MongoDB Document Database Connector

New in 10.1.5. MongoDB is integrated across all four FrameworX data connection surfaces - Dataset Provider, Device Protocol, UNS TagProvider, and Historian StorageLocation. The Dataset Provider also accepts a SQL subset dialect that the connector translates to MongoDB filter, sort, and projection automatically.

MongoDB document databases.

- Name: **MongoDB**
- Version: 1.0.0.1
- Protocol: Proprietary (MongoDB Wire Protocol)
- Interface: TCP/IP
- Runtime: .NET Framework 4.8 (Windows) and .NET 10 (cross-platform)
- Configuration:
  - Datasets / DBs
  - Devices / Channels
  - UNS / TagProviders
  - Historian / Storage Locations
- Minimum server version: MongoDB 6.0 (recommended 7.0 LTS)

Overview
Dataset Provider
Worked example
Device Protocol
UNS TagProvider
Historian StorageLocation
Known limitations and deferrals
Driver Revision History

Overview

MongoDB is the first document database that FrameworX integrates across all four data connection surfaces. One MongoDB instance can serve any combination of these surfaces from the same solution.

Surface	Configure in	What it does
Dataset Provider	Datasets / DBs / Provider = `MongoDB.Driver`	Query collections with JSON directives, aggregation pipelines, or the built-in SQL subset dialect. CRUD on collections through Dataset Tables.
Device Protocol	Devices / Channels / Protocol = `MongoDB`	Map individual document fields as Channel, Node, and Point tags. Write updates the latest document in a collection.
UNS TagProvider (TagDiscovery)	UNS / TagProviders / Protocol = `MongoDB`	Browse collections as a tag tree. Register document fields as UNS tags. Updates write back with `$set`.
Historian StorageLocation	Historian / Storage Locations / Protocol = `MongoDB`	Store historian tag samples in MongoDB Time Series Collections. Append-style writes.

A MongoDB UnsTagProvider row and a Historian StorageLocation row can both point at the same MongoDB instance.

Supported MongoDB versions

- Floor: **MongoDB 6.0** (Time Series Collections require 6.0+).
- Recommended target: **MongoDB 7.0** (current LTS).
- Tested against 6.0, 7.0, 8.0.

Prerequisites

MongoDB .NET drivers ship unconditionally with FrameworX. No separate install and no optional pack. Approximate footprint is 4 MB per framework (net48 and net10).

Dataset Provider

Use MongoDB with the Datasets module to read and write document data from a FrameworX solution. The provider exposes four query shapes through the standard Dataset Query object and binds full document collections through Dataset Table for CRUD work.

Steps to connect:

1. Install a MongoDB server and create a database.
2. Create a Database Connection with the MongoDB provider.
3. Configure the connection string and test the connection.
4. Create Dataset Queries for Find, Aggregate, Count, or SQL subset.
5. Create Dataset Tables to bind collections for insert, update, and delete.
6. Call the queries and tables from scripts, displays, or reports.

Installation

Install a MongoDB server (6.0 or newer, 7.0 LTS recommended), create a database and user, then move to the FrameworX configuration below.

Install the MongoDB server

Follow the official MongoDB installation guide for your platform at mongodb.com/docs/manual/installation. Summary by target:

- **Windows:** download the MongoDB Community Server MSI and run the installer. The default install registers `mongod` as a Windows service on port 27017.
- **Linux:** install the distro package per the vendor instructions (for example, `apt install mongodb-org` on Debian or Ubuntu with the MongoDB repository configured). Start the service with `systemctl start mongod`.
- **macOS:** `brew tap mongodb/brew && brew install mongodb-community`, then `brew services start mongodb-community`.
- **Docker:** `docker run -d --name fx-mongo -p 27017:27017 mongo:7`.

Install the `mongosh` command-line shell alongside the server. Most platform installers include it by default.

