
\$explain	Yes	Yes	
\$hint	Yes	No	
\$orderby	Yes	Yes	

For more information about the MongoDB features, see <http://docs.mongodb.org/manual/reference/>.

Update operators

Informix® supports a subset the MongoDB update operators.

You can use update operators to modify or add data in your database. For example, in the **mongo** shell, to change the username to atlas in the document with the `_id` of 101 in the users collection, you can use the `$set` operator:

```
db.users.update({ "_id":101 }, { "$set": { "username": "atlas" } }).
```

Array update operators

Table 24. Array update operators

MongoDB command	JSON collections	Relational tables	Details
\$	No	No	
\$addToSet	Yes	No	Supported for primitive values only. The operator is not supported on arrays and objects.
\$pop	Yes	No	
\$pullAll	Yes	No	Supported for primitive values only. The operator is not supported on arrays and objects.
\$pull	Yes	No	Supported for primitive values only. The operator is not supported on arrays and objects.
\$pushAll	Yes	No	
\$push	Yes	No	

Array update operators modifiers

Table 25. Array update modifiers

MongoDB command	JSON collections	Relational tables	Details
\$each	Yes	No	
\$slice	Yes	No	
\$sort	Yes	No	
\$position	Yes	No	

Bitwise update operators

Table 26. Bitwise update operators

MongoDB command	JSON collections	Relational tables	Details
\$bit	Yes	No	

Field update operators

Table 27. Field update operators

MongoDB command	JSON collections	Relational tables	Details
\$currentDate	Yes	Yes	
\$inc	Yes	Yes	
\$max	Yes	Yes	
\$min	Yes	Yes	
\$mul	Yes	Yes	
\$rename	Yes	No	
\$setOnInsert	Yes	No	
\$set	Yes	Yes	
\$unset	Yes	Yes	

Isolation update operators

The isolation update operators are not supported.

For more information about the MongoDB features, see <http://docs.mongodb.org/manual/reference/>.

Informix® query operators and modifier

The Informix® query operators and modifier are extensions to the MongoDB API.

Query operators

You can use the Informix® query operators in all MongoDB functions that accept query operators, for example `find()` or `findOne()`.

\$ifxtext

The \$ifxtext query operator is similar to the MongoDB \$text operator, except that it passes the search string as-is to the `bts_contains()` function.

When using relational tables, the MongoDB \$text and Informix® \$ifxtext query operators both require a column name, specified by \$key, in addition to the \$search string.

The search string can be a word or a phrase as well as optional query term modifiers, operators, and stopwords. You can include field names to search in specific fields. The syntax of the search string in the \$ifxtext query operator is the same as the syntax of the search criteria in the **bts_contains()** function that you include in an SQL query.

In the following example, a single-character wildcard search is run for the strings `text` or `test`:

```
db.collection.find( { "$ifxtext" : { "$search" : "te?t" } } )
```

\$like

The \$like query operator tests for matching character strings and maps to the SQL LIKE query operator. For more information about the SQL LIKE query operator, see [LIKE Operator on page 102](#).

In the following example, a wildcard search is run for strings that contain `machine`:

```
db.collection.find( { "$like" : "%machine%" } )
```

Query modifier

You can use the Informix® query modifier in the MongoDB find() function with the addSpecial function.

\$nativeCursor

The \$nativeCursor query modifier holds open a true cursor on the Informix® database server during a query. A native cursor requires more wire listener resources because connections and result set objects are tied to a single session, but the cursor guarantees consistent query results. You can control the cursor idle timeout with the cursor.idle.timeout wire listener property. For REST API queries, use the killCursors command to close the cursor.

In the following example, query results are returned in a cursor:



Note: For Mongo client applications, whether you are able to add Informix specific query or cursor modifiers is highly dependent on the Mongo driver you are using. Many of the latest Mongo driver do not support non-standard options in the addSpecial API.

```
db.collection.find().addSpecial("$nativeCursor",1);
```

Aggregation framework operators

The MongoDB aggregation framework operators that are supported by Informix® are sorted into logical areas.

You can use aggregation framework operators to aggregate and manipulate documents as they move through the aggregation pipeline stages. You can use some operators to aggregate or slice time series data.

- [Pipeline operators on page 102](#)
- [Expression operators on page 103](#)

Pipeline operators

Table 28. Pipeline operators

MongoDB command	JSON collections	Relational tables	Details
\$geoNear	Yes	No	<ul style="list-style-type: none">Supported by using the GeoJSON format. The MongoDB legacy coordinate pairs are not supported.You cannot use dot notation for the distanceField and includeLocs parameters.
\$group	Yes	Yes	For the syntax to aggregate time series data, see Aggregate or slice time series data on page 141 .
\$limit	Yes	Yes	
\$match	Yes	Yes	
\$out	Yes	Yes	
\$project	Partial	Partial	<ul style="list-style-type: none">You can use \$project to include fields from the original document, for example <code>{ \$project : { title : 1 , author : 1 } }</code>.You cannot use \$project to insert computed fields, rename fields, or create and populate fields that hold subdocuments.Projection operators are not supported.You can use the \$slice operator to return part of a time series. For the syntax to slice time series data, see Aggregate or slice time series data on page 141.
\$redact	No	No	
\$skip	Yes	Yes	
\$sort	Yes	Yes	
\$unwind	Yes	No	

Expression operators

\$group operators

Table 29. \$group operators

Command	JSON collections	Relational tables	Time series tables	Details
\$addToSet	Yes	No	No	
\$avg	Yes	Yes	Yes	
\$first	Yes	Yes	Yes	
\$last	Yes	Yes	Yes	
\$max	Yes	Yes	Yes	
\$median	No	No	Yes	An Informix® JSON operator for aggregating time series data. For the syntax to aggregate time series data, see Aggregate or slice time series data on page 141 .
\$min	Yes	Yes	Yes	
\$nth	No	No	Yes	An Informix® JSON operator for aggregating time series data. For the syntax to aggregate time series data, see Aggregate or slice time series data on page 141 .
\$push	Yes	No	No	
\$sum	Yes	Yes	Yes	

For more information about the MongoDB features, see <http://docs.mongodb.org/manual/reference/>.

Informix® REST API

The Informix® REST API allows application developers to access data in their Informix® database server through simple HTTP requests. Applications can query, insert, load, update, and delete data in relational tables, in JSON collections, or in TimeSeries tables all through simple, JSON-based REST requests.

The topics in this guide cover the Informix® REST API, including how to start it, how to secure it, and the syntax for accessing your data through REST. This information is intended for application developers.

About the Informix® REST API

The REST API provides driverless access to your Informix® data.

With the Informix® REST API, you can run query, insert, load, update, and delete operations against relational tables, JSON collections, and TimeSeries tables. The REST API provides application developers driverless access to their Informix® data from any application through simple HTTP requests.

The REST API is a Java™ application based on the [Eclipse Vert.x](#) framework. It is provided as an executable JAR file in the Informix® APIs package. When started, the REST jar will start a mid-tier gateway server that will process HTTP requests from clients. The REST server converts these HTTP requests into database operations (queries, inserts, deletes, etc.) which it then executes against the Informix® database server through a JDBC connection.

The REST API transfers data in JSON format, no matter if the data in the Informix® database resides in traditional relational tables, JSON collections, or TimeSeries tables. The JSON document format provides a way to transfer data in a way that is language neutral and easy to consume in whichever programming language is used by the application.

Requirements for the REST API

Informix® REST API has specific software dependencies and requirements.

Java requirements

To run the Informix® REST API, you must use a supported Java™ runtime environment. Java version 11 or higher is required.

Getting started with REST

These topics provide information on how to get started with the Informix® REST API, including configuration, authentication, and security options.

Starting the REST API

This topic provides a brief tutorial to help you get started with the Informix® REST API.

To start the Informix® REST API:

1. Create a [REST configuration on page 105](#) file. You can refer to the example properties file in the Informix® APIs package.

The REST configuration file must contain a **db.servers** property which defines one or more Informix® for the REST API to connect to. All other configuration properties are optional.

Sample configuration file

```
# REST port number
port: 8080

# List of database servers that the REST API can connect to db.servers:
-
  alias: server1
  host: host1.mycompany.com
```



```

    port: 9088
  -
    alias: server2
    host: host2.mycompany.com
    port: 9088

# Optional, but recommended, properties to enable HTTPS
tls.enable: true
tls.keystore.type: pkcs12
tls.keystore.file: rest-keystore.pkcs12
tls.keystore.password: myPassword

# Optional, but recommended, property to enable anti-CSRF tokens
security.csrf.token.enable: true

```

! **Important:** The **db.servers** property is required. This specifies the alias, host, and port number for the databases servers that HTTP clients will be able to access through the REST API.

! **Important:** In addition to **db.servers**, it is recommended that you configure the TLS (HTTPS) and anti-CSRF configuration properties in order to secure your REST API when running it in a production environment. See the [Securing the REST API on page 112](#) topic for more information on how to properly secure your REST API server.

2. Start the REST server using the following Java™ command:

```
java -jar db-rest.jar -config rest-config.yaml
```

where **rest-config.yaml** is the name of your REST configuration file.

! **Important:** The REST server requires Java 11 or higher.

3. By default, the REST server will write its logging information to a file named `db-rest.log`.

Related information

[The REST configuration file on page 105](#)

The REST configuration file

The Informix® REST API is configured through a yaml file. This configuration file contains, among other things, settings for the connection between the REST API and the database server.

A sample configuration file is provided in the `example-rest-config.yaml` file. In the sample configuration file, all of the properties are commented out by default. To enable a property, you must uncomment the row and customize the property.

To modify the configuration after the REST API is started, you must first stop the REST server, update the configuration file, and then restart the REST API for the changes to take affect.

REST configuration properties



Important: The **db.servers** property is required. All other properties are optional.

- REST server configuration
 - [hostname on page 107](#)
 - [port on page 107](#)
- Informix servers
 - [db.servers on page 107](#)
- Security settings
 - [security.csrf.token.enable on page 107](#)
 - [tls.enable on page 108](#)
 - [tls.keystore.type on page 108](#)
 - [tls.keystore.file on page 108](#)
 - [tls.keystore.password on page 108](#)
 - [tls.pem.cert on page 108](#)
 - [tls.ciphers on page 108](#)
 - [tls.protocols on page 108](#)
- Timeouts
 - [rest.session.timeout on page 109](#)
 - [metadata.cache.timeout on page 109](#)
- Query defaults
 - [query.default.limit on page 109](#)
 - [query.maximumFieldSize on page 109](#)
 - [query.timeseries.default.limit on page 109](#)
- JSON formatting
 - [json.format on page 109](#)
- JDBC connection pool settings
 - [dataSource.IFX_ISOLATION_LEVEL on page 110](#)
 - [pool.connectionTimeout on page 110](#)
 - [pool.idleTimeout on page 110](#)
 - [pool.maximumPoolSize on page 110](#)
 - [pool.minimumIdle on page 110](#)
- TimeSeries loader settings
 - [timeseries.loader.flush.interval on page 110](#)
 - [timeseries.loader.log.level on page 111](#)
 - [timeseries.loader.put.size.maximum on page 111](#)
 - [timeseries.loader.session.timeout on page 111](#)
- Vert.x configuration options
 - [vertex.threads.poolSize on page 111](#)

REST server configuration

hostname

This property specifies the hostname for the REST server. The default value of "0.0.0.0" means that the REST API server will listen on all available interfaces.

```
hostname : { 0.0.0.0 | hostname }
```

port

This property specifies the port for the REST server. The default port is 8080.

```
port : { 8080 | port }
```

Informix servers

db.servers

This required property specifies the list of Informix database servers that the REST API will provide access to. Each database server in the list must have an **alias**, a **hostname**, and a **port**, and can optionally have a set of additional JDBC **properties**. The alias is specific to the REST API and provides the name you will use in the REST API urls to refer to that particular database server.

```
db.servers:
-
  alias: server1
  host: host1.mycompany.com
  port: 9088
-
  alias: server2
  host: host2.mycompany.com
  port: 9088
  properties:
    IFX_LOCK_MODE_WAIT: "20"
    IFX_USEPUT: "1"
```

Security settings

security.csrf.token.enable

This property enables tokens in the REST API to prevent Cross-Site Request Forgery (CSRF) attacks.

