MathOps.Dev Generalized Database Layer

The MathOps system provides a generalized interface for data access that can use a variety of back-end products or persistence strategies, presented to applications as a **generalized database layer** with a flexible **API**.

This is based on a set of generalized classes that represent:

* Logical ***tables***, organized into ***schemas***, that define ***fields*** and store ***rows***. Each schema may provide multiple ***contexts*** (production, development, test, etc.)
* Generalized ***query criteria*** that can be used for queries or to select rows in a table for updates, deletes, or counting
* Generalized ***updated data*** objects that can store new field values for updates.
* A generalization of a ***connection*** to a server product, and support for configuration of multiple products, databases, user logins etc., with pluggable implementations for particular back-ends if needed to interface with legacy data.

The core classes are provided within the **dev.mathops.persistence** package.

It is worth noting some features that are NOT provided by this system:

* Foreign keys and referential integrity between tables.
* Joins, or queries that access multiple tables.
* Transactions. Every operation that changes data is committed at the time it completes or rolled back if it fails.

# Tables, Schemas, Fields, and Rows

A ***table*** is represented by an instance of a concrete subclass of the **dev.mathops.persistence.Table** class. Such subclasses should be immutable (thread-safe) and are typically singletons with private constructor and static instance.

A ***table*** exists within a ***schema*** (a named collection of tables, typically implemented in a single database product or installation). The table stores the unique name of its schema, its unique table name, and a list of the ***fields*** that rows in that table may contain.

Each ***field*** defined within a table is represented by an instance of the (final) **dev.mathops.persistence.Field** class, which is an immutable (thread-safe) object that stores the field name (unique within the table), the field's data type (see below), the field's role in the table (see below), and zero or more ***constraints*** that values for this field must obey. **Field** objects are suitable for use as map keys, and implement **Comparable<Field>** so they can be used in contexts that require a well-defined order (such as keys in a **TreeMap**).

A ***constraint*** on a field of type **T** is an instance of one of a set of pre-defined concrete subclasses of **dev.mathops.persistence.constraint.AbstractFieldConstraint<T>**, or possibly a custom subclass provided by an application.

The data types supported by fields, along with the corresponding Java object type and the pre-defined types of constraint that a field may define for each type are listed below:

* String (java.lang.String)
  + **StringEnumeratedConstraint**, which defines a fixed list of allowed String values.
  + **StringLengthConstraint**, which defines a minimum and maximum allowed length.
* Boolean (java.lang.Boolean)
* Byte (java.lang.Byte)
  + **ByteRangeConstraint**, which defines a minimum and maximum allowed byte value.
* Integer (java.lang.Integer)
  + **IntegerRangeConstraint**, which defines a minimum and maximum allowed int value.
* Long (java.lang.Long)
  + **LongRangeConstraint**, which defines a minimum and maximum allowed long value.
* Float (java.lang.Float)
  + **FloatRangeConstraint**, which defines a minimum and maximum allowed float value and specified whether NaN or infinite values are allowed.
* Double (java.lang.Double)
  + **DoubleRangeConstraint**, which defines a minimum and maximum allowed double value and specified whether NaN or infinite values are allowed.
* Blob (java.sql.Blob)
* LocalDate (java.time.LocalDate)
* LocalTime (java.time.LocalTime)
* LocalDateTime (java.time.LocalDateTime)

The roles within their containing table that a field may be assigned include:

* Partition Key
  + The field participates in the primary key (the tuple of all fields that participate in the primary key together must have a unique value for each row in the table)
  + The field can be used to partition data across multiple servers. Fields used as partition keys should be chosen so that the majority of queries will select only rows with the same value for the partition key.
* Clustering Key
  + The field participates in the primary key
  + The field can be used to cluster data within a single partition for faster selection of data by queries.
* Not-null
  + The field does NOT participate in the primary key.
  + The field may not have a NULL value – it must have a specified value in each row. Note that an empty string is not considered a NULL value.
* Nullable
  + The field does NOT participate in the primary key.
  + The field may have any value, including NULL.

Typically, concrete subclasses of the **Table** class will define their fields, with all required constraints, as class-static instances of **Field** with names that are upper-case underscore-separated versions of the field's name. The field's name itself is often required to map to an actual field name in a database product, so it should limit itself to lowercase ASCII letters and digits and underscores. For example, a field with the name "user\_id" would be defined by a class-static instance of Field named "USER\_ID".