Verify the server is reachable

Run the command below to confirm the server responds:

```
mongosh "mongodb://localhost:27017" --quiet --eval "db.adminCommand('ping').ok"
```

A successful ping returns 1.

Create the database and user

Run these commands in `mongosh` to create a target database, a starter collection, and a user for FrameworX to authenticate as:

```
use plant_data

db.readings.insertOne({ plant: "Plant01", value: 0, ts: new Date() })

db.createUser({
  user: "fxuser",
  pwd: "<choose-a-strong-password>",
  roles: [ { role: "readWrite", db: "plant_data" } ]
})
```

The user is stored against the `plant_data` database above. Use `plant_data` as the **Auth Source** value in the FrameworX connection string in the next section, or use `admin` when the user lives in the admin database.

Platform compatibility

The MongoDB connector targets **netstandard2.0** and runs unchanged on both **.NET Framework 4.8** (Designer and Windows Runtime) and **.NET 10** (cross-platform Runtime on Windows, Linux, and containers). The vendored MongoDB driver ships in both flavors with the FrameworX installation. The `mongod` server is OS-native and has no .NET dependency.

Configuration

Follow the steps below to connect FrameworX to a MongoDB server.

1. Access **Datasets / DBs**.
2. Click the **plus icon** to create a new Database Connection.
3. In the **Name** field, enter a name for the connection, for example **MongoDB1**.
4. Choose **MongoDB Data Provider** as the **Provider**.
5. Click **OK**.
6. In the data grid, click the **Connection String** column of the newly created row.
7. Configure the connection. The Designer dialog shows structured fields for **Server**, **Port**, **User**, **Password**, **Database**, **Auth Source**, and **TLS**. For replica sets and Atlas clusters, use the **Advanced Connection URI** field with a `mongodb://` or `mongodb+srv://` URI, which overrides the structured fields.
8. Click **Test** to verify the connection with the MongoDB server.

Connection string options

Field	Required	Description	Example
Server	Yes	Host name or IP of the MongoDB server.	localhost

Port	No	TCP port. Default is 27017.	27017
User	No	Database user. Leave empty for unauthenticated access.	appuser
Password	No	User password. Stored encrypted in the solution file.	********
Database	Yes	Target database name.	plant_data
Auth Source	No	SCRAM authentication database. Default is `admin`.	admin
TLS	No	Set to **true** for TLS/SSL connections.	true
Connection URI	No	Full MongoDB URI. Overrides the structured fields. Use for replica sets and `mongodb+srv://` Atlas URIs.	mongodb+srv://user:pwd@cluster0.example.net/?retryWrites=true

Query shapes

MongoDB queries accept four input shapes. The provider routes the **Command Text** of a Dataset Query to the correct driver operation based on the input shape:

Input shape	Operation	Description
Plain collection name, no braces or brackets	Find	Returns all documents in the collection, with columns auto-detected from the first document.
JSON object starting with `{`	Find or Count	Reads `collection`, `filter`, `sort`, and `limit` fields. If the object has a `count` key, runs Count and returns a single-row table.
JSON array starting with `[`	Aggregate	Treated as an aggregation pipeline. Requires a default collection set on the Dataset Query.
SQL statement starting with `SELECT`	SQL subset (new in 10.1.5)	Parsed by the built-in SQL-to-MongoDB translator. See the next subsection.

SQL subset dialect (new in 10.1.5)

New in Phase B of the 10.1.5 MongoDB connector, users can write standard `SELECT` statements in the Dataset Query **SqlStatement**. The connector translates the statement to a MongoDB filter, sort, and projection automatically and executes it through the same driver that handles JSON directives and aggregation pipelines.