When these anti-CSRF tokens are enabled, an X-CSRF-TOKEN will be returned in the header of the first HTTP response for an authenticated session. Any subsequent HTTP POST, PUT, or DELETE request using the same session cookie must include the X-CSRF-TOKEN in the header of the HTTP request. If the X-CSRF-TOKEN is not included in the header or an invalid token is sent, the REST API will respond with a FORBIDDEN response.

It is recommended that you enable these tokens in a production deployment of the REST API. When enabled, REST clients must the X-CSRF-TOKEN in the request header whenever they are using a session cookie to re-use an existing REST session.

```
security.csrf.token.enable : { false | true }
```

tls.enable

This property controls whether TLS is enabled. When enabled, the REST server will be started as an HTTPS server.

```
tls.enable : { false | true }
```

tls.keystore.type

This property specifies the keystore type for TLS encryption. Supported values are `jks` for a Java keystore, `pfx` or `pks12` for a PKCS12 keystore, or `pem` to use PEM files.

```
tls.keystore.type : { jks | pfx | pkcs12 | pem }
```

tls.keystore.file

This property specifies the path to the keystore file.

```
tls.keystore.file : /path/to/keystore
```

tls.keystore.password

This property specifies the password for the keystore file.

```
tls.keystore.password : password
```

tls.pem.cert

This property specifies the path to the PEM certificate, if **tls.keystore.type** has been specified as `pem`.

```
tls.keystore.password : password
```

tls.ciphers

This property specifies the list of enabled TLS ciphers. If no value is specified, the JVM's default list of enabled ciphers will be used.

If you are unfamiliar with the Java Cryptography Extension (JCE) and the security implications of the multitude of supported ciphers, you should not specify this property. Consult with a security expert for an up-to-date recommendation based on your use case.

tls.protocols

This property specifies the list of enabled TLS protocols.

```
tls.protocols:  
- TLSv1.2
```

Timeouts

rest.session.timeout

This property specifies the number of milliseconds a REST session can be idle before it is closed. The default session timeout is 10 minutes.

```
rest.session.timeout : { 600000 | timeout }
```

metadata.cache.timeout

This property specifies the number of milliseconds that metadata about Informix databases and tables can remain in the REST API server's cache before they are refreshed. If set to zero, the objects in the cache remain there indefinitely and are only refreshed on an as needed basis.

```
metadata.cache.timeout : { 0 | timeout }
```

Query defaults

query.default.limit

This property specifies the default limit for the number of rows returned in a single HTTP query response. Setting this value to -1 indicates that there should be no default limit on the number of rows returned by a query response. The default limit is 100 rows.

```
query.default.limit : { 100 | limit }
```

query.maximumFieldSize

This property sets a maximum field size for values in relational query results. If a value exceeds the maximum size, it will be truncated. Setting this value to -1 indicates that there should be no limit on the maximum field size.

```
query.maximumFieldSize : { -1 | maxSize }
```

query.timeseries.default.limit

This property specifies the default limit for the number of TimeSeries entries returned in a single HTTP query response. Setting this value to -1 indicates that there should be no default limit on the number of entries in TimeSeries values within a query response. The default limit is 100 entries.

```
query.timeseries.default.limit : { 100 | limit }
```

JSON formatting

json.format

This property specifies the format used for JSON serialization. Allowable values are `RELAXED` or `EXTENDED`. The `EXTENDED` format is compatible with MongoDB's JSON serialization; whereas the `RELAXED` format is slightly more compact for some data types.

```
json.format : { RELAXED | EXTENDED }
```

JDBC connection pool settings

dataSource.IFX_ISOLATION_LEVEL

This property specifies the isolation level for JDBC connections. The default isolation level is 1 (DIRTY READ).

```
dataSource.IFX_ISOLATION_LEVEL : { 1 | isolationLevel }
```

pool.connectionTimeout

This property specifies the number of milliseconds to wait for a JDBC connection to a Informix database server to be established before it times out.

```
pool.connectionTimeout : { 5000 | timeout }
```

pool.idleTimeout

This property specifies the number of milliseconds that a JDBC connection can be idle in the connection pool before it is closed. The default value is 60000 milliseconds (1 minute).

```
pool.idleTimeout : { 60000 | timeout }
```

pool.maximumPoolSize

This property specifies the maximum number of JDBC connections in each connection pool. A connection pool will be established for each unique Informix server, database, and username that is connected to it through the REST API. The **pool.maximumPoolSize** property puts a cap on the total number of open JDBC connections that can be established in each one of these connection pools.

```
pool.maximumPoolSize : { 50 | size }
```

pool.minimumIdle

This property specifies the minimum number of idle JDBC connections in each connection pool. A connection pool will be established for each unique Informix server, database, and username that is connected to it through the REST API. Setting **pool.minimumIdle** to zero indicates that all JDBC connections in the connection pool should be closed when they exceed the **pool.idleTimeout**. Setting **pool.minimumIdle** to a positive integer indicates the number of connections that should be kept open in the connection pool even when they exceed the **pool.idleTimeout**. The default and recommended value is 0.

```
pool.minimumIdle : { 0 | size }
```

TimeSeries Loader settings

timeseries.loader.flush.interval

This property specifies the flush interval, in milliseconds, for the TimeSeries loader.

```
timeseries.loader.flush.interval : { 500 | interval }
```

timeseries.loader.log.level

This property sets the log level used for the TimeSeries Loader stored procedures on the database. This log level effects the level of TimeSeries Loader log messages written to the database server's online.log. Possible values are `info`, `warning`, `error`, and `debug`.

```
timeseries.loader.log.level : { warning | info | error | debug }
```

timeseries.loader.put.size.maximum

This property specifies the maximum buffer size, in bytes, for each TimeSeries Loader put call to the database server.

```
timeseries.loader.put.size.maximum : { 3840 | size }
```

timeseries.loader.session.timeout

This property specifies the number of milliseconds that a TimeSeries loader session can be idle before it is closed. The default value is 300000 milliseconds (5 minutes).

```
timeseries.loader.session.timeout : { 300000 | timeout }
```

Vert.x configuration options

vertx.threads.poolSize

This property specifies the maximum number of worker threads in the thread pool used to process HTTP requests.

```
vertx.threads.poolSize : { 128 | size }
```

Authentication with REST

HTTP clients authenticate to the Informix® REST API through HTTP Basic Authentication using a user and password known to the Informix® database server.

Authentication in the REST API is ultimately done directly by the Informix® database server. HTTP clients authenticate to REST using HTTP Basic Authentication. The REST API server will then establish a JDBC connection to the database server using the username and password sent by the HTTP client. This model ensures that the database server controls all user accounts and privileges and you can audit user activities and configure fine-grained access control directly on the database server.

After the first HTTP request authenticating a user, the REST API will return a session cookie in the HTTP response using the `Set-Cookie` header. The session cookie name is `db-rest.session`. To re-use the same REST session, all subsequent HTTP requests should include the `db-rest.session` cookie as part of the HTTP request. HTTP clients can achieve better performance on their requests if they re-use their established REST session as opposed to re-authenticating with each

subsequent HTTP request. REST sessions timeout by default after 10 minutes of inactivity, but this duration can be configured with the [rest.session.timeout on page 109](#) property.

Securing the REST API

This topic covers important options for securing your Informix® REST API.

Securing your REST API server with TLS (HTTPS)

You can enable Transport Layer Security (TLS) in the REST API server to encrypt all communications between the REST server and clients. By enabling TLS, the REST server will run as an HTTPS server.



Important: It is recommended that you enable TLS and HTTPS whenever deploying the REST API server in a production environment.

The REST server supports three types of keystores to hold your encryption keys: JKS, PKCS12, and PEM files.

Java KeyStore (JKS)

You can use the Java `keytool` utility that comes with your JDK installation to generate a Java KeyStore. Refer to your Java documentation for instructions for creating a keystore.

After you have created a keystore, include the following properties in your [REST configuration file on page 107](#).

```
tls.enable: true
tls.keystore.type: jks
tls.keystore.file: mykeystore.jks
tls.keystore.password: mypassword
```

PKCS12

You can use a PKCS12 keystore to encrypt communications with your REST API server by setting the following properties in your [REST configuration file on page 107](#).

```
tls.enable: true
tls.keystore.type: pkcs12
tls.keystore.file: mykeystore.pkcs12
tls.keystore.password: mypassword
```

PEM files

You can use OpenSSL to generate a .PEM certificate and key files. Refer to your OpenSSL documentation for instructions for generating a certificate and keys.

To use a .PEM certificate and key files, include the following properties in your [REST configuration file on page 107](#).

```
tls.enable: true
tls.keystore.type: pem
tls.keystore.file: rest_key.pem
tls.pem.cert: rest_cert.pem
```


Securing the REST server's JDBC connections with SSL

You can encrypt the JDBC connections between the REST server and the Informix® database server with the Secure Sockets Layer (SSL) protocol.

First, you must have SSL configured for the database server. See [Configuring a server instance for secure sockets layer connections](#).

Then you will need to add three JDBC properties – SSLCONNECTION, TRUSTSTORE, and SSL_TRUSTSTORE_PASSWORD – to your server connection information in your [REST configuration file on page 107](#).

```
db.servers:
-
  alias: server1
  host: host1.mycompany.com
  port: 9088
  properties:
    SSLCONNECTION: true
    SSL_TRUSTSTORE: client_keystore.jks
    SSL_TRUSTSTORE_PASSWORD: myKeystorePass
```



Important: If you have more than one server in your [db.servers on page 107](#) list, you must add these SSL-related JDBC connection properties to each server in the list that you want to secure with SSL.

Enabling Anti Cross-Site Request Forgery tokens

The REST API server supports anti-CSRF tokens to secure it against Cross-Site Request Forgery (CSRF) attacks.

When these anti-CSRF tokens are enabled, an `X-CSRF-TOKEN` will be returned in the header of the first HTTP response for an authenticated session. Any subsequent HTTP POST, PUT, or DELETE request using the same session cookie must include the `X-CSRF-TOKEN` in the header of the HTTP request. If the `X-CSRF-TOKEN` is not included in the header or an invalid token is sent, the REST API will respond with a FORBIDDEN response.



Important: It is recommended that you enable these tokens in a production deployment of the REST API server. When enabled, REST clients must send the X-CSRF-TOKEN in the request header whenever they are using a session cookie to re-use an existing REST session.

Logging in the REST API server

This topic discusses logging options in the Informix® REST API server.

The default logging mechanism for the REST API is Log4j. Log4j uses a configuration file to customize the level, style and target location for log messages. You can customize the logging output to fit your needs or to provide diagnostics for a technical support representative.

Below is a generic example log4j2.xml file which allows you control logging for the REST API using the Java system properties specified in the table.