A ***row*** is represented by an immutable instance of the **dev.mathops.persistence.Row** class, which contains an immutable array of values for all fields defined in a table. A row carries a reference to the table to which it belongs so its fields can be interpreted at runtime. Each field value in a row has either the Java object type that corresponds to its field, or is null. Field values are guaranteed to satisfy the constraints of the corresponding field definition.

A concrete subclass of **Table** should provide static utility methods to retrieve field values from rows with their proper type. These methods should take a single **Row** argument, and should have a method name that is the CamelCase rendition of the field name, plus "Of".

For example, if a table defines a String field named "last\_name", it should provide a static method with this signature:  
 **static String lastNameOf(Row row)**

This method should test that the row's owning table matches the single instance of the **Table** that owns the field, but it does not need to test the data type of the field value, since it will have been validated on construction.

# Selection Criteria

When performing queries, updates, deletes, and counts, applications can provide ***selection criteria*** to determine which rows from a table are considered. This is provided to the database layer in the form of an immutable instance of the final **dev.mathops.persistence.SelectionCriteria** class.

This class stores a reference to the table being queried, as well as an array of zero or more ***field criterion*** objects.

If no field criterion objects are provided, every row in the table is to be considered. If one or more criterion objects are provided, only rows that satisfy ALL supplied criteria will be considred.

***Field criterion*** objects are concreate subclasses of the **dev.mathops.persistence.criteria.AbstractFieldCriterion** class, and are either one of a pre-defined set of criteria, or a custom criteria provided by the application. Every implementation of the database layer for a particular database engine or product will have to interpret these slection criteria objects and convert them into, for example, SQL "where" clauses, or other forms of criteria as appropriate to the product.

The set of pre-defined field criteria for each allowed field type are:

* String
  + **StringFieldCriterion**, which specifies a match type with associated data:  
    IS\_NULL  
    IS\_NOT\_NULL  
    EXACT\_IN (with a list of matching String values)  
    EXACT\_NOT\_IN (with a list of excluded String values)  
    CASE\_INSENSITIVE\_IN (with a list of matching String values)  
    CASE\_INSENSITIVE\_NOT\_IN (with a list of excluded String values)  
    EXACT\_STARTS\_WITH (with a list of matching String prefixes)  
    EXACT\_NOT\_STARTS\_WITH (with a list of excluded String prefixes)  
    CASE\_INSENSITIVE\_STARTS\_WITH (with a list of matching String prefixes)  
    CASE\_INSENSITIVE\_NOT\_STARTS\_WITH (with a list of excluded String prefixes)
* Boolean
  + **BooleanFieldCriterion**, which specifies a match type:  
    IS\_NULL  
    IS\_NOT\_NULL  
    IS\_TRUE  
    IS\_FALSE
* Byte
  + **ByteFieldCriterion**, which specifies a match type with associated data:  
    IS\_NULL  
    IS\_NOT\_NULL  
    EXACT\_IN (with a list of matching Byte values)  
    EXACT\_NOT\_IN (with a list of excluded Byte values)  
    GREATER\_THAN (with a single Byte value)

GREATER\_THAN\_OR\_EQUAL (with a single Byte value)  
LESS\_THAN (with a single Byte value)  
LESS\_THAN\_OR\_EQUAL (with a single Byte value)  
BETWEEN\_EXCLUDE\_BOUNDS (with an ordered pair of Byte values)

BETWEEN\_EXCLUDE\_LOWER\_BOUND (with an ordered pair of Byte values)

BETWEEN\_EXCLUDE\_UPPER\_BOUND (with an ordered pair of Byte values)

BETWEEN\_INCLUDE\_BOUNDS (with an ordered pair of Byte values)

* Integer
  + **IntegerFieldCriterion**, which specifies a match type with associated data (as the match types for Byte, but with Integer values rather than Byte values)
* Long
  + **LongFieldCriterion**, which specifies a match type with associated data (as the match types for Byte, but with Long values rather than Byte values)
* Float
  + **FloatFieldCriterion**, which specifies a match type with associated data (as the match types for Byte, but with Float values rather than Byte values)
* Double
  + **DoubleFieldCriterion**, which specifies a match type with associated data (as the match types for Byte, but with Double values rather than Byte values)
* LocalDate
  + **LocalDateFieldCriterion**, which specifies a match type with associated data:  
    IS\_NULL  
    IS\_NOT\_NULL  
    EXACT\_IN (with a list of matching LocalDate values)  
    EXACT\_NOT\_IN (with a list of excluded LocalDate values)  
    GREATER\_THAN (with a single LocalDate value)

