Doradus Logging Database

1. Introduction

This document describes Doradus Logging databases. In Doradus, an *application* is a named database with its own schema and data. Every schema selects a specific *storage service* to manage its data. When a schema selects the Doradus Logging storage service, the deployed instance is called a Doradus Logging database. The Logging storage service is optimized to manage immutable, time-series data such as log records and events. This document describes the unique features of the Logging service, including its data model, query language, and REST API commands.

This document is organized into the following sections:

- Logging Database Overview: An overview of the Logging database including its unique features and the type of applications it is best suited for.
- Logging Data Model: Describes the data model concepts use by Doradus Logging.
- Doradus Query Language (DQL): A detailed description of DQL specific to Doradus Logging.
- REST Commands: Describes the REST commands supported by the Logging service for application management, updating data, and queries.

This documentation is relative to the v2.4 release, which is the first release of the Doradus Logging service.

2. Logging Database Overview

This section describes usage and unique features of the Doradus Logging service.

2.1 Logging Service Features

Doradus uses a NoSQL database such as Cassandra for persistence. Each *storage service* organizes data into rows and columns using techniques beneficial to specific application types. The unique features of the Logging service are summarized below:

- **Designed for immutable time-series data**: The Logging service is designed for time-stamped objects that are not modified once added. Special storage techniques are used that leverage each object's timestamp and immutability, yielding fast loading, fast time-based queries, and dense storage.
- **Irregular data**: Logging application objects can be highly irregular, using different fields for each object. This is useful, for example, when log records have a variable format or when logs from multiple user applications are stored in the same table.
- **Space usage**: The Logging service "hyper compresses" data using special storage and compression techniques. For example, a batch of 64,000 objects that requires 10MB uncompressed and 1MB using GZIP compression requires only 250KB when stored in the Logging service.
- **Fast loading**: The Logging service can load data up to 500K objects per second on a single node.
- **Queries**: Object and aggregate queries can scan millions of objects per second without indexes. Query processing is designed to make time-constrained queries especially efficient. Even with irregular objects, all fields are queryable.
- **Live data**: All data can be queried as soon as it is stored. Multiple applications can store objects with overlapping and out-of-sequence timestamps simultaneously.
- **Data aging**: A Logging application schema can define automatic data aging, causing expired objects to be deleted automatically.

Because of its focus, the Logging service imposes restrictions compared to other Doradus storage services. In addition to requiring immutable, time-stamped objects, the Logging service has these restrictions:

- **No object IDs**: Logging service objects do not require nor use object IDs.
- **SV scalar fields only**: Link fields are not supported; each consists of single-valued (SV) scalar fields only.
- **Text field semantics**: Except for the required `Timestamp` field, which is a timestamp value, queries treat all fields like text fields even if they contain data that is numeric, Boolean, etc. Queries that use numeric literals will work in some cases such as the clause `EventID IN [500 TO 600]`. However, the clause `EventID IN [500 TO 1000]` will not work as expected since `EventID` and the values `500` and `1000` are treated as text strings.

- **Metric functions**: In aggregate queries, the Logging service currently only supports the metric function `COUNT(*)`. Similarly, metric *expressions* (e.g., `COUNT(*)+10`) are not supported.
- **Single-level grouping**: In aggregate queries, only a single grouping expression can be provided. Also, some grouping functions such as `BATCH` and `SETS` are not yet supported.

Most of these restrictions will be lifted as the Logging service evolves.

2.2 Starting the Logging Service

The Doradus server enables the Logging service when it is started with the following option in the `doradus.yaml` file:

```
storage_services:
    - com.dell.doradus.logservice.LoggingService
```

If the `LoggingService` is the first storage service listed, it becomes the default for new applications. Otherwise, new applications must use the `StorageService` option to select the `LoggingService`. For example (in JSON):

```
{"MyLogs": {
    "options": {"StorageService": "LoggingService"},
    ...
}
```

Details of installing and configuring the Doradus server are covered in the **Doradus Administration** document, but here's a review of the ways Doradus can be used:

- **Standalone**: The Doradus server can be started as a standalone application using the provided Jetty web server to serve the REST API. When Doradus is built using Maven, an example command line from the directory `{doradus_home}/doradus-jetty` is:

  ```
  java -cp ./target/classes:./target/dependency/* com.dell.doradus.core.DoradusServer
  ```

  Command line arguments can be added to such as "`-restport 5711`" to change the REST API port to 5711.