-Dlog4j2.configurationFile	Specify the location of the XML configuration file
-Dapp.logtarget	Specify 'console' or 'file'
-Dapp.logfile	If logtarget is 'file' give a file path here
-Dapp.loglevel	Specify the log level of the application
-Djdbc.loglevel	Specify the log level of the underlying JDBC driver

```

<?xml version="1.0" encoding="UTF-8"?>
<!--
  Generic Log4j2 configuration file that gives custom
  levels for JDBC and the application itself
-->
<Configuration monitorInterval="30" status="WARN">
  <Properties>
    <Property name="lgtarget">${sys:app.logtarget:-console}</Property>
    <Property name="lgfile">${sys:app.logfile:-app.log}</Property>
    <Property name="app">${sys:app.loglevel:-info}</Property>
    <Property name="jdbc">${sys:jdbc.loglevel:-error}</Property>
    <Property name="pattern">%d{yyyy-MM-dd HH:mm:ss.SSS} | %-5level | %t | %c{1} | %method | %marker
  | %msg%n</Property>
  </Properties>
  <Appenders>
    <Routing name="Router">
      <Routes pattern="{lgtarget}">
        <Route ref="Console" key="console" />
        <Route ref="File" key="file" />
      </Routes>
    </Routing>
    <Console name="Console">
      <PatternLayout pattern="{pattern}" />
    </Console>
    <File name="File" fileName="{lgfile}">
      <PatternLayout pattern="{pattern}" />
    </File>
  </Appenders>
  <Loggers>
    <!-- JDBC Driver packages -->
    <Logger name="com.onedb" level="{jdbc}" />
    <Logger name="com.onedb.jdbc" level="{jdbc}" />
    <Logger name="com.onedb.jdbcx" level="{jdbc}" />
    <!-- Disable arcs/hikari logging except for errors -->
    <Logger name="com.informix.arcs" level="error" />
    <Logger name="com.zaxxer.hikari" level="warn" />

    <Root level="{app}">
      <AppenderRef ref="Router"/>
    </Root>
  </Loggers>
</Configuration>

```

REST API syntax

A subset of the HTTP methods is supported by the REST API. These methods are DELETE, GET, POST, and PUT.

- [POST on page 115](#)
- [PUT on page 117](#)
- [GET on page 118](#)
- [DELETE on page 121](#)

The examples that are shown in this topic contain line breaks for page formatting; however, the REST API does not allow line breaks.

POST

The POST method maps to the MongoDB insert or create command.

Table 30. Supported POST method syntax

Method	Path	Description
POST	/	Create a database.
POST	/databaseName	Create a collection. databaseName The database name.
POST	/databaseName/collectionName	Create a document. databaseName The database name. collectionName The collection name.

Create a database

This example creates a database with the locale specified.

Request:

Specify the POST method:

```
POST /
```

Data:

Specify database name mydb and an English UTF-8 locale:

```
{name:"mydb", locale:"en_us.utf8"}
```

Response:

The following response indicates that the operation was successful:

```
{"msg":"created db 'mydb'","ok":true}
```

Create a collection

This example creates a collection in the mydb database.

Request:

Specify the POST method and the database name as mydb:

```
POST /mydb
```

Data:

Specify the collection name as bar:

```
{name:"bar"}
```

Response:

The following response indicates that the operation was successful:

```
{"msg":"created collection mydb.bar","ok":true}
```

Create a relational table

This example creates a relational table in an existing database.

Request:

Specify the POST method and stores_mydb as the database:

```
POST /stores_mydb
```

Data:

Specify the table attributes:

```
{ name: "rel",
  options: {
    columns:[{name:"id",type:"int",primaryKey:true},
              {name:"name",type:"varchar(255)"},
              {name:"age",type:"int",notNull:false}]
  }
}
```

Response:

The following response indicates that the operation was successful:

```
{msg: "created collection stores_mydb.rel" ok: true}
```

Insert a single document

This example inserts a document into an existing collection.

Request:

Specify the POST method, mydb database, and people collection:

```
POST /mydb/people
```

Data:

Specify John Doe age 31:

```
{firstName:"John",lastName:"Doe",age:31}
```

Response:

Here is a successful response:

```
{"n":1,"ok":true}
```

Insert multiple documents into a collection

This example inserts multiple documents into a collection.

Request:

Specify the POST method, mydb database, and people collection:

```
POST /mydb/people
```

Data:

Specify John Doe age 31 and Jane Doe age 31:

```
[{"firstName":"John","lastName":"Doe",age:31}, {"firstName":"Jane","lastName":"Doe",age:31}]
```

Response:

Here is a successful response:

```
{"n":2,"ok":true}
```

PUT

The PUT method maps to the MongoDB update command.

Table 31. Supported PUT method syntax

Method	Path	Description
PUT	<code>/databaseName/collectionName?queryParameters</code>	Update a document. databaseName The database name. collectionName The collection name. queryParameters The supported Informix® <i>queryParameters</i> are query and options . The parameter named query maps to the equivalent MongoDB query. The options parameter can contain {"upsert":true/false} or {"multiUpdate":true/false}.

Update a document in a collection

This example updates the value for Larry in an existing collection, from age 49 - 25:

```
[{"_id":{"$oid":"536d20f1559a60e677d7ed1b"},"firstName":"Larry",
"lastName":"Doe","age":49}, {"_id":{"$oid":"536d20f1559a60e677d7ed1c"},
"firstName":"Bob","lastName":"Doe","age":47}]
```

Request:

Specify the PUT method and query the name Larry:

```
PUT /mydb/people?query={firstName:"Larry"}
```

Data:

Specify the MongoDB \$set operator with age 25:

```
{"$set":{"age":25}}
```

Response:

Here is a successful response:

```
{"n":1,"ok":true}
```


GET

The GET method maps to the MongoDB query command.

Table 32. Supported GET method syntax

Method	Path	Description
GET	/	List databases
GET	/databaseName	List collections
	databaseName	The database name.
GET	/databaseName/collectionName?queryParameters	Query the collection.
	databaseName	The database name.
	collectionName	The collection name.
	queryParameters	The query parameters.
		The supported Informix® <i>queryParameters</i> are batchSize, query, fields, and sort. These map to the equivalent MongoDB batchSize, query, fields, and sort parameters.
GET	/databaseName/system.sql?query={"\$sql?::?sql_statement?"}	Run SQL passthrough query.

Table 32. Supported GET method syntax (continued)

Method	Path	Description
		 Note: You must enable SQL operations by setting security.sql.passthrough=true in the wire listener properties file.
		databaseName The database name.
		sql_statement Any SQL query or statement.
GET	<code>/databaseName/\$cmd?query={command_document}</code>	Run the Informix® or MongoDB JSON command. databaseName The database name. command_document The Informix® or MongoDB JSON command document. Specify the command document in the same format that is used by the db.runCommand() in the mongo shell.

List databases

This example lists all of the databases on the server.

Request:

Specify the GET method and forward slash (/):

```
GET /
```

Data:

None.

Response:

Here is a successful response:

```
[ "mydb" , "test" ]
```

List all collections

This example lists all of the collections in a database.

Request:

Specify the GET method and mydb database:

```
GET /mydb
```

Data:

None.

Response:

Here is a successful response:

```
["bar"]
```

Query a collection and sort the results in ascending order

This example sorts the query results in ascending order by age.

Request:

Specify the GET method, mydb database, people collection, and query with the sort parameter. The sort parameter specifies ascending order (age:1), and filters id (_id:0) and last name (lastName:0) from the response:

```
GET /mydb/people?sort={age:1}&fields={_id:0,lastName:0}
```

Data:

None.

Response:

The first names are displayed in ascending order with the _id and lastName filtered from the response:

```
[{"firstName":"Sherry","age":31},
{"firstName":"John","age":31},
{"firstName":"Bob","age":47},
{"firstName":"Larry","age":49}]
```

Run the collStats command to get statistics about a collection

This example submits the MongoDB collStats command by using the REST API to get statistics about the jsonlog collection.

Here is the MongoDB shell syntax:

```
db.runCommand({collStats:"jsonlog"})
```

Request:

Specify the GET method, mydb database, and the collStats command document as the query:

```
GET /mydb/$cmd?query={collStats:"jsonlog"}
```

Data:

None.

Response:

```
[
  {
    "ns":"mydb.jsonlog",
    "count":1000,
    "size":322065,
```



```

    "avgObjSize":322,
    "storageSize":323584,
    "numExtents":158,
    "nindexes":1,
    "lastExtentSize":2048,
    "paddingFactor":0,
    "flags":1,
    "indexSizes":
    {
        "_id_":49152
    },
    "totalIndexSize":49152,
    "ok":1
  }
]

```

Run an SQL function

This example runs an SQL function that adds two values.

Here is the MongoDB shell syntax:

```
> db.runCommand({runFunction:"add_values", "arguments":[3,6]}
```

Request:

Specify the GET method, mydb database, the **runFunction** parameter with the function name, and the **arguments** parameter with the argument values as the query:

```
GET mydb/$cmd?query={"runFunction":"add_values","arguments":[3,6]}
```

Data:

None

Response:

```
[{"returnValue": 9,"ok":1.0}]
```

Query with a native cursor

Use the following format to return query results in a native cursor:

```
GET /dbname/collectionname?query={query_condition}&nativeCursor=1
```

After you obtain the results, run the killCursors command to close the cursor.

DELETE

The DELETE method maps to the MongoDB delete command.

Table 33. Supported DELETE method syntax

Method	Path	Description
DELETE	/	Delete all databases.
DELETE	/databaseName	Delete a database.

Table 33. Supported DELETE method syntax (continued)

Method	Path	Description
DELETE	<code>/databaseName/collectionName</code>	databaseName The database name.
		Delete a collection.
		databaseName The database name. collectionName The collection name.
DELETE	<code>/databaseName/collectionName?queryParameter</code>	Delete all documents that satisfy the query from a collection.
		databaseName The database name.
		collectionName The collection name. queryParameters The query parameters. The supported Informix® <i>queryParameter</i> is query. This map to the equivalent MongoDB query parameter.

Delete a database

This example deletes a database called mydb.

Request:

Specify the DELETE method and the mydb database:

```
DELETE /mydb
```

Data:

None.

Response:

Here is a successful response:

```
{msg: "dropped database", ns: "mydb", ok: true}
```

Delete a collection

This example deletes a collection from a database.

Request:

Specify the DELETE method, mydb database, and bar collection:

```
DELETE /mydb/bar
```

Data:

None.

Response:

Here is a successful response:

```
{"msg":"dropped collection", "ns":"mydb.bar", "ok":true}
```

Delete documents from a collection

This example deletes documents from a collection that contains the user "bob".

Request:

Specify the DELETE method, mydb database, people collection, and the query condition:

```
DELETE /mydb/people?query={user:"bob"}
```

Data:

None.

Response:

Here is a successful response where *n* indicates the number of documents deleted.

```
{"n":1,"ok":true}
```

Running SQL passthrough queries through REST

You can run any SQL statement and retrieve results back using SQL passthrough queries with the REST listener.

Before you begin

You must enable SQL operations by setting **security.sql.passthrough=true** in the wire listener properties file.

Use **GET** method and `system.sql` as the collection name and `$sql` as the query operator, followed by the SQL statement.

```
GET /databaseName/system.sql?query={"$sql": "sql_statement"}
```

To use host variables, include question marks in the SQL statement, and include the `$bindings` operator with an array that contains a value for each host variable in order of appearance.

```
GET /databaseName/system.sql?query={"$sql": "sql_statement", "$bindings": [values]}
```

Example**Examples****Create an table**

```
GET /mydb/system.sql?query={"$sql": "create table foo (c1 int)"}
```

Run an SQL query

```
GET /mydb/system.sql?query={"$sql": "select count(*) as count from foo"}
```

Sample response:

```
[ { "count": 10 } ]
```

Delete rows based on a host variable

```
GET /mydb/system.sql?query={"$sql": "delete from foo where c1 < ?", "$bindings" : [100] }}
```

Sample response:

```
[ { "n": 2 } ]
```

Troubleshooting the REST API server

Several troubleshooting techniques, tools, and resources are available for resolving problems that you encounter with the Informix® REST API.