Supported grammar (v1)

The translator accepts this subset of standard SQL:

- **Clauses:** `SELECT cols | *`, `FROM collection`, `WHERE expr`, `ORDER BY col [ASC|DESC] (, col [ASC|DESC]...)`, `LIMIT n`.
- **Comparison operators:** =, <>, `!=`, <, <=, >, >=.
- **Boolean operators:** `AND`, `OR`, `NOT`.
- **Predicates:** `IS [NOT] NULL`, `[NOT] BETWEEN ... AND ...`, `[NOT] IN (...)`, `[NOT] LIKE`.
- **Identifier forms:** bare (`plant`), double-quoted (`"plant"`), bracketed (`[plant]`), back-ticked (`` `plant` ``).
- **Dotted BSON paths:** identifiers like `site.line.station` resolve to nested BSON fields.
- **Numeric literals:** integer and decimal, including negative values such as `-42` or `-3.14`.
- **Case sensitivity:** identifiers are case-sensitive by design. BSON field names are case-sensitive in MongoDB and the translator preserves that behavior.

Semantic notes

- `x IS NULL` and `x = NULL` both match documents where the field is missing or has a `null` value. This follows the MongoDB idiom for missing-or-null match.
- `LIKE` translates to an anchored regular expression. The wildcard `%` becomes `.*` and `_` becomes `.`. Case-folding collation is not applied.
- No collation. Comparisons follow BSON default collation.
- No parameter plumbing at the translator layer. The Dataset host pre-interpolates `@Tag.Name` references before the statement reaches the SQL-to-MongoDB bridge, so tag values appear as literals in the parsed statement.

Three query shapes side by side

All three shapes below coexist at the same **Provider = MongoDB.Driver**. Pick the shape that fits the call site.

JSON directive.

```
{ "collection": "readings", "filter": { "plant": "Plant01" }, "sort": { "ts": -1 }, "limit": 10 }
```

Aggregation pipeline.

```
[
  { "$match": { "plant": "Plant01" } },
  { "$sort": { "ts": -1 } },
  { "$limit": 10 }
]
```

SQL subset.

```
SELECT plant, value, ts
FROM readings
WHERE plant = 'Plant01'
ORDER BY ts DESC
LIMIT 10
```

Supported grammar

The SQL subset covers `SELECT` with `WHERE`, `ORDER BY`, `LIMIT`, `OFFSET`, `GROUP BY` and `DISTINCT`; the aggregate functions `COUNT(*)`, `COUNT(col)`, `SUM`, `AVG`, `MIN` and `MAX`; projection `AS` aliases and `FROM` aliases; and the DML statements multi-row `INSERT`, multi-field `UPDATE` and `DELETE`.

Not supported

The translator raises `MongoSqlTranslationException` for any construct outside the supported grammar:

- `HAVING` — filter script-side, or write the pipeline directly as a JSON aggregation directive.
- `CASE WHEN` expressions.
- `COUNT(DISTINCT col)` — use `SELECT DISTINCT col FROM X` and count the rows in the calling script.
- `JOIN`, `UNION` and subqueries — MongoDB's `$lookup` is async-batch and does not fit row-by-row reads; use the aggregation pipeline directly.
- `UPSERT` / `MERGE` / `ON CONFLICT` — run an `UPDATE`, check `RowsAffected`, and `INSERT` script-side when it is zero.
- `INSERT ... SELECT` (cross-collection insert).
- Right-hand-side arithmetic in `UPDATE SET` (for example `SET counter = counter + 1`) — use the JSON directive `{"$inc": {"counter": 1}}`.
- Date and time functions (`NOW`, `INTERVAL`, `DATE_ADD`).
- String functions (`UPPER`, `LOWER`, `CONCAT`, `SUBSTRING`).

Write semantics

- The **Save** call on a Dataset Table dispatches a single MongoDB `BulkWrite`, which applies inserts, updates and deletes according to each row's state. Only the tracked columns are written; fields outside the tracked column list are preserved.
- Updated and deleted rows are matched on the document `_id`. A row whose `_id` is missing or null is **skipped silently** — it is not written and raises no error, so keep `_id` in the Dataset Table's column mapping whenever the table is used for updates or deletes.
- `BulkWrite` returns operation counts rather than rows, so the Dataset Table is not re-read from the server after a save.
- Authentication supports SCRAM-SHA-256, TLS, replica sets, and `mongodb+srv://` URIs.