- **Tomcat**: Doradus can use Apache Tomcat to host its REST API. The `doradus-tomcat` folder provides a README file with instructions.
- **Windows service**: Doradus can be started as a Windows service by using the procrun package from Apache Commons. See the Doradus Administration document for details.
- **Embedded app**: Doradus can be embedded in another JVM process. This is especially useful for bulk load or other applications where performance is critical. As an embedded application, Doradus is started by calling the following method:

  ```
  com.dell.doradus.core.DoradusServer.startEmbedded(String[] args, String[] services)
  ```

where:

- `args` is the same arguments parameter passed to `main()` when started as a console application. For example `{"-restport", "5711"}` sets the REST port to 5711.
- `services` is the list of Doradus services to initialize. Each string must be the full package name of a service. Required services are automatically included. An embedded application must include at least one storage service. Other services should be included when the corresponding functionality is needed. See the comments in doradus.yaml for a list of the current services.

3. Logging Data Model

Doradus Logging uses a simplified version of the core Doradus data model. This section describes the requirements and options for creating a Logging application and the nature of objects.

3.1 Applications

An application is a named schema hosted in a Doradus cluster. An application's name is a unique identifier, which must begin with a letter and contain only letters, digits, or underscores. An application's data is stored in *tables*, which are isolated from other applications. Example application names are `MyLogs` and `Magellan_1`.

Each application is defined in a *schema*. An example schema for declaring a Logging application called `LogDepot` is shown below in XML:

```
<application name="LogDepot">
    <options>
        <option name="StorageService">LoggingService</option>
    </options>
    <tables>
        <table name="AppLogs"/>
    /tables>
</application>
```

The same schema is shown below in JSON:

```
{"LogDepot": {
    "options": {
        "StorageService": "LoggingService"
    },
    "tables": {
        "AppLogs": {}
    }
}}
```

The application is created with the REST command `POST /_applications` using the appropriate `Content-Type`. (See the **REST Commands** section for more details.) This schema creates the `LogDepot` application and assigns storage management to the Logging service. It also declares a single table, called `AppLogs`. Additional tables can be defined if we want separate collections. More tables can be added later by modifying the schema and sending it with the REST command `PUT /_applications/LogDepot`.

Applications can be assigned an optional *key*, which is declared as follows:

```
{"LogDepot": {
    "key": "Sassafras",
    ...
}}
```

When a `key` is defined, it must be provided when modifying the application's schema or deleting it. This provides an extra safety check to prevent accidental application changes.

3.2 Tables

A table is a named set of objects. Table names are identifiers that must be unique within the same application. Compared to other services, the Logging service does not require nor allow fields to be explicitly defined: they are dynamically discovered as data is added. Consequently, a table definition consists only of its name and table-level options. The only option available for Logging tables is automatic data aging, which is defined via the `retention-age` option as shown below:

```
{"LogDepot": {
    "options": {
        "StorageService": "LoggingService"
    },
    "tables": {
        "AppLogs": {
            "options": {"retention-age": "3 MONTHS"}
        }
    }
}}
```

In this example, the `AppLogs` table sets its `retention-age` option to 3 months. Objects are automatically deleted when their `Timestamp` field value is older than the given age. The `retention-age` option must be in the form of "N <units>" where N is a positive integer and <units> is any of `DAYS`, `MONTHS`, or `YEARS` or the singular spelling of these mnemonics. When a table enables data aging, a background tasks periodically checks for and deletes expires objects.

When a Logging application defines multiple tables, each table can select data aging independently.

3.3 Objects and Fields

The addressable member of a table is an *object*, whose values are stored within *fields*. What constitutes an object is user application-defined, but generally an object corresponds to a single time-stamped log record, event, transaction, or other entity. In some cases, it makes sense to store multiple records as a single object, such as an ERROR log record and its related stack trace. The following rules are used by the Logging service in managing objects:

- No object ID: Unlike other storage services, Logging service objects do not have an object ID. Individual objects are not separately identifiable unless they are assigned a unique timestamp value.
- Timestamp required: Every object must have a timestamp, assigned to the field called `Timestamp`. All timestamps are considered UTC values and must be given in the format:

  ```
  YYYY-MM-DD hh:mm:ss.fff
  ```

  Where `YYYY-MM-DD` is the date, `hh:mm:ss` is the time, and `fff` is a fractional time. The time elements are optional from right-to-left, and omitted elements default to zero. For example, the value `2015-09-03 14:01` is the same as `2015-09-03 14:01:00.000`. Multiple objects can be assigned the same timestamp.

- **Dynamic fields**: Fields are not predefined in the schema but are assigned dynamically as objects are added. Objects in the same table can have different fields.
- **SV scalars only**: Logging objects are considered "flat" and must consist of simple scalar values. Multi-valued scalar and link fields are not currently supported.
- **Immutability**: Objects cannot be modified once added. The only update operation supported by the Logging service is adding new objects.