GREATER\_THAN\_OR\_EQUAL (with a single LocalDate value)  
LESS\_THAN (with a single LocalDate value)  
LESS\_THAN\_OR\_EQUAL (with a single LocalDate value)  
BETWEEN\_EXCLUDE\_BOUNDS (with an ordered pair of LocalDate values)

BETWEEN\_EXCLUDE\_LOWER\_BOUND (with an ordered pair of LocalDate values)

BETWEEN\_EXCLUDE\_UPPER\_BOUND (with an ordered pair of LocalDate values)

BETWEEN\_INCLUDE\_BOUNDS (with an ordered pair of LocalDate values)

* LocalTime
  + **LocalTimeFieldCriterion**, which specifies a match type with associated data (as the match types for LocalDate, but with LocalTime values rather than LocalDate values)
* LocalDateTime
  + **LocalDateTimeFieldCriterion**, which specifies a match type with associated data (as the match types for LocalDate, but with LocalTime values rather than LocalDateTime values)

# Updated Values

When performing updates, applications need to supply new values for a subset of the fields in a table, but the **Row** class is unsuitable for this task since every row must have values that match its constraints for every field.

Therefore, an immutable (final) **dev.mathops.persistence.UpdatedValues** class is defined as a container for new values for an update operation.

This class contains a reference to the table being updated, and a list of new field values, some of which may be null to indicate the corresponding field is not to be updated. A special **dev.mathops.persistence.NullValue** object is provided to allow applications to specify that a field's value is to be set to NULL.

# Implementation Layer

For a given back-end implementation (a database with a set of tables that will get mapped to the database objects defined here), there will be an implementation class that can dispatch requests to either generalized code to handle tables defined in standard ways, or to custom code that translates requests into legacy or database-specific operations.

This layer will connect directly to the database engine, and will be used when an API call is processed. Client code is free to use the implementation layer direcly rather than use the API to avoid the overhead.

**Data Configuration**

A system may have many implementations active at once. A **data configuration** objects selects implementations and routes operations to the correct implementation.

This object selects a single implementation for every defined **schema** and **context**. When the client makes a call that references a schema and context, the corresponding implementation is used.

The data layer itself stores the set of data configurations available. These are guaranteed to support the complete set of schemas and contexts defined within the data layer in order to be advertised as available. Configurations may be named, and client code can use those names to look up and use the configurations (subject to permissions granted and the authorization token used in the request).

**Implementation Object**

Every implementation has a single object that provides the implementaiton interface. This object implements the **dev.mathops.persistence. IImplementation** interface. It may internally process all requests directly (when the underlying database has a very regular structure), or it may delegate requests to objects that implement custom code appropriate to the underlying database.

# Service API

The database layer provides a service-oriented REST API that supports secure queries and management. The API is delivered through the HTTP protocol over TLS connections.

API requests include an an auhorization token that encodes the user's permissions. This token is generated as part of an authentication process, which can take place through the API, or through a web-based front-end to the service.

The API uses a binary format (which may be compressed) rather than XML or JSON, for efficiency. Requests can use a transfer encoding of "chunked" or "gzip" to send data to endpoints, and responses will use one of these two encodings when data is returned.

**Authorization Token Encoding**

A 128-bit binary value. The high-order 64 bits is a random ID generated by the authentication service. The low-order 64 bits has, as its high-order 32-bits, the role of the authenticating user, and in its low-order 32-bits, the effective role ID for the transaction. Users may execute transactions with any role for which their primary role grants access. This can be used for testing role permissions, or for executing operations in under a "safer" role with fewer unnecessary permissions. This also allows an "aggregate" role to be defined that grants permissions to a number of other roles without having to grant the aggregate role a superset of permissions of the other roles.

This token is transmitted ONLY over TLS-secured connections.