Problem	Solution
How can I find version information for the REST API?	<p>You can find the version of your REST API jar file by running</p> <pre>java -jar rest.jar -version</pre> <p>For example:</p> <pre>> java -jar rest-3.0.0.0.jar -version HCL Informix REST Connector 3.0.0.0</pre>
How can I find the full build information for the REST API?	<p>You can find the build information for your REST API jar file by running</p> <pre>java -jar rest.jar -buildInformation</pre> <p>For example:</p> <pre>> java -jar rest-3.0.0.0.jar -buildInformation Product-Name: HCL Informix REST Connector Build-Version: 3.0.0.0 Build-Number: 12 Build-Date: 2024-10-10T10:09:26-0500 Commit: v3.0.0.0</pre>
How can I enable debug logging in the REST API server?	<p>You can customize the logging level of the REST API server by creating your own <code>logback.xml</code> file. This <code>logback.xml</code> file can either be in the current directory or in the classpath when you start the REST API server on page 104. For more information on logging options, see Logging in the REST API server on page 113.</p> <p>An example <code>logback.xml</code> file that turns on DEBUG logging to a file named <code>onedb-rest.log</code> is found below.</p> <pre><configuration scan="true" scanPeriod="5 seconds"> <!-- This appender section produces the typical rest.log file --> <!-- See https://logback.qos.ch/manual/layouts.html#ClassicPatternLayout for details --></pre>

Problem**Solution**

```

<appender name="FILE" class="ch.qos.logback.core.FileAppender">
  <append>true</append>
  <file>onedb-rest.log</file>
  <encoder>
    <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
    </pattern>
  </encoder>
</appender>

<!-- The base logging level is set here -->
<!-- You can choose from (TRACE, DEBUG, INFO, WARN, ERROR) -->
<root level="DEBUG">
  <appender-ref ref="FILE" />
</root>

<!-- You can configure custom logging levels (TRACE, DEBUG, INFO,
WARN,
ERROR) for any java package name -->
<!--<logger name="com.onedb.rest.api" level="DEBUG" />-->
<!--<logger name="com.onedb.rest.sql" level="DEBUG" />-->
<!--<logger name="com.informix.jdbc" level="INFO" />-->
<!--<logger name="com.zaxxer.hikari" level="INFO" />-->
</configuration>

```

Why is it that the first time I run a REST query (or any HTTP POST) it works, but the second time, I get an HTTP 403 FORBIDDEN response?

This will happen if you have [Anti Cross-Site Request Forgery tokens on page 107](#) enabled, but your REST HTTP POST, PUT, or DELETE requests do not include the `X-CSRF-TOKEN` header.

On your first HTTP request, you will be authenticated against the database server. A session will be created for you in the REST server and the first HTTP response you receive will include a `Set-Cookie` directive with your session identifier. The first HTTP response will also include an `X-CSRF-TOKEN` in the header. Any subsequent HTTP POST, PUT, or DELETE requests that include your session cookie – and many HTTP clients will do the work of adding it for you automatically – must also include your `X-CSRF-TOKEN` token in the header of the request. If you send an HTTP request with a valid session cookie but with a missing or invalid `X-CSRF-TOKEN`, the HTTP response will be 403 FORBIDDEN.

See [Enabling Anti Cross-Site Request Forgery tokens on page 113](#) for more information.

Manage time series through the wire listener

You can create and manage time series through the wire listener. You interact with time series data through a virtual table.

You can create, load, and query time series through the MongoDB API or the REST API. For example, you can program sensor devices that do not have client drivers to load time series data directly into the database with HTTP commands from the REST API. Because you act on a virtual table, the **TimeSeries** row type does not need to contain a BSON column.

You can load time series data through the MQTT protocol if your time series data is stored in a BSON column in the **TimeSeries** row type.

The following restrictions apply when you create a time series through the wire listener:

- You cannot define hertz or compressed time series.
- You cannot define rolling window containers.
- You cannot load time series data through a loader program. You must load time series data through a virtual table.
- You cannot run time series SQL routines or methods from the time series Java™ class library. You operate on the data through a virtual table.

Creating a time series through the wire listener

You can create time series with the REST API or the MongoDB API through the wire listener. You create time series objects by adding definitions to time series collections.

Before you begin

You must understand time series concepts, the properties of your data, and how much storage space your data requires. For an overview of time series concepts and guidance on how to design your time series solution, see [Informix® TimeSeries solution on page](#) .

Perform the following prerequisite tasks:

- Connect to a database in which to create the time series table. You run all methods in the database.
- Configure the wire listener for the MongoDB API or the REST API. For more information, see [Configuring the wire listener for the first time on page 8](#).
- Configure storage spaces for your time series data.

To create a time series through the wire listener:

1. Choose a predefined calendar from the `system.timeseries.calendar` collection or create a calendar by adding a document to the `system.timeseries.calendar` collection.
2. Create a **TimeSeries** row type by adding a document to the `system.timeseries.rowType` collection.
The row type must include one BSON column for the JSON data.
3. Create a container by adding a document to the `system.timeseries.container` collection.
4. Create a time series table with the time series table format syntax.
5. Instantiate the time series by creating a virtual table with the time series virtual table format syntax.
6. Load time series data.
You can use the REST API or the MongoDB API to load time series data through a virtual table. You can use the MQTT protocol to load time series data into the time series base table.

What to do next

After you create and load a time series, you query the data through the virtual table with MongoDB and REST clients.

Time series collections and table formats

You can add, view, and remove documents from the time series collections with REST API and MongoDB API methods to create and manage your time series. You must use a specific format to create time series tables and virtual tables that are based on time series tables.

For the REST API, use the GET, POST, and DELETE methods to view, insert, or delete data in the time series collections.

For the MongoDB API, use the query, create, or remove methods to view, insert, or delete data in the time series collections.

The time series collections are virtual collections that are used to manage the objects that are required to store time series data in a database.

- [system.timeseries.calendar collection on page 127](#)
- [system.timeseries.rowType collection on page 128](#)
- [system.timeseries.container collection on page 128](#)
- [Time series table format on page 129](#)
- [Virtual table format on page 130](#)

system.timeseries.calendar collection

The `system.timeseries.calendar` collection stores the definitions of predefined and user-defined calendars. A calendar controls the times at which time series data can be stored. The calendar definition embeds the calendar pattern definition. For details and restrictions about calendars, see [Calendar data type on page](#) . For a list of predefined calendars, see [Predefined calendars on page](#) .

Use the following format to add a calendar to the `system.timeseries.calendar` collection.

calendar

```
{ name : " calendar_name " , calendarStart : " start_date " , patternStart : " pattern_date " , pattern : { type : " interval " , intervals : [ { duration : " num_intervals " , on : { true | false } } ] } }
```

name

The name of the calendar.

calendarStart

The start date of the calendar.

patternStart

The start date of the calendar pattern.

pattern

The calendar pattern definition.

type

The time interval. Valid values for *interval* are: `second`, `minute`, `hour`, `day`, `week`, `month`, `year`.

intervals

The description of when to record data.

duration

The number of intervals, as a positive integer.

on

Whether to record data during the interval:

`true` = Recording is on.

`false` = Recording is off.

system.timeseries.rowType collection

The `system.timeseries.rowType` collection stores **TimeSeries** row type definitions. The **TimeSeries** row type defines the structure for the time series data within a single column in the database. For details and restrictions on **TimeSeries** row types, see [TimeSeries data type on page](#) .

Use the following format to add a **TimeSeries** row type to the `system.timeseries.rowType` collection.

```
{ name : " rowtype_name " , fields : [ { name : " field_name " , type : " data_type " } ] }
```

name

The `rowtype_name` is the name of the **TimeSeries** row type.

fields

name

The name of the field in the row data type. The `field_name` must be unique for the row data type.

The number of fields in a row type is not restricted.

type

Must be `datetime year to fraction(5)` for the first field, which contains the time stamp.

The data type of the field. Most data types are valid for fields after the time stamp field.

system.timeseries.container collection

The `system.timeseries.container` collection stores container definitions. Time series data is stored in containers. For details and restrictions on containers, see [TSContainerCreate procedure on page](#) . Rolling window container syntax is not supported.

Use the following format to add a container to the `system.timeseries.container` collection.

```
{ name : " container_name " , dbSpaceName : " dbspace_name " , rowTypeName : " rowtype_name " , firstExtent : extent_size , nextExtent : next_extent_size }
```


name

The *container_name* is the name of the container. The container name must be unique.

dbspaceName

The *dbspace_name* is the name of the dbspace for the container.

rowTypeName

The *rowtype_name* is the name of an existing **TimeSeries** row type in the `system.timeseries.rowType` collection.

firstExtent

The *extent_size* is a number that represents the first extent size for the container, in KB.

nextExtent

The *next_extent_size* is a number that represents the increments by which the container grows, in KB. The value must be equivalent to at least 4 pages.

Time series table format

A time series table must have a primary key column that does not allow null values. The last column in the time series table must be the **TimeSeries** column. For details and restrictions on time series tables, see [Create the database table on page](#).

The following format describes the simplest structure of a time series table. You can include other options and columns in a time series table.

```
{ collection : " table_name " , options : { columns : [ { name : " col_name " , type : " data_type " , primaryKey : true ,
notNull : true } , { name : " col_name " , type : " timeseries ( rowtype_name ) " } ] } }
```

collection

The *table_name* is the name of the time series table.

options

The collection definition.

columns

The column definitions.

name

The *col_name* is the name of the column.

type

The *data_type* is the data type of the column.

For the **TimeSeries** column, the *rowtype_name* is the name of an existing **TimeSeries** row type in the `system.timeseries.rowType` collection.

primaryKey

`true` = The column is the primary key.

notNull

`true` = The column does not allow null values.

Virtual table format

You use a virtual table that is based on the time series table to insert and query time series data.

```
{ collection : " virtualtable_name " , options : { timeseriesVirtualTable : { baseTableName : " table_name " , newTimeSeries : " calendar ( calendar_name ) , origin ( origin ) , container ( container_name ) [ { , irregular | , regular } ] , virtualTableMode : mode , timeseriesColumnName : " col_name " } } }
```

collection

The *virtualtable_name* is the name of the virtual table.

options

timeseriesVirtualTable

The definition of the virtual table.

baseTableName

The *table_name* is the name of the time series table.

newTimeSeries

The time series definition.

calendar

The *calendar_name* is the name of a calendar in the `system.timeseries.calendar` collection.

origin

The *origin* is the first time stamp in the time series. The data type is DATETIME YEAR TO FRACTION(5).

container

The *container_name* is the name of a container in the `system.timeseries.container` collection.

regular

Default. The time series is regular.

irregular

The time series is irregular.

virtualTableMode

The *mode* is the integer value of the `TSVMode` parameter that controls the behavior and display of the virtual table for time series data. For the settings of the `TSVMode` parameter, see [The TSVMode parameter on page](#) .

timeseriesColumnName

The *col_name* is the name of the **TimeSeries** column.

Example: Create a time series through the wire listener

This example shows how to create, load, and query a time series with the MongoDB API or the REST API through the wire listener.

Before you begin

Before you start this example, ensure these tasks are complete:

- Connect to a database in which to create the time series table. You run all methods in the database.
- Configure the wire listener for the MongoDB API or the REST API. For more information, see [Configuring the wire listener for the first time on page 8](#).
- Define a dbspace that is named **dbspace1**. For more information, see [Dbspaces on page](#) .