Worked example

For an end-to-end Dataset configuration that reads, aggregates, and writes documents in a MongoDB collection (one Dataset DB, three Queries, one Dataset Table, three Script Tasks), see the dedicated example page: Datasets MongoDB Example.

The example uses the seed data described in the Installation section above (database `plant_data`, collection `readings`).

Device Protocol

When to use

Use the Device Protocol surface when you want each MongoDB document field addressed as an individual Tag with a Node-level connection. This is the classic SCADA Channel / Node / Point model.

Use UNS TagProvider instead (next section) when you want dynamic browse and tree-style registration.

Configure

Channel. Devices / Channels / New. Protocol = `MongoDB`. Interface is automatically **Custom** (no CommAPI).

Node. One Node per MongoDB server plus database. PrimaryStation is a delimited string with the shape:

```
ConnectionUri;Database;AuthSource;TlsEnabled
```

Example value:

```
mongodb://localhost:27017;plant_a;admin;false
```

Point. Address is a single-dot `collection.field` string. Example: `sensors.temperature`.

Read semantics

On each scan cycle, the driver runs one `find` per configured address, sorts by `_id` descending, limits to 1, and projects the requested field. The value is returned as a string on every successful read. Quality is 192 (Good) on hit, 0 (Bad) when the collection is empty or the field is absent.

Values round-trip as strings in the Device path. Numeric or boolean tags show their value as text. Use the UNS TagProvider surface (next section) when your integration needs the value native type preserved.

Write semantics - UPDATE, not append

`WriteTagValue` finds the most recent document in the collection and runs `$set` on that document `_id`. If the collection is empty, a fresh document is inserted with the field plus a `ts` UTC timestamp.

This is config-style or settings-style write behavior. Every write mutates the latest document. It does NOT append a new time-series sample. Use a Historian StorageLocation (later section) for append-style time-series writes.

Address limits

Only single-dot addresses are accepted in 10.1.5. `collection.field` works. `collection.field.subfield` is rejected. Nested BSON path support is planned for a later release.

Configuration Example

This example maps a single MongoDB document field to a Tag using the Channel / Node / Point model. Prerequisites: a running `mongod` with the `plant_data` database and the `sensors` collection from the Installation section above, plus a UNS Tag named **SensorTemp** of type **Double**.

1. In **Devices / Channels**, create a Channel:
    - Name: **MongoDB1**.
    - Protocol: `MongoDB`. The Interface is set to **Custom** automatically.
2. In **Devices / Nodes**, create a Node under that Channel:
    - Name: **Plant_A**.
    - Channel: **MongoDB1**.
    - PrimaryStation:

      ```
      mongodb://localhost:27017;plant_data;admin;false
      ```
3. In **Devices / Points**, create a Point bound to the tag **SensorTemp**:
    - TagName: **SensorTemp**.
    - Node: **Plant_A**.
    - Address: `sensors.temperature`.
    - AccessType: **ReadWrite**.
4. Run the solution. **SensorTemp** reads the latest `temperature` value from the most recent document in the `sensors` collection on each scan cycle.
5. From a Script Task, write the tag back: `@Tag.SensorTemp = 75.0;`. The driver runs `$set` on the latest `sensors` document. The collection's most recent document is mutated in place. This is config-style write behavior, not append. Use the Historian StorageLocation surface (later section) for append-style time-series writes.

UNS TagProvider

When to use

Use the UNS TagProvider surface when you want to browse a MongoDB instance from the Designer UNS tree, discover field names dynamically, and register fields as UNS tags.