3.4 Choosing Logging Object Fields

Other than `Timestamp`, which fields should you assign to objects? Below is an example batch that shows a minimalistic approach, using as few fields as possible. Suppose we want to create an object for the following log record:

```
2015-03-24 14:05:42.893 INFO MBeanProvider: Unregistering the StorageManager MXBean
```

Within a batch, this record could be loaded with the following URI and JSON message:

```
POST /LogDepot/AppLogs
{"batch": {
    "docs": [
       {"doc": {
           "Timestamp": "2015-03-24 14:05:42.893",
           "Message": "INFO MBeanProvider: Unregistering the StorageManager MXBean"
       }},
       {"doc": {
           ...
       }},
    ]
}}
```

Other than the `Timestamp`, the rest of the log record is stored in a single field called `Message`. This allows us to find, for example, all log records that contain the term "info" with a query such as the following:

```
GET /LogDepot/AppLogs/_query?q=Message:info
```

However, this query will return all objects that contain the term *info*—not INFO-level records. To allow more granular queries, we could separate the log `Level` and `Module` info separate fields:

```
...
{"doc": {
    "Timestamp": "2015-03-24 14:05:42.893",
    "Level": "INFO",
    "Module": "MBeanProvider",
    "Message": "Unregistering the StorageManager MXBean"
}}
...
```

Now we can query for all INFO level records using the query:

```
GET /LogDepot/AppLogs/_query?q=Level=info
```

To search for INFO level records that were created with a module whose name begins with `MBean`, we can add a wildcard clause:

```
GET /LogDepot/AppLogs/_query?q=Level=info AND Module="mbean*"
```

Note that field names such as `Level` and `Module` are case-sensitive whereas text field values are not.

Sometimes it makes sense to store both the original, raw log record and separate individual values into their own fields. Then we can use fine-grained queries but also see the full, original log record if needed. For example, here's a typical Apache log record with both the raw message stored in the message field and many other fields extracted:

```
{"doc": {
    "Timestamp": "2011-05-18 19:40:18.000",
    "agent": "\"Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; .NET CLR 2.0.50727;
       .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; InfoPath.2; .NET4.0C; .NET4.0E)\"",
    "auth": "-",
    "bytes": "1189",
    "clientip": "134.39.72.245",
    "datetime": "2011-05-18T19:40:18.000Z",
    "enccred": "U3VwZXJNYW46TG9pcwo=",
    "filename": "access",
    "host": "node2.origin.dev.us.platform.dell.com",
    "httpversion": "1.1",
    "ident": "-",
    "message": "134.39.72.245 - - [18/May/2011:12:40:18 -0700] \"GET /favicon.ico HTTP/1.1\" 200
       1189 \"-\" \"Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; .NET CLR
       2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; InfoPath.2; .NET4.0C; .NET4.0E)\"",
    "path": "/var/lib/openshift/55b6908dd229e7d004000008/app-root/logs/access.log",
    "referrer": "\"-\"",
    "request": "/favicon.ico",
    "response": "200",
    "timestamp": "18/May/2011:12:40:18 -0700",
    "type": "apache",
    "verb": "GET"
}}
```

Notice that message contains fields that are also extracted to other fields such as `clientip`, `request`, and `verb`.

4. Doradus Query Language (DQL)

The Doradus query language (DQL) is used in object and aggregate queries. DQL is analogous to full text languages used by search engines such as Lucene and from which DQL borrows concepts such as *terms*, *phrases*, and *ranges*. The Doradus Logging service supports a subset of DQL, described in this section.

4.1 Query Perspective

A DQL instance is a Boolean expression that selects objects from a *perspective* table. Logically, the query expression is evaluated against each perspective object; if the expression evaluates to "true", the query selects the object. Requested fields are returned by the query for each selected perspective object. In aggregate queries, selected objects are included in one or more metric computations.

4.2 Clauses

A DQL query is comprised of one or more Boolean expressions called *clauses*. Each clause examines a field that is related to perspective objects. The clause evaluates to true for a given object if the examined field matches the clause's condition. Example clauses are shown below:

```
Level:Info
Module=jetty
Timestamp:[2001 TO "2005-06-31"]
NOT Message:Err*
```

These examples show a variety of comparison types: qualified *contains* and *equals*, ranges, and pattern matching.

4.3 Clause Negation: NOT

Any clause can be negated by prefixing it with the uppercase keyword NOT. To use "not" as a literal term instead of a keyword, either enclose it in single or double quotes or use non-uppercase. Double negation (`NOT NOT`) and any even number of NOT prefixes cancel out.