About this task

In this example, you create a time series that contains sensor readings about the temperature and humidity in a house. Readings are taken every 10 minutes. The following table lists the time series properties that are used in this example.

Table 34. Time series properties used in this example

Time series property	Definition
Timepoint size	10 minutes
When timepoints are valid	Every 10 minutes
Data in the time series	The following data: <ul style="list-style-type: none">• Timestamp• A float value that represents temperature• A float value that represents humidity
Time series table	The following columns:

Table 34. Time series properties used in this example (continued)

Time series property	Definition
	<ul style="list-style-type: none">• A meter ID column of type INTEGER• A TimeSeries data type column
Origin	2014-01-01 00:00:00.00000
Regularity	Regular
Where to store the data	In a container that you create
How to load the data	Through a virtual table
How to access the data	Through a virtual table

To create a time series with the MongoDB API **mongo** shell or the REST API:

1. Create a time series calendar. The time series calendar is named **ts_10min**, with a calendar and pattern start date of **2014-01-01 00:00:00**, a calendar pattern that is defined with intervals of minutes, and data is recorded in 10 minute increments after the origin.

MongoDB API

Add to the predefined `system.timeseries.calendar` collection.

```
db.system.timeseries.calendar.insert({"name":"ts_10min",
  "calendarStart":"2014-01-01 00:00:00",
  "patternStart":"2014-01-01 00:00:00",
  "pattern":{"type":"minute",
    "intervals":[{"duration":"1","on":"true"},
      {"duration":"9","on":"false"}]}})
```

REST API

Request:

Specify the POST method and the **system.timeseries.calendar** collection:

```
POST /stores_demo/system.timeseries.calendar
```

Data:

Specify the calendar attributes:

```
{"name":"ts_10min",
  "calendarStart":"2014-01-01 00:00:00",
  "patternStart":"2014-01-01 00:00:00",
  "pattern":{"type":"minute",
    "intervals":[{"duration":1,"on":true},
      {"duration":9,"on":false}]}}
```

Response:

The following response indicates that the operation was successful:

```
[{"ok":true}]
```

2. Create a **TimeSeries** row type. The row type is named **reading** and includes fields for timestamp, temperature, and humidity.

Example

MongoDB API

Add to the predefined system.timeseries.rowType collection.

```
db.system.timeseries.rowType.insert({"name":"reading",
  "fields":[{"name":"tstamp","type":"datetime year to fraction(5)"},
    {"name":"temp","type":"float"},
    {"name":"hum","type":"float"}]})
```

REST API

Request:

Specify the POST method and the **system.timeseries.rowType** collection:

```
POST /stores_demo/system.timeseries.rowType
```

Data:

Specify the row type attributes:

```
{"name":"reading",
  "fields":[{"name":"tstamp","type":"datetime year to fraction(5)"},
    {"name":"temp","type":"float"},
    {"name":"hum","type":"float"}]}
```

Response:

The following response indicates that the operation was successful:

```
[{"ok":true}]
```

3. Create a container. The container is named **c_0** and is created in the **dbspace1** dbspace, in the **reading** time series row, with a first extent size of **1000**, and with growth increments of **500**.

Example

MongoDB API

Add to the predefined system.timeseries.container collection.

```
db.system.timeseries.container.insert({"name":"c_0",
  "dbspaceName":"dbspace1",
  "rowTypeName":"reading",
  "firstExtent":1000,
  "nextExtent":500})
```

REST API

Request:

Specify the POST method and the system.timeseries.container collection:

```
POST /stores_demo/system.timeseries.container
```

Data:

Specify the container attributes:

```
{ "name": "c_0",
  "dbspaceName": "dbspace1",
  "rowTypeName": "reading",
  "firstExtent": 1000,
  "nextExtent": 500 }
```

Response:

The following response indicates that the operation was successful:

```
[{"ok": true}]
```

4. Create a time series table. The time series table is named **ts_data1** and includes **id** and **ts** columns.

Example**MongoDB API**

Create the **ts_data1** time series table:

```
db.runCommand({"create": "ts_data1",
  "columns": [{"name": "id", "type": "int", "primaryKey": true, "notNull": true},
              {"name": "ts", "type": "timeseries(reading)"}]})
```

REST API**Request:**

Specify the GET method:

```
GET /stores_demo/$cmd?query={create:"ts_data1",
  "columns": [{"name": "id", "type": "int", "primaryKey": true, "notNull": true},
              {"name": "ts", "type": "timeseries(reading)"}]}
```

Data:

None.

Response:

The following response indicates that the operation was successful:

```
[{"ok": true}]
```

5. Create a virtual table. The virtual table is named **ts_data1_v** and is based on the time series table that is named **ts_data1** and its timeseries column **ts**, using the **ts_10min** calendar, starting on **2014-01-01 00:00:00.00000**, in the time series container **c_0**, with the virtualTableMode parameter set to **0** (default).

Example**MongoDB API**

Create the **ts_data1_v** virtual table:

```
db.runCommand({"create": "ts_data1_v",
  "timeseriesVirtualTable":
    {"baseTableName": "ts_data1",
     "newTimeseries": "calendar(ts_10min),origin(2014-01-01
00:00:00.00000),container(c_0)",
```

```
"virtualTableMode":0,
"timeseriesColumnName":"ts"}})
```

REST API

Request:

Specify the GET method:

```
GET /stores_demo/$cmd?query={"create":"ts_data1_v",
"timeseriesVirtualTable":
  {"baseTableName":"ts_data1",
   "newTimeseries":"calendar(ts_10min),
                     origin(2014-01-01 00:00:00.000000),
                     container(c_0)",
   "virtualTableMode":0,
   "timeseriesColumnName":"ts"}}
```

Data:

None.

Response:

The following response indicates that the operation was successful:

```
[{"ok":true}]
```

6. Load records into the time series by inserting documents into the **ts_data1_v** virtual table.

Because this time series is regular, you are not required to include the time stamp. The first record is inserted for the origin of the time series, 2014-01-01 00:00:00.000000. The second record has the time stamp 2014-01-01 00:10:00.000000, and the third record has the time stamp 2014-01-01 00:20:00.000000.

MongoDB API

Add documents to the **ts_data1_v** virtual table:

```
db.ts_data1_v.insert([{"id":1,"temp":15.0,"hum":20.0},
{"id":1,"temp":16.2,"hum":19.0}, {"id":1,"temp":16.5,"hum":22.0}])
```

REST API

Request:

Specify the POST method:

```
POST /stores_demo/ts_data1_v
```

Data:

Specify the documents to load:

```
[{"id":1,"temp":15.0,"hum":20.0},
{"id":1,"temp":16.2,"hum":19.0},
{"id":1,"temp":16.5,"hum":22.0}]
```

Response:

The following response indicates that the operation was successful:

```
{"ok":true}
```

7. Query the time series data by using the **ts_data1_v** virtual table.

MongoDB API

Query the **ts_data1_v** virtual table:

```
db.ts_data1_v.find()

Results:
> db.ts_data1_v.find()
{"id":1,"tstamp":ISODate("2014-01-01T06:00:00Z"),"temp":15,"hum":20}
{"id":1,"tstamp":ISODate("2014-01-01T06:10:00Z"),"temp":16.2,"hum":19}
{"id":1,"tstamp":ISODate("2014-01-01T06:20:00Z"),"temp":16.5,"hum":22}
```

REST API

Request:

```
GET /stores_demo/ts_data1_v
```

Data:

None.

Response:

The following response indicates that the operation was successful:

```
[{"id":1,"tstamp":{"$date":1388556000000},"temp":15.0,"hum":20.0},
{"id":1,"tstamp":{"$date":1388556600000},"temp":16.2,"hum":19.0},
{"id":1,"tstamp":{"$date":1388557200000},"temp":16.5,"hum":22.0}]
```

Example queries of time series data by using the wire listener

These examples show how to query time series data by using the MongoDB API or the REST API.

Before using these examples, you must configure the wire listener for the MongoDB or REST API. For more information, see [Configuring the wire listener for the first time on page 8](#). These examples are run against the **stores_demo** database. For more information, see [dbaccessdemo command: Create demonstration databases on page](#) . These examples query the **ts_data_v** virtual table that stores the device ID in the **loc_esi_id** column.

- [List all device IDs on page 136](#)
- [List device IDs that have a value greater than 10 on page 137](#)
- [Find the data for a specific device ID on page 138](#)
- [Find and sort data with multiple qualifications on page 138](#)
- [Find all data for a device in a specific date range on page 139](#)
- [Find the latest data point for a specific device on page 140](#)
- [Find the 100th data point for a specific device on page 141](#)

For examples of aggregating or slicing time series data, see [Aggregate or slice time series data on page 141](#).

List all device IDs

This query returns all unique device IDs.

MongoDB API

Run a distinct command on the ts_data_v virtual table:

```
db.ts_data_v.distinct("loc_es_i_id")

Results:
["4727354321000111","4727354321046021","4727354321090954",...]
```

REST API**Request:**

Specify the GET method on the stores_demo database with the query parameter specified:

```
GET /stores_demo/$cmd?query={"distinct":"ts_data_v",
"key":"loc_es_i_id"}
```

Data:

None.

Response:

The following response indicates that the operation was successful:

```
[{"values":["4727354321000111","4727354321046021",
"4727354321090954",...],"ok":1.0}]
```

List device IDs that have a value greater than 10

This query returns the list of device IDs that have at least one measured value in the time series that is greater than 10.

MongoDB API

Run a distinct command on the ts_data_v table, with \$gt value comparison operator specified:

```
db.ts_data_v.distinct("loc_es_i_id",{"value":{"$gt":10}})

Results:
["4727354321046021","4727354321132574","4727354321289322",...]
```

REST API**Request:**

Specify the GET method with the query condition on the ts_data_v table and the \$gt value comparison operator specified:

```
GET /stores_demo/$cmd?query={"distinct":"ts_data_v",
"key":"loc_es_i_id","query":{"value":{"$gt":10}}}
```

Data:

None.

Response:

The following response indicates that the operation was successful:

```
[{"values":["4727354321046021","4727354321132574",
"4727354321289322",...],"ok":1.0}]
```

Find the data for a specific device ID

This query returns the data for the device with the ID of 4727354321046021.

MongoDB API

Run a find command on the ts_data_v virtual table with the loc_esid value specified:

```
db.ts_data_v.find({"loc_esid":4727354321046021})

Results:
{"loc_esid":"4727354321046021","measure_unit":"KWH",
"direction":"P","tstamp":ISODate("2010-11-10T06:00:00Z"),
"value":0.041}
{"loc_esid":"4727354321046021","measure_unit":"KWH",
"direction":"P","tstamp":ISODate("2010-11-10T06:15:00Z"),
"value":0.041}
{"loc_esid":"4727354321046021","measure_unit":"KWH",
"direction":"P","tstamp":ISODate("2010-11-10T06:30:00Z"),
"value":0.04}
...]
```

REST API

Request:

Specify the GET method on the ts_data_v virtual table, with the loc_esid specified on the query operator:

```
GET /stores_demo/ts_data_v?query=
{"loc_esid":4727354321046021}
```

Data:

None.