Configure

UNS / TagProviders / New. Protocol = `MongoDB`. Station uses the same delimited format as the Device Protocol:

```
ConnectionUri;Database;AuthSource;TlsEnabled
```

Browse semantics

The browse view has two levels:

- **Level 0** (tree root for this TagProvider). Lists all collections in the configured database.
- **Level 1** (inside a collection). Samples the most recent document with `sort({_id:-1}).limit(1)` and lists that document top-level field names. The `_id` field is hidden from the browse view.

The browse is schema-less. If different documents in a collection carry different fields, the browse reflects only the sample document. Register the fields you need and define them explicitly on the UNS Tag when the schema varies.

Designer refresh does not mutate state. Browse is side-effect-free and safe to call on every tree expansion.

Register a tag

Select a field under a collection and use **Add to UNS**. The registered address is `collection.field`. The attribute type is `string` in 10.1.5 (all scalars). Typed registration is planned for a later release.

Read and write semantics

Identical to the Device Protocol section:

- Read pulls the latest document field value and tags quality 192 on hit, 0 on miss.
- Write runs `$set` on the latest document `_id`, or inserts a minimal document when the collection is empty.

Address limits

Single-dot `collection.field` only. The registration call rejects multi-dot addresses with the message `Multi-segment paths not supported in 10.1.5, use 'collection.field' with a single dot`.

Configuration Example

This example browses a MongoDB instance from the Designer UNS tree and registers a document field as a UNS tag. Prerequisites: a running `mongod` with the `plant_data` database and the `sensors` collection from the Installation section above.

1. In **UNS / TagProviders**, create a TagProvider:
    - Name: **MongoTags**.
    - Protocol: `MongoDB`.
    - ServiceType: **TagDiscovery**.
    - PrimaryStation:

      ```
      mongodb://localhost:27017;plant_data;admin;false
      ```

2. Open the UNS tree in Designer and expand **MongoTags**. The tree shows the collections in `plant_data` at level 0 (for example `readings` and `sensors`).
3. Expand **sensors**. The browse samples the most recent document and lists its top-level field names (for example `temperature`, `plant`, `ts`). The `_id` field is hidden.
4. Right-click the `temperature` field and choose **Add to UNS**. A tag `MongoTags/sensors/temperature` appears in the UNS Tags list with attribute type `string`.
5. Run the solution. `@Tag.MongoTags/sensors/temperature` reads the latest temperature value from the collection on each scan cycle.
6. Writing the tag from a Script Task runs `$set` on the latest `sensors` document, with the same UPDATE semantics as the Device Protocol surface.

Historian StorageLocation

When to use

Use this surface when you want FrameworX Historian to write tag samples into MongoDB as an append-style time series. This is the append behavior most users expect from a historian.

From FrameworX 10.1.6, a Historian table on a MongoDB Storage Location stores its samples. In 10.1.5, it stored none. A MongoDB TagProvider with read-only access refuses Historian samples, and the Historian table's **Last Error Message** reads `Write not allowed in Table`.

Prerequisites

A MongoDB UNS TagProvider row must already exist (previous section). The Historian StorageLocation references it by name.

Configure

Historian / Storage Locations / New. The Protocol combo includes MongoDB when the TagProvider row has `IsHistorian="true"` (the default for 10.1.5). The DataRepository field selects which MongoDB UnsTagProvider to use.

Storage model - Time Series Collections

Each FrameworX tag-collection maps to a MongoDB Time Series Collection (MongoDB 6.0+). The canonical document schema is:

```
{
  "t": "<UTC DateTime>",
  "m": { "tag": "<address>" },
  "v": "<value>",
  "q": "<quality ushort>"
}
```

The collection is created on first write if absent. Granularity is hard-coded to `seconds` in 10.1.5. This value is immutable on the MongoDB side after collection creation. Station-configurable granularity is planned for a later release.