4.4 Clause Connectors: AND, OR, and Parentheses

Clauses are *implicitly* connected with "and" semantics by separating them with at least one whitespace character. For example:

```
LastName:Wright Department:Sales
```

This query consists of two clauses and is identical to the following query expression:

```
LastName:Wright AND Department:Sales
```

Clauses can be *explicitly* connected with the uppercase keywords `AND` and `OR`. When a DQL query has a mixture of `AND` and `OR` clauses, the `AND` clauses have higher evaluation precedence. For example:

```
Origin=3 AND Platform=2 OR Timestamp='2011-10-01' AND Size=1000
```

This is evaluated as if the following parentheses were used:

```
(Origin=3 AND Platform=2) OR (Timestamp='2011-10-01' AND Size=1000)
```

Parentheses can be used to change the evaluation order. For example:

```
Origin=3 AND (Platform=2 OR Timestamp='2011-10-01') AND Size=1000
```

4.5 Literals

With the Logging service, all fields are considered scalar text fields. In DQL expressions, literals syntactically can be and of the following:

- Boolean: `true`, `false`
- Numeric: `0`, `-123`, `3.14`, `2.718e-6`
- Timestamp: `'2015-01-01' '2015-03-13 23:59:59'`
- Terms: `John`, `Fo*`, `Event_413`
- Strings: `"John Smith"`, `'cihSptpZrCM6oXaVQH6dwA=='`

However, a literal is always interpreted as a text value when compared to a field. Within a quoted string value, escaping can be used for non-printable or other characters. Doradus uses the backslash for escaping, and there are two escape formats:

- `\x`: Where `x` can be one of these characters: `t` (tab character), `b` (backspace), `n` (newline or LF), `r` (carriage return), `f` (form feed), `'` (single quote), `"` (double quote), or `\` (backslash).
- `\uNNNN`: This escape sequence can be used for any Unicode character. `NNNN` must consist of four hex characters 0-F.

For example, the string `"Hello!World"` can also be specified as `"Hello\u0021World"`.

4.6 Null Values

A field that has no value for a given object is *null*. Doradus treats null as a "value" that will not match any literal. For example, the clause `Size=0` will be false if the `Size` field is null. This means that `NOT(`*`clause`*`)` will be true if *`clause`* references a null field.

4.7 IS NULL Clause

A scalar or link field can be tested for nullity by using the clause *`field`* `IS NULL`. Examples:

```
Level IS NULL
```

An `IS NULL` clause can be negated with `NOT` as a clause prefix:

```
NOT Level IS NULL
```

4.8 *Contains* Clauses

Text fields can be queried for values that *contain* specific terms. In contrast, equality clauses compare the entire field value. Different types of contains clauses are described below.

4.8.1 Term Clauses

A *term clause* searches a specific field for one or more terms. To designate it as a *contains* clause, the field name must be followed by a colon. Example:

```
Name:Smith
```

This clause searches perspective objects' `Name` field for the term `Smith` (case-insensitive). To specify multiple terms, enclose them in parentheses. Example:

```
Name:(John Smith)
```

To match this clause, an object's `Name` field must contain both `John` and `Smith`, but they can appear in any order and be separated by other terms.

Be sure to enclose multi-term clauses in parentheses! The following query looks like it searches for `John` and `Smith` in the `Name` field:

```
Name:John Smith      // doesn't do what you think!
```

But, this sequence is actually interpreted as two clauses that are AND-ed together:

```
Name:John AND *:Smith
```

Matching objects must have `John` in the `Name` field and `Smith` in any field.

A term can use the wildcards to select terms that match a given pattern. Currently, only the multi-character wildcard * is supported. Example:

```
Name:J*n
```

This clause matches any `Name` that contains a term that begins with 'j' and ends with 'n'.

4.8.2 Phrase Clauses

A *phrase clause* is a *contains* clause that searches for a field for a specific term *sequence*. Its terms are enclosed in single or double quotes. For example:

```
Name:"John Sm*th"
```

This phrase clause searches the `Name` field for the term `John` immediately followed by a term that matches the pattern `Sm*th`. The matching terms may be preceded or followed by other terms, but they must be in the specified order and with no intervening terms. As with term clauses, phrases clauses can use wildcards, and searches are performed without case sensitivity.

4.8.3 Any-field Clauses

Term and phrase clauses can search all fields of the perspective table by eliminating the field name qualifier. The simplest form of an any-field clause is a term clause:

```
John
```

This clause searches all fields of the perspective table for a value contains the term `John` (case-insensitive).

Multiple terms can be provided by separating them by at least one space. For example:

```
John Smith
```

This multi-term clause searches for the terms `John` and `Smith` in any order and across any field. For example, an object will match if it has a `FirstName` field that contains `John` and a `LastName` field that contains `Smith`. If all terms are contained in the same field, they can appear in any order and be separated by other terms. For example, an object will match the example above if it has a `Name` field whose value is "`John Smith`", "`John Q. Smith`", or "`Smith, John`".