**Context Encoding**

The context ID as an Integer (Tinyint, Byte, Short, or Integer)  
The context name as a a String (ASCII1, ASCII2, String1, String2)

**Schema Encoding**

The schema ID as an Integer (Tinyint, Byte, Short, or Integer)  
The schema name as a a String (ASCII1, ASCII2, String1, String2)  
The number of contexts (**N**) under which this schema may be accessed.  
**N** repetitions of:   
 The context ID as an Integer (Tinyint, Byte, Short, or Integer)

**Table Encoding** (used when a list of tables and field definitions is to be returned)

The owning schema ID as an Integer (Tinyint, Byte, Short, or Integer)  
The table ID as an Integer (Tinyint, Byte, Short, or Integer)  
The table name as a a String (ASCII1, ASCII2, String1, String2)  
The number of fields (**N**) as an Integer (Tinyint, Byte, or Short)  
**N** repetitions of:   
 An enumerated value (Tinyint) indicating the field type  
 An enumerated value (Tinyint) indicating the field's role  
 The number (**M**) of constraints associated with the field, as an integer (Tinyint, Byte, or Short)  
 **M** repeitions of constraint definitions:  
 If type is STRING\_ENUMERATED, the number (P) of enumerated values as an Integer (Tinyint, Byte, or  
 Short) followed by P Strings (ASCII1, ASCII2, ASCII4, String1, String2, or String4)   
 If type is STRING\_LENGTH, the minimum length as an Integer (Tinyint, Byte, Short, or Integer) followed  
 by the maximum length as an integer (Tinyint, Byte, Short, or Integer)   
 If type is BYTE\_RANGE, the minimum value as a Byte followed by the maximum value as a Byte.   
 If type is INTEGER\_RANGE, the minimum value as an integer (Tinyint, Byte, Short, or Integer) followed by  
 the maximum value as an integer (Tinyint, Byte, Short, or Integer).   
 If type is LONG\_RANGE, the minimum value as a long integer (Tinyint, Byte, Short, or Integer, or Long)  
 followed by the maximum value as an integer (Tinyint, Byte, Short, Integer, or Long).   
 If type is FLOAT\_RANGE, the minimum value as a Float followed by the maximum value as a Float.   
 If type is DOUBLE\_RANGE, the minimum value as a Double followed by the maximum value as a Double.

**Binary Encodings**

All numeric fields are encoded in big-endian byte ordering (most significant byte first).

Type: An 8-bit type field that predeces any field.   
0x01 = ASCII1 0x02 = ASCII2 0x03 = ASCII4  
0x04 = String1 0x05 = String2 0x06 = String4   
0x07 = Boolean 0x08 = Byte 0x09 = Short  
0x0A = Integer 0x0B = Long 0x0C = Float  
0x0D = Double 0x0E = LocalDate 0x0F = LocalTime  
0x10 = LocalDateTime 0x11 = BLOB1 0x12 = BLOB2  
0x13 = BLOB4 0x30-0xFF = Tinyint (unsigned integer from 0x00 to 0xCF)

ASCII1: An 8-bit unsigned length (N) followed by (N) ASCII characters

ASCII2: A 16-bit unsigned length (N) followed by (N) ASCII characters

ASCII4: A 31-bit unsigned length (N) followed by (N) ASCII characters

String1: An 8-bit unsigned length (N) followed by (N) UTF-16 code points

String2: A 16-bit unsigned length (N) followed by (N) UTF-16 code points

String4: A 31-bit unsigned length (N) followed by (N) UTF-16 code points

Boolean: An 8-bit value 0x01 or 0x00

Byte: An 8-bit signed integer or enumerated value

Short: A 16-bit signed integer or enumerated value

Integer: A 32-bit signed integer or enumerated value

Long: A 64-bit signed integer

Float: A 32-bit floating-point number

Double: A 64-bit floating point number

LocalDate: A 32-bit number whose low-order 5 bits are the unsigned day of the month (1-31), next highest 4 bits are the unsigned month (1 to 12), and remaining high-order 23 bits are the signed year. Dates whose year falls outside this range cannot be transmitted.

LocalTime: A 32-bit number whose low-order 10 bits are the unsigned millisecond (0 to 999), next highest 6 bits are the unsigned second (0 to 59), next highest 6 bits are the unsigned minute (0 to 59), and next highest 5 bits are the unsigned hour (0 to 23). The remaining 5 bits should be zero.

LocalDateTime: A 32-bit LocalDate (as above) followed by a 32-bit LocalTime (as above).