Response:

The following response indicates that the operation was successful:

```
[{"loc_esid":"4727354321046021","measure_unit":"KWH",
"direction":"P","tstamp":{"$date":1289368800000},"value":0.041},
{"loc_esid":"4727354321046021","measure_unit":"KWH",
"direction":"P","tstamp":{"$date":1289369700000},"value":0.041},
{"loc_esid":"4727354321046021","measure_unit":"KWH",
"direction":"P","tstamp":{"$date":1289370600000},"value":0.040},
...]
```

Find and sort data with multiple qualifications

This query finds all data for the device with the ID of 4727354321046021 with a value greater than 10.0 and a direction of P. The query returns the tstamp and value fields, and sorts the results in descending order by the value field.

To query for specific dates when using the REST API, convert the dates to milliseconds since the epoch. For example:

- 2011-01-01 00:00:00 = 1293861600000
- 2011-01-02 00:00:00 = 1293948000000

MongoDB API

Run a find command on the ts_data_v table, with the \$and boolean logical operator specified:

```
db.ts_data_v.find({"$and":[{"loc_esl_id":4727354321046021},
{"value":{"$gt":10.0}},{"direction":"P"}]},
{"tstamp":1,"value":1}).sort({"value":-1})
```

Results:

```
{"tstamp":ISODate("2011-01-25T16:15:00Z"),"value":14.58}
{"tstamp":ISODate("2011-01-26T00:45:00Z"),"value":12.948}
{"tstamp":ISODate("2011-01-26T02:30:00Z"),"value":12.768}
...
```

REST API**Request:**

Specify the GET method on the ts_data_v table, with the \$and boolean logical operator specified:

```
GET /stores_demo/ts_data_v?query={"$and":[{"loc_esl_id":
4727354321046021},{"value":{"$gt":10.0}},{"direction":"P"}]}
&fields={"tstamp":1,"value":1}&sort={"value":-1}
```

Data:

None.

Response:

The following response indicates that the operation was successful:

```
[{"tstamp":{"$date":12959721000000},"value":14.580},
{"tstamp":{"$date":12960027000000},"value":12.948},
{"tstamp":{"$date":12960090000000},"value":12.768},
...]
```

Find all data for a device in a specific date range

This query returns the data from midnight January 1, 2011 to January 2, 2011 for device ID 4727354321000111. The date that is queried is greater than 1293861600000 and less than 1293948000000. The query returns the tstamp and value fields.

MongoDB API

Run a find command on the ts_data_v table, with values specified for the \$and boolean logical query operator:

```
db.ts_data_v.find({"$and":[{"loc_esl_id":"4727354321000111"},
{"tstamp":{"$gte":ISODate("2011-01-01 00:00:00")}},
{"tstamp":{"$lt":ISODate("2011-01-02 00:00:00")}}]},
{"tstamp":"1","value":"1"})
```

Results:

```
{"tstamp":ISODate("2011-01-01T00:00:00Z"),"value":0.343 }
{"tstamp":ISODate("2011-01-01T00:15:00Z"),"value":0.349 }
{"tstamp":ISODate("2011-01-01T00:30:00Z"),"value":1.472 }
...]
```

REST API

Request:

Specify the GET method on the ts_data_v table in the stores_demo database, with values specified for the \$and boolean logical query operator:

```
GET /stores_demo/ts_data_v?query={"$and":
[{"loc_esi_id":4727354321000111}, {"tstamp":{"$gte":
{"$date":1293861600000}}}, {"tstamp":{"$lt":
{"$date":1293948000000}}}]&fields={"tstamp":1,"value":1}
```

Data:

None.

Response:

The following response indicates that the operation was successful:

```
[{"tstamp":{"$date":1293840000000},"value":0.343},
{"tstamp":{"$date":1293840900000},"value":0.349},
{"tstamp":{"$date":1293841800000},"value":1.472},
...]
```

Find the latest data point for a specific device

This query sets the sort parameter to order the tstamp field in descending order and sets the limit parameter to 1 to return only the latest value. The device ID is 4727354321000111 and the query returns the tstamp and value fields.

MongoDB API

Run a find command on the ts_data_v table, with sort and limit values specified:

```
db.ts_data_v.find({"loc_esi_id":"4727354321000111"},
{"tstamp":"1","value":"1"}).sort({"tstamp":-1}).limit(1)

Results:
{"tstamp":ISODate("2011-02-08T05:45:00Z"),"value":1.412 }
```

REST API

Request:

Specify the GET method on the ts_data_v table, with sort and limit values specified in the query parameter:

```
GET /stores_demo/ts_data_v?query={"loc_esi_id":4727354321000111}
&fields={"tstamp":1,"value":1}&sort={"tstamp":-1}&limit=1
```

Data:

None.

Response:

The following response indicates that the operation was successful:

```
[{"tstamp":{"$date":1297143900000},"value":1.412}]
```

Find the 100th data point for a specific device

This query sets the sort parameter to order the `tstamp` field in ascending order and sets the skip parameter to 100 to return the 100th value. The device ID is 4727354321000111 and the query returns the `tstamp` and `value` field.

MongoDB API

Run the find command on the `ts_data_v` table, with values specified for sort, limit and skip:

```
db.ts_data_v.find({"loc_es_i_id":4727354321000111},
{"tstamp":1,"value":1}).sort({"tstamp":1}).limit(1).skip(100)
```

Results:

```
{"tstamp":ISODate("2010-11-11T07:00:00Z"),"value":0.013}
```

REST API

Request:

Specify the GET method on the `ts_data_v` table, with values specified for sort, limit, and skip in the query parameter:

```
GET /stores_demo/ts_data_v?query={"loc_es_i_id":4727354321000111}
&fields={"tstamp":1,"value":1}&sort={"tstamp":1}&limit=1&skip=100
```

Data:

None.

Response:

The following response indicates that the operation was successful:

```
[{"tstamp":{"$date":1289458800000},"value":0.013}]
```

Aggregate or slice time series data

You can use the MongoDB aggregation pipeline commands to aggregate time series values or return a slice of a time series.

When you run an aggregation query on a time series table, internally the time series Transpose function converts the aggregated or sliced data to tabular format and then the `genBSON` function converts the results to BSON format. Therefore, the output of the `$group` or `$project` stage in the aggregation pipeline is collection-style JSON data. Any subsequent stages of the aggregation pipeline can process the data as JSON documents.

The aggregate and slice operations return JSON documents that include the primary key columns of the time series table. You can remove the primary key columns with the `$project` operator in the next stage of the aggregation pipeline.

To run the examples of aggregating and slicing time series data, create a JSON time series by following the instructions for loading hybrid data: [Example for JSON data: Create and load a time series with JSON documents on page 142](#).

- [Aggregate: The \\$group operator syntax on page 142](#)
- [Slice: The \\$slice operator syntax on page 145](#)

Aggregate: The \$group operator syntax

To aggregate time series values, you use the \$group operator and include a \$calendar object to define the aggregation period, and include one or more aggregation operator expressions to define the type of operation and the data to aggregate. The data to aggregate must be numeric and able to be cast to float values. The \$group operator produces the same results as running the time series AggregateBy function. If you have multiple **TimeSeries** columns in a table, you can aggregate values with the \$group operator for only the first **TimeSeries** column.

```
{ $group : { $calendar : { { <Calendar definition> | name : " calendar_name " } } , <Aggregation operator expression> } }
```

Calendar definition

```
" interval : number , "  
" timeunit : " unit " , "  
" start : " start_time " "  
" [ , end : " end_time " ] "  
" [ , discrete : { true | false } ] "  
" } "
```

Aggregation operator expression

```
" field_name : { "  
" { operator : { " $ column . field " | " $ column " } | $nth : [ { " $ column . field " | " $ column " } , position ] } "  
" } "
```

\$calendar

The calendar that defines the aggregation period. You can specify the name of an existing calendar with the following document: {name: "calendar_name"}. The calendar must exist in the **CalendarTable** table.

You can define a calendar for the aggregation operation with a document that contains the following fields:

interval

The *number* is a positive integer that represents number of time units in the aggregation period. For example, if the interval is 1 and the time unit is DAY, then the values are aggregated for each day.

timeunit

The *unit* is the size of the time interval for the aggregation period. Can be SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, or YEAR.

start

The *start_date* is the start date of the aggregation operation in DATETIME YEAR TO FRACTION(3) format.

end

Optional. The *end_date* is the end date of the aggregation operation in DATETIME YEAR TO FRACTION(3) format. If you omit the end date, the aggregation operation continues through the latest time series element.

discrete

Optional. Controls whether the data remains as discrete values or is smoothed to be continuous.

true = Default. The data remains discrete.

false = The data is smoothed. You might want to smooth your data if you want to treat your data as continuous, for example, temperature data. Smoothing data can accurately compensate for missing data. You can only use the \$avg, \$min, and \$max aggregation operators on smoothed data. You cannot use the \$sum, \$median, \$first, \$last, or \$nth aggregation operators on smoothed data.

For example, the following calendar definition produces an aggregate value per day for a month:

```
{ $calendar: { interval: 1,
               timeunit: "DAY",
               start: "2015-07-03 15:40:03.000",
               end: "2015-08-03 15:40:03.000",
               discrete: true }
```

Aggregation operator expression

The *field_name* is a descriptive name for the results of the aggregation operation.

The *operator* can be \$sum, \$avg, \$min, \$max, \$median, \$first, \$last, or \$nth. The \$nth operator requires a position value.

The *column* is the name of the column to aggregate in the **TimeSeries** row type. If the column contains BSON data, include a dot followed by the field name to aggregate within the BSON documents. For example, if the column name is **sensor_data** and the field name is **value**, the column name is specified as

```
"$sensor_data.value".
```

The *position* is an integer that follows the \$nth operator to represent the position of the value to return within the aggregation period. Positive integers begin at the first value. A position of 1 is the same as using the \$first operator. Negative integers begin at the latest value. A position of -1 is the same as using the \$last operator.

Example: Daily average value

The following example returns the daily average of a value over the period of three days for the **v1** field in the **sensor_data** column in the **tstable_j** table for the sensor 1:

```
db.tstable_j.aggregate(
  {$match: {id: 1 } },
  {$group: { $calendar: { interval: 1,
    timeunit: "DAY",
    start: "2014-03-01 00:00:00.000",
    end: "2014-03-03 23:59:59.000",
    discrete: true },
    val_AVG: {$avg: "$sensor_data.v1"} } }
)

{
  "result" : [
    {
      "id" : "1",
      "tstamp" : ISODate("2014-03-01T00:00:00Z"),
      "val_avg" : 1.416666666666667
    },
    {
      "id" : "1",
      "tstamp" : ISODate("2014-03-02T00:00:00Z"),
      "val_avg" : 1.4437500000000003
    },
    {
      "id" : "1",
      "tstamp" : ISODate("2014-03-03T00:00:00Z"),
      "val_avg" : 1.4447916666666671
    }
  ],
  "ok" : 1
}
```

Example: Get the maximum value for each month

The following example returns the maximum value for each month over a six-month period for the **v2** field in the **sensor_data** column in the **tstable_j** table for the sensor 1:

```
db.tstable_j.aggregate(
  {$match: {id: 1 } },
  {$group: { $calendar: { interval: 1,
    timeunit: "MONTH",
    start: "2014-01-01 00:00:00.000",
    end: "2014-6-30 23:59:59.000",
    discrete: true },
    maximum: {$max: "$sensor_data.v2"} } }
)

{
  "result" : [
    {
      "id" : "1",
      "tstamp" : ISODate("2014-01-01T00:00:00Z"),
      "maximum" : 22.9
    },
  ],
}
```



```

{
  {
    "id" : "1",
    "tstamp" : ISODate("2014-02-01T00:00:00Z"),
    "maximum" : 23.4
  },
  {
    "id" : "1",
    "tstamp" : ISODate("2014-03-01T00:00:00Z"),
    "maximum" : 23.1
  },
  {
    "id" : "1",
    "tstamp" : ISODate("2014-04-01T00:00:00Z"),
    "maximum" : 22.9
  },
  {
    "id" : "1",
    "tstamp" : ISODate("2014-05-01T00:00:00Z"),
    "maximum" : 24.0
  },
  {
    "id" : "1",
    "tstamp" : ISODate("2014-06-01T00:00:00Z"),
    "maximum" : 24.8
  }
],
"ok" : 1
}

```

Slice: The \$slice operator syntax

To slice a time series, you use the \$project operator to identify the time series and include a document with a \$slice operator to specify the time range of the time series elements to return. The \$slice operator produces the same results as running the time series Clip or ClipCount functions.

```
{ $project : { time_series : { $slice : { N [ N , flag ] [ tstamp , N [ , flag ] ] [ begin_tstamp , end_tstamp [ , flag ] ] } } } }
```

\$project

The *time_series* is the name of the time series column.