An any-field clause can be requested using the field qualifier syntax by using the field name "*". Example:

```
*:(John Smith)
```

This is the same as the unqualified term clause:

```
John Smith
```

Phrase clauses can also use the field qualifier syntax and request an any-field search:

```
*:"John Smith"
```

This query finds object where the terms `John` and `Smith` appear in succession in any field. The phrase can be preceded or followed by other terms.

4.9 Range Clauses

Range clauses can be used to select a scalar field whose value falls within a specific range. The math operators `=`, `<`, `<=`, and `>=` are allowed:

```
Level > error
LastName <= Q
```

Doradus also allows *bracketed* ranges with both *inclusive* ([ ]) and *exclusive* ({ }) bounds. For example:

```
User = [Jane TO Joe}
```

This is shorthand for:

```
User >= Jane AND User < Joe
```

For range clauses, ":" and "=" are identical – both perform an *equals* search. Hence, the previous example is the same as:

```
User:[Jane TO Joe}
```

With the Logging service, the Timestamp field can only be queried using a range clause. Example:

```
Timestamp=['2014-01-01' TO '2014-06-13 12:00']
```

Note that timestamp literals must be quoted, and time elements can be omitted from right-to-left; omitted elements are considered zero.

4.10 Equality Clauses

An equality clause searches a field for a value that matches a literal constant, or, for text fields, a pattern. The equals sign (=) is used to search for an exact field value. The right hand side must be a literal value. Example:

```
Name="John Smith"
```

This searches the `Name` field for objects that exactly match the string "John Smith". The search is case-insensitive, so an object will match if its `Name` field is "`john smith`", "`JOHN SMITH`", etc.

For text fields, wildcards can be used to define patterns. For example:

```
Name="J* Sm?th"
```

This clause matches "`John Smith`", "`Joe Smyth`", etc.

Not equals is written by negating an equality clause with the keyword `NOT`. Example:

```
NOT Size=1024
```

4.11 IN Clause

A field can be tested for membership in a set of values using the `IN` clause. Examples:

```
EventID IN (512,1024,2048,4096,8192,16384,32768,65536)
LastName IN (Jones, Smi*, Vledick)
```

As shown, values are enclosed in parentheses and separated by commas. String values can use wildcards. Literals that are not a simple term must be enclosed in single or double quotes. The `IN` clause is a shorthand for a series of `OR` clauses. For example, the following clause:

```
LastName IN (Jones, Smi*, Vledick)
```

is the same as the following `OR` clauses:

```
LastName = Jones OR LastName = Smi* OR LastName = Vledick
```

As a synonym for the `IN` keyword, Doradus also allows the syntax *`field`*`=(`*`list`*`)`. For example, the following two clauses are equivalent:

```
LastName IN (Jones, Smi*, Vledick)
LastName=(Jones, Smi*, Vledick)
```

4.12 Timestamp Clauses

The `Timestamp` field can be queried using date/time *subfields*. A subfield is accessed using "dot" notation and an upper-case mnemonic. Each subfield is an integer value and can be compared to an integer constant. Only equality (=) comparisons are allowed for subfields. Examples:

```
Timestamp.MONTH = 2         // month = February, any year
Timestamp.DAY = 15          // day-of-month is the 15th
NOT Timestamp.HOUR = 12     // hour other than 12 (noon)
```

The recognized subfields of a timestamp field and their possible range values are:

- `YEAR`: any integer
- `MONTH`: 1 to 12
- `DAY`: 1 to 31
- `HOUR`: 0 to 23
- `MINUTE`: 0 to 59
- `SECOND`: 0 to 59

Though timestamp fields will store sub-second precision, there are no subfields that allow querying the sub-second portion of specific values.

5. REST Commands

This section describes the Doradus REST API and the REST commands supported by Doradus Logging.

5.1 Command Summary

A list of all REST commands available for a specific Doradus server based on its current configuration can be retrieved with the command:

```
GET /_commands
```

Some REST administrative commands are described in the **Doradus Administration** manual. The REST API commands most relevant to Doradus Logging applications are summarized below:

REST Command	Method and URI
Application Management Commands	
Create Application	`POST /_applications`
Modify Application	`PUT /_applications/{application}`
List All Applications	`GET /_applications`
List Application	`GET /_applications/{application}`
Delete Application	`DELETE /_applications/{application}` `DELETE /_applications/{application}/{key}`
Object Update Commands	
Add Batch	`POST /{application}/{table}`
Query Commands	
Object Query	`GET /{application}/{table}/_query?{params}`
Aggregate Query	`GET /{application}/{table}/_aggregate?{params}`
Browse Objects	`GET /_lsapp`
Task Status Command	
Get All Task Status	`GET /_tasks`

As previously described, when Doradus is executing in multi-tenant mode, these *application* commands may be directed to a specific tenant in which case the `?tenant` URI parameter and `Authorization` header must be included in the command.