BLOB1: An 8-bit unsigned length (N) followed by (N) bytes

BLOB2: A 16-bit unsigned length (N) followed by (N) bytes

BLOB4: A 32-bit signed length (N) followed by (N) bytes

**Selection Criteria Encoding** (NOTE: table name is already defined when this object is encoded)

An enumerated type (Tinyint, Byte, Short, or Integer) to indicate the block is Selection Criteria  
An integer (Tinyint, Byte, Short, or Integer) encoding the number (N) of field criteria that follows.  
N repetitions of:  
 An enumerated value (Tinyint, Byte, Short, or Integer) indicating the constraint type  
The field name String (ASCII1, ASCII2, ASCII4, String1, String2, or String4)  
 An enumerated value (Tinyint, Byte, Short, or Integer) indicating match type  
 If type is STRING: An integer (M) with the number of match strings  
 M repetitions of a String (ASCII1, ASCII2, ASCII4, String1, String2, or String4)   
 If type is BOOLEAN, no additional fields   
 If type is BYTE  
 An integer (M) with the number of match bytes  
 M repetitions of a Byte  
 If type is INTEGER  
 An integer (M) with the number of match integers  
 M repetitions of an Integer (Tinyint, Byte, Short, or Integer)  
 If type is LONG  
 An integer (M) with the number of match integers  
 M repetitions of an Integer (Tinyint, Byte, Short, Integer, or Long)  
 If type is FLOAT  
 An integer (M) with the number of match floats  
 M repetitions of an Float  
 If type is DOUBLE  
 An integer (M) with the number of match doubles  
 M repetitions of Double  
 If type is LCOAL\_DATE  
 An integer (M) with the number of match local dates  
 M repetitions of LocalDate  
 If type is LCOAL\_TIME  
 An integer (M) with the number of match local times  
 M repetitions of LocalTime  
 If type is LCOAL\_DATE\_TIME  
 An integer (M) with the number of match local date/times  
 M repetitions of LocalDateTime

**Row Encoding** (NOTE: schema and table name are already defined when this object is encoded)

An enumerated type (Tinyint, Byte, Short, or Integer) to indicate the block is a Record   
An integer (Tinyint, Byte, Short, or Integer) encoding the number (N) of field values that follow.  
N repetitions of:  
 The integer field index (Tinyint, Byte, Short, or Integer)  
 An enumerated value (Tinyint, Byte, Short, or Integer) indicating the field type  
 If type is STRING, a String (ASCII1, ASCII2, ASCII4, String1, String2, or String4)   
 If type is BOOLEAN, a Boolean  
 If type is BYTE, a Byte   
 If type is INTEGER, an Integer (Tinyint, Byte, Short, or Integer)  
 If type is LONG, a long integer (Tinyint, Byte, Short, Integer, or Long)  
 If type is FLOAT, a Float   
 If type is DOUBLE, a Double   
 If type is LOCAL\_DATE, a LocalDate   
 If type is LOCAL\_TIME, a LocalTime   
 If type is LOCAL\_DATE\_TIME, a LocalDateTime  
(fields with null values are excluded)

**Updated Values Encoding** (NOTE: schema and table name are already defined when this object is encoded)

An enumerated type (Tinyint, Byte, Short, or Integer) to indicate the block is New Values  
An integer (Tinyint, Byte, Short, or Integer) encoding the number (N) of field values that follow.  
N repetitions of:  
 The integer field index (Tinyint, Byte, Short, or Integer)  
 An enumerated value (Tinyint, Byte, Short, or Integer) indicating the field type  
 If type is NULL, no further data is needed  
 If type is STRING, a String (ASCII1, ASCII2, ASCII4, String1, String2, or String4)  
 If type is BOOLEAN, a Boolean  
 If type is BYTE, a Byte   
 If type is INTEGER, an Integer (Tinyint, Byte, Short, or Integer)  
 If type is LONG, a long integer (Tinyint, Byte, Short, Integer, or Long)  
 If type is FLOAT, a Float   
 If type is DOUBLE, a Double   
 If type is LOCAL\_DATE, a LocalDate   
 If type is LOCAL\_TIME, a LocalTime   
 If type is LOCAL\_DATE\_TIME, a LocalDateTime

In the API endpoints documented below, the endpoint name is the portion of the HTTP request path that follows that part that identifies the host and service. Request parameters are provided in the request body in a binary format.