\$slice

The *N* is an integer that represents the number of elements to return. Positive values return elements from the beginning of the time series or starting at the specified time stamp. Negative values return elements from the end of the time series or ending with the specified time stamp.

The *tstamp* is a DATETIME value that represents the start or end time stamp of the elements to return.

The *begin_tstamp* is the beginning time stamp of the elements to return.

The *end_tstamp* is the ending time stamp of the elements to return.

The *flag* controls the configuration of the resulting time series. For values, see the [Clip function on page 145](#).

Example: Get the next five elements

The following example returns the first five elements, beginning at March 14, 2014, at 9:30 AM, from the **tstable_j** table for the sensor with the ID of 1:

```
db.tstable_j.aggregate(
  { $match: { id: 1 } },
  { $project: { sensor_data: { $slice: ["2014-03-14 09:30:00.000", 5] } } }
)

{
  "result" : [
    {
      "id" : "1",
      "tstamp" : ISODate("2014-03-14T09:30:00Z"),
      "v1" : 1.7,
      "v2" : 20.9
    },
    {
      "id" : "1",
      "tstamp" : ISODate("2014-03-14T09:45:00Z"),
      "v1" : 1.6,
      "v2" : 17.4
    },
    {
      "id" : "1",
      "tstamp" : ISODate("2014-03-14T10:00:00Z"),
      "v1" : 1.6,
      "v2" : 20.3
    },
    {
      "id" : "1",
      "tstamp" : ISODate("2014-03-14T10:15:00Z"),
      "v1" : 1.8,
      "v2" : 20.4
    },
    {
      "id" : "1",
      "tstamp" : ISODate("2014-03-14T10:30:00Z"),
      "v1" : 1.3,
      "v2" : 17.1
    }
  ],
  "ok" : 1
}
```

Example: Get the previous three elements

The following example returns the previous three elements, ending at March 14, 2014, at 9:30 AM, from the **tstable_j** table for the sensor with the ID of 1:

```
db.tstable_j.aggregate(
  { $match: { id: 1 } },
  { $project: { sensor_data: { $slice: ["2014-03-14 09:30:00.000", -3] } } }
)
```

```
{
  "result" : [
    {
      "id" : "1",
      "tstamp" : ISODate("2014-03-14T09:00:00Z"),
      "v1" : 1,
      "v2" : 22.8
    },
    {
      "id" : "1",
      "tstamp" : ISODate("2014-03-14T09:15:00Z"),
      "v1" : 1.8,
      "v2" : 21.6
    },
    {
      "id" : "1",
      "tstamp" : ISODate("2014-03-14T09:30:00Z"),
      "v1" : 1.7,
      "v2" : 20.9
    }
  ],
  "ok" : 1
}
```

Example: Get elements in a range

The following example returns the elements between March 14, 2014, at 9:30 AM and March 14, 2014, at 10:30 AM, from the **tstable_j** table for the sensor with ID 1:

```
db.tstable_j.aggregate(
  { $match: { id: 1 } },
  { $project: { sensor_data: { $slice: ["2014-03-14 09:30:00.000",
                                         "2014-03-14 10:30:00.000"] } } }
)

{
  "result" : [
    {
      "id" : "1",
      "tstamp" : ISODate("2014-03-14T09:30:00Z"),
      "v1" : 1.7,
      "v2" : 20.9
    },
    {
      "id" : "1",
      "tstamp" : ISODate("2014-03-14T09:45:00Z"),
      "v1" : 1.6,
      "v2" : 17.4
    },
    {
      "id" : "1",
      "tstamp" : ISODate("2014-03-14T10:00:00Z"),
      "v1" : 1.6,
      "v2" : 20.3
    },
    {
      "id" : "1",
      "tstamp" : ISODate("2014-03-14T10:15:00Z"),

```

```

        "v1" : 1.8,
        "v2" : 20.4
    },
    {
        "id" : "1",
        "tstamp" : ISODate("2014-03-14T10:30:00Z"),
        "v1" : 1.3,
        "v2" : 17.1
    }
],
"ok" : 1
}

```

Loading time series data with the MQTT protocol

You can load JSON documents into time series through the MQTT wire listener. The MQTT wire listener publishes data directly to the time series base table by internally running time series loader routines.

Before you begin

You cannot create a time series through the MQTT wire listener. Create a JSON time series with the REST API, the MongoDB API, or SQL statements.

For instructions on creating a JSON time series with SQL statements, see [Example for JSON data: Create and load a time series with JSON documents on page](#) .

To load JSON data through the MQTT wire listener:

1. Set the following parameters in the wire listener configuration file:
 - Set the wire listener type to MQTT: **listener.type=mqtt**
 - Optional. Set the number of connections between the wire listener and each time series table: Set **timeseries.loader.connections** to the number of connections that you want.
2. Restart the wire listener.
3. From the MQTT clients, load the data into the time series table by publishing data as JSON documents.
 - For BSON timeseries tables, i.e. tables where the timeseries row type contains only a timestamp plus one BSON column, the *message* argument of the PUBLISH packet must contain the following fields within the JSON documents:
 - One or more fields that identifies the primary key of the time series table. The field names must be the same as the primary key column names in the time series table.
 - A field that identifies the time stamp. The field name must be the same as the time stamp column in the **TimeSeries** row type.
 - One or more fields to insert into the BSON column in the **TimeSeries** row type. All fields that are not identified as a primary key column or the time stamp field are inserted into the BSON column.

For example"

```
{ "pkey": value, "tstamp": value, "field1": value, "field2": value, ...}
```

where "pkey" is the name of the primary key column, "tstamp" is the name of the timestamp column, and "field1", "field2", etc. are whatever fields you want in the BSON column of the timeseries.



Note: For BSON timeseries tables, you do not use the BSON column name from the row type. The timeseries loader will extract the primary key field(s) and timestamp fields; all other fields will be inserted into the BSON column of the row type.

- For non-BSON timeseries tables, the *message* argument of the PUBLISH packet must contain the following fields within the JSON documents:
 - One or more fields that identify the primary key of the time series table. The field names must be the same as the primary key column names in the time series table.
 - A field that identifies the time stamp. The field name must be the same as the time stamp column in the **TimeSeries** row type.
 - One or more fields that match the names of the other columns in the **TimeSeries** row type.

For example"

```
{ "pkey": value, "tstamp": value, "rowtypeField1": value, "rowTypeField2": value, ...}
```

where the fields in addition to primary key and timestamp match the column name in the timeseries row type.

If the timeseries row type has an integer column (named "intData") and a BSON column (named "bsonData"), the data will take the following format:

```
{ "pkey": value, "tstamp": value, "intData": 10, "bsonData":
  { "value1": 1.234, "label": "any fields can go here within the bsonData document..."}}
```

Example

Example 1

The following example creates a BSON **TimeSeries** row type, a time series table, a time series container, and a time series instance:

```
CREATE ROW TYPE ts_data_j2(
    tstamp      datetime year to fraction(5),
    tsdata      BSON);

CREATE TABLE IF NOT EXISTS tstable_j2(
    id  VARCHAR(50) NOT NULL PRIMARY KEY,
    ts  timeseries(ts_data_j2)
) LOCK MODE ROW;

EXECUTE PROCEDURE
    TSContainerCreate('container_j', 'dbspace1', 'ts_data_j2', 512, 512);

INSERT INTO tstable_j2 VALUES(1, 'origin(2014-01-01 00:00:00.000000),
                                calendar(ts_15min), container(container_j),
                                regular, threshold(0), []');
```

For this example, the *message* argument has an **id** field for the primary key, a **tstamp** field for the time stamp, and two fields for the BSON column:

```
{"id": "value", "tstamp": "time_stamp", "reading": number, "sensor_type": "string"}
```

The following sample Java code connects a client to the MQTT wire listener, loads a sensor reading, and disconnects from the client:

```
String broker = "tcp://localhost:1883";
String topicName = "mydb/tstable_j2";
String clientId = "mqttclient1";
MemoryPersistence persistence = new MemoryPersistence();

MqttClient sampleClient = new MqttClient(broker, clientId, persistence);
MqttConnectOptions connOpts = new MqttConnectOptions();
connOpts.setCleanSession(true);
sampleClient.connect(connOpts);

String content = "[{ \"id\": \"sensor1234\", \"tstamp\": \"2016-01-01 00:00:00\" ,
                    \"reading\": 87.5, \"sensor_type\": \"TEMP\"}]";
MqttMessage message = new MqttMessage(content.getBytes());
message.setQos(2);
sampleClient.publish(topicName, message);

sampleClient.disconnect();
```

Troubleshooting Informix® JSON compatibility

Several troubleshooting techniques, tools, and resources are available for resolving problems that you encounter with Informix® JSON compatibility.

Problem	Solution
How do I start the wire listener?	<p>If the wire listener does not automatically start:</p> <ol style="list-style-type: none">1. Verify that the user was created. For more information, see Configuring the wire listener for the first time on page 8.2. Manually start the wire listener. For more information, see Starting the wire listener on page 44.
How can I debug wire listener problems?	<p>From the wire listener command line, run the <code>-loglevel level</code> command, where <i>level</i> is the logging level. Log level options are:</p> <ul style="list-style-type: none">• error• warn• info• debug• trace <p>For more information, see Wire listener command line options on page 42.</p>

Problem	Solution
How can I view all of the current properties for the wire listener properties file?	From the wire listener command line, you can run the <code>-listProperties</code> command. This command prints all of the supported properties and their default values. For more information, see The wire listener configuration file on page 9 .
How do I access the wire listener help?	You can view a list of available command line options by running the <code>-help</code> command.