5.2 REST API Overview

The Doradus REST API can be managed by an embedded Jetty server or an external web server such as Apache Tomcat. All REST commands support XML and JSON messages for requests and/or responses as used by the command. By default, Doradus uses unsecured HTTP, but HTTP over TLS (HTTPS) can be configured, optionally with mandatory client authentication. See the **Doradus Administration** document for details on configuring TLS.

The REST API is accessible by virtually all programming languages and platforms. GET commands can also be entered by a browser, though a plug-in may be required to format JSON or XML results. The `curl` command-line tool is also useful for testing REST commands.

Unless otherwise specified, all REST commands are synchronous and block until they are complete. Object queries can use stateless paging for large result sets.

5.2.1 Common REST Headers

Most REST calls require extra HTTP headers. The most common headers used by Doradus are:

- **`Content-Type`**: Describes the MIME type of the input entity, optionally with a `Charset` parameter. The MIME types supported by Doradus are `text/xml` (the default) and `application/json`.
- **`Content-Length`**: Identifies the length of the input entity in bytes. The input entity cannot be longer than `max_request_size`, defined in the doradus.yaml file.
- **`Accept`**: Indicates the desired MIME type of an output (response) entity. If no `Accept` header is provided, it defaults to input entity's MIME type, or `text/xml` if there is no input entity.
- **`Content-Encoding`**: Specifies that the input entity is compressed. Only `Content-Encoding: gzip` is supported.
- **`Accept-Encoding`**: Requests the output entity to be compressed. Only `Accept-Encoding: gzip` is supported. When Doradus compresses the output entity, the response includes the header `Content-Encoding: gzip`.
- **`Authorization`**: This header is used to provide credentials when required, e.g., when Doradus is operating in multi-tenant mode and the command is directed to a specific tenant. Multi-tenant operation is described later.

Header names and values are case-insensitive.

5.2.2 Common REST URI Parameters

REST support the following general URI query parameters:

- **`format=[json|xml]`**: Requests the output message format in JSON or XML, overriding the `Accept` header if present.
- **`tenant={tenant}`**: Directs the request to the given tenant. This parameter is required for application REST commands when Doradus is operating in multi-tenant mode. The REST command is routed to specified `{tenant}`.

These parameters can be used together or independently. They can be added to any other parameters already used by the REST command, if any. Examples:

```
GET /_applications?format=json
GET /_tasks?tenant=HelloKitty&api=2
```

```
GET /LogDepot/AppLogs/_query?q=EventID=512&s=5&tenant=HelloKitty&format=json
```

5.2.3 REST Commands in Multi-Tenant Mode

By default, Doradus executes in single-tenant mode, hence all applications are owned by a single *default* tenant. In this case, REST commands do not need the `?tenant` parameter described in the previous section, and tenant credentials are not used. Even when Doradus is configured to operate in multi-tenant mode, REST commands intended for applications belonging to the default tenant omit the `?tenant` parameter and authentication credentials. (The default tenant *may* be disabled in multi-tenant mode—see the **Doradus Administration** documentation for details.)

When Doradus is configured to operate in multi-tenant mode, REST commands intended for a specific tenant must include the `?tenant` parameter. Each command must also include valid credentials for the corresponding tenant. Credentials are provided using *basic authorization*, which uses the general `Authorization` header format shown below:

```
Authorization: Basic xxx
```

Where xxx is the tenant user ID and password, separated by a colon, and base64-encoded. For example, if the user ID and password are `Katniss:Everdeen`, the header would look like this:

```
Authorization: Basic S2F0bmlzczpFdmVyZGVlbgo=
```

When Doradus receives this header, the base64 value is decoded and validated against the given tenant. Note that `curl` supports basic authentication by adding the "-u" parameter. Example:

```
curl -u Katniss:Everdeen http://localhost:1123/HelloKitty/...
```

If the tenant user ID or password is incorrect for the identified tenant, the REST command returns a `401 Unauthorized` response.

The only commands that should not provide the `?tenant` parameter in multi-tenant mode are *system* commands, which are not directed to a specific tenant. These commands are used by administrators to create tenants and perform diagnostic operations. In multi-tenant mode, system commands must use basic authorization and provide *super user* credentials. See the **Doradus Administration** documentation for details.

In the remainder of this documentation, all examples are displayed as if Doradus is operating in single-tenant mode or the command is intended for the default tenant in multi-tenant mode.