**GET all\_contexts**

Retrieves the set of defined contexts.  
*Request body:*  
 16-byte authorization token  
*Response body:* Enumerated result code {SUCCESS | FAILURE} (Tinyint, Byte or Short)   
 On SUCCESS:  
 Integer number of contexts (**N**) (Tinyint, Byte, Short, or Integer)  
 **N** repetitions of:  
 **Context Encoding** (see above)  
 On FAILURE:  
 Enumerated failure code (Tinyint, Byte, or Short)  
 Error message (ASCII1, ASCII2, ASCII4, String1, String2, or String4)

**GET all\_schemas**

Retrieves the set of defined schemas.  
*Request body:*  
 16-byte authorization token  
*Response body:* Enumerated result code {SUCCESS | FAILURE} (Tinyint, Byte or Short)   
 On SUCCESS:  
 Integer number of schemas (**N**) (Tinyint, Byte, Short, or Integer)  
 **N** repetitions of:  
 **Schema Encoding** (see above)  
 On FAILURE:  
 Enumerated failure code (Tinyint, Byte, or Short)  
 Error message (ASCII1, ASCII2, ASCII4, String1, String2, or String4)

**GET all\_tables**

Retrieves the set of defined tables and their fields and constraints. If a schema ID is provided, only the tables in that schema are returned.  
*Request body:*  
 16-byte authorization token  
 Flags (Short) – 0x01 = Include row count  
 [OPTIONAL] Schema ID (Tinyint, Byte, Short, or Integer)  
*Response body:* Enumerated result code {SUCCESS | FAILURE} (Tinyint, Byte or Short)   
 Flags (Short) – copied from request  
 On SUCCESS:  
 Integer number of tables (**N**) (Tinyint, Byte, Short, or Integer)  
 **N** repetitions of:  
 **Table Encoding** (see above)  
 If Flags indicates row count included:  
 Integer number of rows in the table (Tinyint, Byte, Short, Integer, or Long)  
 On FAILURE:  
 Enumerated failure code (Tinyint, Byte, or Short)  
 Error message (ASCII1, ASCII2, ASCII4, String1, String2, or String4)

**GET table**Retrieves the fields and constraints of a single table.  
*Request body:*  
 16-byte authorization token  
 Flags (Short) – 0x01 = Include row count  
 The schema ID as an integer (Tinyint, Byte, Short, or Integer)  
 The table ID as an integer (Tinyint, Byte, Short, or Integer)  
*Response body:* Enumerated result code {SUCCESS | FAILURE} (Tinyint, Byte or Short)  
 Flags (Short) – copied from request  
 On SUCCESS:  
 **Table Encoding** (see above)  
 If Flags indicates row count included:  
 Integer number of rows (Tinyint, Byte, Short, Integer, or Long)  
 On FAILURE:  
 Enumerated failure code (Tinyint, Byte, or Short)  
 Error message (ASCII1, ASCII2, ASCII4, String1, String2, or String4)

**GET count**

Retrieves the number of rows in the requested table that match criteria.  
*Request body:*  
 16-byte authorization token  
 The schema ID as an integer (Tinyint, Byte, Short, or Integer)  
 The context ID as an integer (Tinyint, Byte, Short, or Integer)  
 The table ID as an integer (Tinyint, Byte, Short, or Integer)  
 [OPTIONAL] **Selection Criteria** (see above – if omitted, all rows match)  
*Response body:* Enumerated result code {SUCCESS | FAILURE} (Tinyint, Byte or Short)   
 On SUCCESS:  
 Integer number of rows (Tinyint, Byte, Short, Integer, or Long)  
 On FAILURE:  
 Enumerated failure code (Tinyint, Byte, or Short)  
 Error message (ASCII1, ASCII2, ASCII4, String1, String2, or String4)

**GET query**

Retrieves all rows in a specified table that match a set of match criteria.  
*Request body:*  
 16-byte authorization token  
 The schema ID as an integer (Tinyint, Byte, Short, or Integer)  
 The context ID as an integer (Tinyint, Byte, Short, or Integer)  
 The table ID as an integer (Tinyint, Byte, Short, or Integer)  
 [OPTIONAL] **Selection Criteria** (see above – if omitted, all records match)  
*Response body:* Enumerated result code {SUCCESS | FAILURE} (Tinyint, Byte or Short)   
 On SUCCESS:  
 Integer number of rows (**N**) (Tinyint, Byte, Short, Integer, or Long)  
 **N** repetitions of **Row Encoding** (see above)  
 On FAILURE:  
 Enumerated failure code (Tinyint, Byte, or Short)  
 Error message (ASCII1, ASCII2, ASCII4, String1, String2, or String4)