Index

Special Characters

- \$group
- operators 101

A

- addShard command 56
- admin() functions
 - cdr add trustedhost argument 55
- Aggregating time series data 141
- aggregation framework operators
 - \$group 101
 - pipeline 101
 - supported 101
- anti Cross-Site Request Forgery 112
- anti-CSRF 112
- API 103, 103
- authentication
 - authentication.enable 49, 50
 - MongoDB 49, 50
 - user access 49, 50
- Authentication
 - MongoDB 47
 - PAM 47, 51
- authentication.enable
 - properties file 9
- authentication.localhost.bypass.enable
 - properties file 9

B

- bts
 - \$ifxtext 100
 - \$text 100
 - query 100

C

- cdr add trustedhost argument 55
- changeShardCollection command 56, 61
- collection methods
 - collection 69
 - db.collection 69
 - supported 69
 - unsupported 69
- Collections for configuring time series 127
- command line
 - arguments 42
- command.listDatabases.sizeStrategy
 - properties file 9
- commands
 - buildinformation 42
 - command line 42
 - config 42
 - database 73
 - logfile 42
 - loglevel 42
 - port 42
 - projection 95
 - query 95
 - start 42
 - stop 42
 - update 99
 - version 42
 - wait 42
- communication
 - encrypted 53, 53
- compatible.maxBsonObjectSize.enable
 - properties file 9
- Concepts

- MongoDB
 - and Informix 68
- configuration file
 - authorized user 8
 - configuring 8
 - creating 8
 - DBSERVERALIASES 8
 - dynamic host IPv6 8
 - ifxjson 8
 - installing 8
 - modify 41
 - MongoDB 8
 - MQTT 8
 - REST API 8
 - sample 8
 - sharding 8
 - template 8
- copy
 - properties file 9
- cursor.idle.timeout
 - properties file 9

D

- database commands
 - aggregation 73
 - diagnostic 73
 - instance administration 73
 - query and write operation 73
 - replication 73
 - sharding 73
 - supported 73
 - unsupported 73
- database.buffer.enable
 - properties file 9
- database.cache.enable
 - properties file 9
- database.create.enable
 - properties file 9
- database.dbspace
 - properties file 9
- database.locale.default
 - properties file 9
- database.log.enable
 - properties file 9
- dataSource.IFX_ISOLATION_LEVEL 105
- db.servers 105
- dbspace.strategy
 - properties file 9
- DELETE
 - example 115
 - REST API 115
 - support 115
- deleteInsert
 - properties file 9
- documentIdAlgorithm
 - properties file 9

E

- encryption
 - wire listener 52, 53, 53
- ensureIndex command 56, 58, 61

F

- files
 - configuration file 41
 - properties file 41
- Files

- .properties 55
- fragment.count
- properties file 9
- Functions, SQL administration API
 - cdr add trustedhost argument 55

G

- GET
 - example 115
 - REST API 115
 - support 115

H

- high availability
 - JSON 54
 - wire listener 54
- Horizontal partitioning 54, 55, 56, 58, 60, 61, 63
- hostname 105
- HTTP 103, 103
- HTTP Basic Authentication 111
- HTTPS 103, 103, 112

I

- IFMXMONGOAUTH environment variable 51
- ifxjson
 - configuration file 8
 - properties file 8
 - replication 8
 - sharding 8
 - user permissions 8
- import
 - collections 69
 - data 69
- index
 - create 71
 - createIndex
 - supported options 71
 - ensureIndex
 - supported options 71
 - supported options 71
- index.cache.enable
 - properties file 9
- index.cache.update.interval
 - properties file 9
- Informix
 - configuration parameters
 - REMOTE_SERVER_CFG 55
- Informix
 - REST API listener
 - querying time series 136
- Informix
 - wire listener
 - creating time series 131
- insert.batch.enable
 - properties file 9
- insert.batch.queue.enable
 - properties file 9
- insert.batch.queue.flush.interval
 - properties file 9
- insert.preparedStatement.cache.enable
 - properties file 9
- IPv4
 - configuration 8
- IPv6
 - configuration 8

J

- Java 5, 104

- Java requirement 7
- JSON
 - dots in field names 5
 - SQL access 91, 123
- JSON compatibility
 - about 3
 - MongoDB 3
- json.format 105
- jsonListener.log
 - location 150

K

- killCursors 100

L

- listener.connectionPool.closeDelay.time
 - properties file 9
- listener.connectionPool.closeDelay.timeunit
 - properties file 9
- listener.http.accessControlAllowCredentials
 - properties file 9
- listener.http.accessControlAllowHeaders
 - properties file 9
- listener.http.accessControlAllowMethods
 - properties file 9
- listener.http.accessControlAllowOrigin
 - properties file 9
- listener.http.accessControlExposeHeaders
 - properties file 9
- listener.http.accessControlMaxAge
 - properties file 9
- listener.http.headers.size.maximum
 - properties file 9
- listener.idle.timeout
 - properties file 9
- listener.idle.timeout.minimum
 - properties file 9
- listener.input.buffer.size
 - properties file 9
- listener.onException
 - properties file 9
- listener.output.buffer.size
 - properties file 9
- listener.pool.keepAliveTime
 - properties file 9
- listener.pool.queue.size
 - properties file 9
- listener.pool.size.core
 - properties file 9
- listener.pool.size.maximum
 - properties file 9
- listener.port
 - properties file 9
- listener.rest.cookie.domain
 - properties file 9
- listener.rest.cookie.httpOnly
 - properties file 9
- listener.rest.cookie.length
 - properties file 9
- listener.rest.cookie.name
 - properties file 9
- listener.rest.cookie.path
 - properties file 9
- listener.rest.cookie.secure
 - properties file 9
- listener.type
 - MongoDB 9
 - REST API 9
- listShards command 63
- log file
 - about 46

- jsonListener.log 150
 - location 46
 - logging level 42
 - settings 42, 46
- log4j 113
- Logback 46

M

- metadata.cache.timeout 105
- methods
 - collection 69
 - mongo shell 69
- mongo.api.version
 - properties file 9
- MongoDB
 - dependencies 5
 - getting started 65
 - supported version 5
- MongoDB API
 - creating time series 131
 - querying time series 136
 - relational tables 92
 - SQL 92
- MongoDB
 - API wire listener
 - start 44
 - authentication 47
 - commands
 - addShard 56
 - changeShardCollection 56, 61
 - ensureIndex 56, 58, 61
 - listShards 63
 - shardCollection 56, 58
 - concepts 68
 - language drivers 69
 - shell
 - version 69
 - utilities
 - mongodump 69
 - mongoexport 69
 - mongoimport 69
 - mongorestore 69
- MQTT
 - loading time series 148

N

- non-root install
 - considerations 150

O

- onstat -g shard 60
- operators
 - aggregation framework
 - \$group 101
 - pipeline 101
 - Informix
 - support 95
 - MongoDB
 - 95
 - projection 95
 - query 95
 - supported 95
 - unsupported 95

- update 99

P

- PAM authentication 47, 51
- pipeline
 - operators 101
- Pluggable authentication module 51
- pool.connections.maximum
 - properties file 9
- pool.connectionTimeout 105
- pool.idle.timeout
 - properties file 9
- pool.idle.timeunit
 - properties file 9
- pool.idleTimeout 105
- pool.maximumPoolSize 105
- pool.minimumIdle 105
- pool.semaphore.timeout
 - properties file 9
- pool.semaphore.timeunit
 - properties file 9
- pool.service.interval
 - properties file 9
- pool.service.timeunit
 - properties file 9
- pool.size.initial
 - properties file 9
- pool.size.maximum
 - properties file 9
- pool.size.minimum
 - properties file 9
- pool.type
 - properties file 9
- pool.typeMap.strategy
 - properties file 9
- port 105
- POST
 - example 115
 - REST API 115
 - support 115
- preparedStatement.cache.enable
 - properties file 9
- preparedStatement.cache.size
 - properties file 9
- projection operators
 - supported 95
 - unsupported 95
- properties file 55
 - configuring 8
 - creating 8
 - DBSERVERALIASES 8
 - dynamic host IPv6 8
 - MongoDB 8
 - MQTT 8
 - optional 9
 - parameters 9
 - required
 - url 9
 - REST API 8
 - sample 8
 - sharding 8, 9
 - template 8
 - view all properties 150
- properties file parameters
 - properties file parameters
 - sharding.parallel.query.enable 55
 - url 55
 - sharding.enable 55
 - url 55

Q

- query operators

- supported 95
- unsupported 95
- query.default.limit 105
- query.maximumFieldSize 105
- query.timeseries.default.limit 105

R

- relational database
 - \$sql 91
 - run commands using MongoDB 91
 - run MongoDB operations 92
 - SQL passthrough queries through REST 123
 - system.sql 91
- REMOTE_SERVER_CFG configuration parameter 55
- response.documents.count.maximum
 - properties file 9
- response.documents.size.maximum
 - properties file 9
- rest
 - build information 124
 - debug 124
 - forbidden 124
 - help 124
 - version 124
- REST 103, 103
 - authentication 111
 - config 105
 - configuration file 105
 - dependencies 104
 - getting started 104
 - logging 113
 - properties file 105
 - security 112
- REST API
 - configuring 8
 - creating time series 131
 - DELETE 115
 - examples 115
 - GET 115
 - listener.type 8
 - POST 115
 - querying time series 136
 - syntax 115
- REST API listener
 - querying time series 136
- REST API wire listener
 - start 44
- rest.session.timeout 105

S

- SCRAM-SHA-1 authentication 47
- search
 - bts 100
 - text 100
- security.csrf.token.enable 105, 112
- security.sql.passthrough
 - properties file 9
- Shard cluster
 - viewing participants 63
- shard clusters 54
- Shard clusters 54
- shard servers
 - adding 60
 - deleting 60
 - listing 60
- Shard servers 54
- Shard-cluster definition
 - changing 56, 61
 - creating 56, 56, 58
- shardCollection command 56, 58

- sharding
 - authorized user 8
 - enable 55
 - ifxjson 8
 - JSON 54, 55, 56, 56, 58, 60, 61, 63
 - properties file 8
 - Relational data 56, 58, 61
 - shard-cluster defining 56, 56, 58, 61
 - shard-cluster viewing 63
 - update.client.strategy 9
 - wire listener 55
- sharding.enable
 - properties file 9
- sharding.enable configuration parameter 55
- sharding.parallel.query.enable configuration parameter 55
- software requirement 5, 104
- SQL
 - \$sql 91
 - JSON access 91, 123
 - REST listener 123
 - system.sql 91
 - using MongoDB API 91
- SQL administration API functions
 - cdr add trustedhost argument 55
- SSL
 - client applications 53
 - database server 53
 - wire listener 53, 53
- start
 - MongoDB
 - API wire listener
 - command line 44
 - listener.type 44
 - start REST API wire listener
 - command line 44
 - listener.type 44
- starting
 - REST 104
- stop wire listener
 - command line 46

T

- task() functions
 - cdr add trustedhost argument 55
- Time series
 - aggregate 141
 - collections 127
 - creating 126
 - creating with MongoDB
 - API
 - 131
 - creating with REST API 131
 - example for wire listener 131
 - loading with MQTT listener 148
 - MongoDB API 125
 - MQTT protocol 125
 - query example for MongoDB
 - API listener
 - 136
 - query example for REST API listener 136
 - query with MongoDB
 - API
 - 136
 - query with REST API 136
 - REST API 125
 - slice 141
 - wire listener 125

- timeseries.loader.connections
 - properties file 9
- timeseries.loader.flush.interval 105
- timeseries.loader.log.level 105
- timeseries.loader.put.size.maximum 105
- timeseries.loader.session.timeout 105
- TLS 112
- tls.ciphers 105
- tls.enable 105
- tls.keystore.file 105
- tls.keystore.password 105
- tls.keystore.type 105
- tls.pem.cert 105
- tls.protocols 105

U

- updateCursor
 - properties file 9
- update operators
 - supported 99
 - unsupported 99
- update.client.strategy
 - properties file 9
- update.mode
 - properties file 9
- update.one.enable
 - properties file 9
- url configuration parameter 55
- url.jdbc.afterNewConnectionCreation
 - properties file 9
- user permission
 - grant access 8
 - required access 8
 - sharding 8

V

- version
 - wire listener 42
- vertx.threads.poolSize 105

W

- wire listener
 - build information 42
 - change 41
 - debug 150
 - getting started 65
 - help 150
 - log file 150
 - modify 41
 - stop 46
 - version 42
- Wire listener
 - creating time series 131
 - Java version 7
 - MongoDB
 - 7
 - using 7
- wire listener parameters 9
- Wire listener parameters
 - sharding.enable 55
 - sharding.parallel.query.enable 55
 - url 55, 55