5.2.4 Common JSON Rules

In JSON, Boolean values can be text or JSON Boolean constants. In both cases, values are case-insensitive. The following two members are considered identical:

```
"AutoTables": "true"
"AutoTables": TRUE
```

Numeric values can be provided as text literals or numeric constants. The following two members are considered identical:

```
"Size": 70392
"Size": "70392"
```

Null or empty values can be provided using either the JSON keyword `NULL` (case-insensitive) or an empty string. For example:

```
"Occupation": null
"Occupation": ""
```

In JSON output messages, Doradus always quotes literal values, including Booleans and numbers. Null values are always represented by a pair of empty quotes.

5.2.5 Illegal Characters in XML Content

Although Doradus allows text fields to use all Unicode characters, XML does not allow element content to contain certain characters. The only characters allowed within XML content are in the following:

```
#x9 | #xA | #xD | [#x20-#xD7FF] | [#xE000-#xFFFD] | [#x10000-#x10FFFF]
```

In an output message, when Doradus creates an XML element whose content would contain any characters outside of this set, it base64-encodes the entire element value and adds the attribute `encoding="base64"` to the element. For example, suppose a field value contains the Unicode value "ABC\u0000". The 4th character, hex 0x00, is not valid in XML, hence the field value would be returned as follows:

```
<field name="foo" encoding="base64">QUJDAA==</field>
```

Similarly, in an input XML message, element content that contains invalid XML characters must be base64-encoded, and the application must add the `encoding="base64"` attribute to the corresponding element.

5.2.6 Common REST Responses

When the Doradus Server starts, it listens to its REST port and accepts commands right away. However, if the underlying Cassandra database cannot be contacted (e.g., it is still starting and not yet accepting commands), REST commands that use the database will return a `503 Service Unavailable` response such as the following:

```
HTTP/1.1 503 Service Unavailable
Content-length: 43
Content-type: Text/plain

Database is not reachable. Waiting to retry
```

When a REST command succeeds, a `200 OK` or `201 Created` response is typically returned. Whether the response includes a message entity depends on the command.

When a command fails due to user error, the response is usually `400 Bad Request` or `404 Not Found`. These responses usually include a plain text error message (similar to the 503 response shown above).

When a command fails due to a server error, the response is typically `500 Internal Server Error`. The response includes a plain text message and may include a stack trace of the error.

5.3 Application Management Commands

REST commands that create, modify, and list applications are sent to the `_applications` resource. Application management REST commands supported by Doradus Logging are described in this section.

5.3.1 Create Application

A new application is created by sending a POST request to the `_applications` resource:

```
POST /_applications
```

The request must include the application's schema as an input entity in XML or JSON format. If the request is successful, a `200 OK` response is returned with no message body.

Because Doradus uses *idempotent* update semantics, using this command for an existing application is not an error and treated as a Modify Application command. If the identical schema is added twice, the second command is treated as a no-op. An example Logging application schema is shown below in JSON:

```
{"LogDepot": {
    "options": {
        "StorageService": "LoggingService"
    },
    "tables": {
        "AppLogs": {
            "options": {"retention-age": "3 MONTHS"}
        }
    }
}}
```

5.3.2 Modify Application

An existing application's schema is modified with the following REST command:

```
POST /_application/{application}
```

where `{application}` is the application's name. The request must include the modified schema in XML or JSON as specified by the request's `content-type` header. Because an application's name cannot be changed, `{application}` must match the application name in the schema. If an application key was used in the original schema, the same key must be provided. If the application was defined without a key, a key can be added by including it in the updated schema. If the request is successful, a `200 OK` response is returned with no message body.

Modifying an application *replaces* its current schema. All schema changes are allowed, including adding, modifying, and removing tables. However, there is no way to rename an existing table. If a table is removed, all of its existing data is deleted.

5.3.3 List Application

A list of all application schemas is obtained with the following command:

```
GET /_applications
```

The schemas are returned in the format specified by the `Accept` header.

The schema of a specific application is obtained with the following command:

```
GET /_applications/{application}
```

where `{application}` is the application's name.

5.3.4 Delete Application

An existing application—including all of its data—is deleted with one of the following commands:

```
DELETE /_applications/{application}
DELETE /_applications/{application}/{key}
```

where `{application}` is the application's name. If the application's schema defines a key, the `{key}` must be provided in the command, and it must match. When an application is deleted, all of its underlying tables and data are deleted.