**POST insert**

Inserts one or more rows into a specified table. All rows will be inserted if successful; none will be inserted on failure.  
*Request body:*  
 16-byte authorization token  
 The schema ID as an integer (Tinyint, Byte, Short, or Integer)  
 The context ID as an integer (Tinyint, Byte, Short, or Integer)  
 The table ID as an integer (Tinyint, Byte, Short, or Integer)  
 Integer number of rows (**N**) (Tinyint, Byte, Short, or Integer)  
 **N** repetitions of **Row Encoding** (see above)  
*Response body:* Enumerated result code {SUCCESS | FAILURE} (Tinyint, Byte or Short)  
 On SUCCESS:  
 Integer number of rows inserted (Tinyint, Byte, Short or, Integer)  
 On FAILURE:  
 Enumerated failure code (Tinyint, Byte, or Short)  
 Error message (ASCII1, ASCII2, ASCII4, String1, String2, or String4)

**POST insert\_multi**

Inserts one or more rows into each of a set of specified tables. All rows will be inserted if  
successful; none will be inserted on failure.  
*Request body:*  
 16-byte authorization token  
 Integer number of tables (**M**) (Tinyint, Byte, Short, or Integer)  
 **M** repetitions of:  
 The schema ID as an integer (Tinyint, Byte, Short, or Integer)  
 The context ID as an integer (Tinyint, Byte, Short, or Integer)  
 The table ID as an integer (Tinyint, Byte, Short, or Integer)  
 Integer number of records (**N**) (Tinyint, Byte, Short, Integer)  
 **N** repetitions of **Row Encoding** (see above)  
*Response body:* Enumerated result code {SUCCESS | FAILURE} (Tinyint, Byte or Short)  
 On SUCCESS:  
 Integer number of rows inserted (Tinyint, Byte, Short, Integer)  
 On FAILURE:  
 Enumerated failure code (Tinyint, Byte, or Short)  
 Error message (ASCII1, ASCII2, ASCII4, String1, String2, or String4)

**POST delete**

Deletes all rows matching a set of selection criteria.  
*Request body:*  
 16-byte authorization token  
 The schema ID as an integer (Tinyint, Byte, Short, or Integer)  
 The context ID as an integer (Tinyint, Byte, Short, or Integer)  
 The table ID as an integer (Tinyint, Byte, Short, or Integer)  
 [OPTIONAL] **Selection Criteria Encoding** (see above – if omitted, all records match)  
*Response body:* Enumerated result code {SUCCESS | FAILURE} (Tinyint, Byte or Short)   
 On SUCCESS:  
 Integer number of rows deleted (Tinyint, Byte, Short, Integer, or Long)  
 On FAILURE:  
 Enumerated failure code (Tinyint, Byte, or Short)  
 Error message (ASCII1, ASCII2, ASCII4, String1, String2, or String4)

**POST update**

Updates all rows matching a set of selection criteria to new values.  
*Request body:*  
 16-byte authorization token  
 The schema ID as an integer (Tinyint, Byte, Short, or Integer)  
 The context ID as an integer (Tinyint, Byte, Short, or Integer)  
 The table ID as an integer (Tinyint, Byte, Short, or Integer)  
 [OPTIONAL] **Selection Criteria Encoding** (see above – if omitted, all records match)  
 **Updated Values Encoding** (see above)  
*Response body:* Enumerated result code {SUCCESS | FAILURE} (Tinyint, Byte or Short)   
 On SUCCESS:  
 Integer number of rows updated (Tinyint, Byte, Short, Integer, or Long)  
 On FAILURE:  
 Enumerated failure code (Tinyint, Byte, or Short)  
 Error message (ASCII1, ASCII2, ASCII4, String1, String2, or String4)

Question: Is a "delete\_multi" needed that deletes from multiple tables, where all will be deleted on success, or none are deleted on failure?

Question: Is an "update\_multi" needed that updates multiple tables, where all will be updated on success, or none are updated on failure?

**Authentication**

Authentication uses the SCRAM-SHA-256 protocol as defined in RFC 5802 and RFC 7677 over a TLS-secured connection.

The client begins by sending a **client\_first** message, to which the server responds with a **server\_first** message. The client then sends a **client\_final** message, to which the server responds with a **server\_final** message.

**POST authorization**