Write semantics - BATCHED APPEND

`WriteHistorianDataEx` groups the incoming batch by collection name and calls `InsertMany` with `IsOrdered=false`. Every sample becomes a new document. Partial-batch failures are surfaced on a per-document basis. A bad document does not fail the whole batch.

Read semantics - raw samples only in 10.1.5

`GetSamples` does a range query (`$gte` and `$lte` on the `t` field) with ascending sort. The `interval` and `getSamplesMode` parameters are ignored in 10.1.5. Raw samples only.

Aggregation (`Average`, `Min`, `Max`, `Sum` with non-zero interval) is planned for a later release using MongoDB `$bucket` and `$bucketAuto` stages.

Known edge case - same collection as both TagProvider and StorageLocation

If a user configures the same collection as BOTH a UNS TagProvider browse target AND a Historian StorageLocation target, the Historian write path succeeds silently into a plain (non-Time-Series) collection but reads behave incorrectly. A runtime guard is planned for a later release. Until it lands, use distinct collection names for the two roles.

Configuration Example

This example appends historian samples for a UNS tag to a MongoDB Time Series Collection. Prerequisites: a UNS TagProvider **MongoTags** exists from the previous section and points at the same MongoDB instance, plus a UNS Tag **SensorTemp** of type **Double** being updated by a Device, Script, or external source.

1. In **Historian / Storage Locations**, create a StorageLocation:
   - Name: **MongoStore**.
   - DataRepository: `TagProvider.MongoTags`.
2. In **Historian / Tags**, bind **SensorTemp** to **MongoStore** with a sample interval (for example one second) and a deadband appropriate for the signal.
3. Run the solution. On the first sample, the connector creates a Time Series Collection in `plant_data` with the canonical schema documented in the Storage model section above (granularity hard-coded to `seconds` in 10.1.5). Subsequent samples are appended via `InsertMany` with `IsOrdered=false`.
4. Read history with `@Tag.SensorTemp.GetSamples(...)` from a Script or Trend control. The connector runs a range query on the `t` field with ascending sort and returns raw samples. Aggregation (`Average`, `Min`, `Max`, `Sum` with non-zero interval) is planned for a later release.

Known limitations and deferrals

The table below collects per-surface limits that apply to the 10.1.5 release.

Area	Limit	Workaround
Dataset SQL subset	`HAVING`, `CASE WHEN`, `COUNT(DISTINCT col)`, `JOIN` / `UNION` / subqueries, `UPSERT` / `MERGE`, `INSERT ... SELECT`, right-hand-side arithmetic in `UPDATE SET`, and date/time or string functions are not supported. These raise `MongoSqlTranslationException`.	Use an aggregation pipeline, or perform the step script-side.

Dataset SQL subset - collation	No collation is applied. Comparisons follow BSON default collation and identifiers are case-sensitive.	Normalize case on ingest, or use the aggregation pipeline with an explicit `$regexMatch` for case-insensitive matching.
Device and UNS address format	Single-dot `collection.field` only in 10.1.5.	Flatten nested documents on ingest, or wait for a later release.
Device and UNS values	Round-trip as strings.	Use typed tag bindings on the UNS side for numerical dashboards.
Historian granularity	Hard-coded to `seconds`.	Choose the most precise granularity at collection creation. Recreate the collection to change.
Historian aggregation	Raw samples only. Average, Min, Max, Sum with an interval are not supported.	Request raw samples and aggregate in FX Trend or Script.
Historian connection pool	One `ConnectionInfo` per call in `GetSamples` (no cache).	Scope time ranges tightly on dashboards.
UNS and Historian on the same collection	No runtime guard against mixing.	Use distinct collection names for the two roles.

Driver Revision History

MongoDB Revision History	
Version	**Notes**
1.0.0.1	Historian data is now stored in MongoDB on a standard installation (the connector could not locate its libraries); station and address editors provide proper input fields with defaults; reads use the document's own quality and timestamp fields when present, and browsing returns complete element information.

In this section...