5.4 Add Objects Command

The Logging service supports a single Add Objects command to add new data. Since data is immutable, existing objects cannot be modified. Objects are automatically deleted when table-level data aging is requested. A batch of new objects is added to a specific table using the following REST command:

```
POST /{application}/{table}
```

where `{application}` is the application name and `{table}` is the table in which the objects are to be added. The command must include an input entity that contains the objects to be added. An example input message in JSON is shown below:

```
{"batch": {
   "docs": [
      {"doc": {
          "Timestamp": "2015-03-24 14:05:42.893",
          "Level": "INFO",
          "Module": "MBeanProvider",
          "Message": "Unregistering the ServerManager MXBean”
      }},
      {"doc": {
          "Timestamp": "2015-03-24 14:05:43.403",
```

```
            "Level": "INFO",
            "Module": "MBeanProvider",
            "Message": "Unregistering the StorageManager MXBean”
        }},
        ...
    ]
}}
```

Notes about adding batches are described below:

- Only the `Timestamp` field is required, which contains the object's timestamp. This value determines how the object is stored and accessed during queries.
- Objects do not need to be added in time order, and multiple objects can have the same `Timestamp` value.
- Logging service objects do not use object IDs, so the `_ID` field should not be set.
- Each field name must start with a letter and contain only letters, digits, or underscores.
- Field values should be simple scalar values: text, numbers, timestamps, etc.
- Updates are more efficient when objects are loaded in batches. However, small batches can be added and are merged into larger batches by a background task.

If the update request is successful, a `201 Created` response is returned with a simple status result message. Example:

```
{"batch-result": {
    "status": "OK"
}}
```

5.5 Object Query Command

An *object query* is a DQL query that selects and returns selected fields for objects in the perspective table. With Doradus Logging, all query parameters are passed in the URI of a GET request. The general form is:

```
GET /{application}/{table}/_query?{params}
```

where `{application}` is the application name, `{table}` is the perspective table to be queried, and `{params}` are URI parameters, separated by ampersands (&) and encoded as necessary. The following parameters are supported:

- **`q=`*`expression`*** (required): A DQL query expression that defines which objects to select. The special expression "*" selects all objects. Examples:

```
q=*    // selects all objects
q=EventID=512
q=Description:(Fail* Error)
```

- **s=*size*** (optional): Limits the number of objects returned. If absent, the page size defaults to the `search_default_page_size` option in the doradus.yaml file. The page size can be set to 0 to disable paging, causing all results to be returned in a single page. Examples:

```
s=50
s=0    // return all objects -- be careful!
```

- **f=*fields*** (optional): A comma-separated list of fields to return for each selected object. Without this parameter, all scalar fields of each object are returned. This is the same as using the special value `*` or `_all`. Examples:

```
f=*
f=Size,EventID
```

- **k=*count*** (optional): Causes the first *count* objects in the query results to be skipped. If the `&k` parameter is omitted or set to 0, the first page of objects is returned. Examples:

```
k=100
k=0    // returns first page
```

- **o=Timestamp [ASC | DESC]** (optional): Causes the object query results to be sorted by ascending or descending `Timestamp` order. By default, object query results are sorted in descending `Timestamp` order (newest objects first). Specifying `o=Timestamp ASC` causes oldest objects to be returned first.

- **g=*timestamp*** (optional): Returns a secondary page of objects starting "at" the given timestamp. When a query returns a full page of objects, a `continue` element is included in the results to indicate what value can be used for this parameter. The objects returned in the secondary page are "greater than or equal to" the `continue` value. The `&g` and `&e` parameters cannot both be used in the same query.

- **e=*timestamp*** (optional): Returns a secondary page of objects "after" the given timestamp. When a query returns a full page of objects, a `continue` element is included in the results to indicate what value can be used for this parameter. The objects returned in the secondary page are "greater than" the `continue` value. The `&g` and `&e` parameters cannot both be used in the same query.

An object query always returns an output entity even if there are no objects matching the query request. The outer element is `results`, which contains a single `docs` element. If the query returns no results, the `docs` element is empty. In XML:

```
<results>
   <docs/>
</results>
```

In JSON:

```
{"results": {
   "docs": []
}}
```

When objects are selected by an object query, each object is returned in a `doc` element with each requested `field`. For example:

```
GET /LogDepot/AppLogs/_query?q=*&f=EventID,Timestamp,UserSID
```

This query returns all objects in the `AppLogs` table and produces a result such as this:

```
<results>
   <docs>
      <doc>
         <field name="EventID">672</field>
         <field name="Timestamp">2006-01-13 00:00:00.000</field>
         <field name="UserSID">S-1-5-18</field>
      </doc>
      <doc>
         <field name="EventID">538</field>
         <field name="Timestamp">2006-01-13 00:00:00.000</field>
         <field name="UserSID">S-1-5-21-1874068349-1688302950-636360099-52692</field>
      </doc>
      ...
   </docs>
</results>
```

In JSON:

```
{"results": {
   "docs": [
      {"doc": {
         "EventID": "672",
         "Timestamp": "2006-01-13 00:00:00.000",
         "UserSID": "S-1-5-18"
      }},
      {"doc": {
         "EventID": "538",
         "Timestamp": "2006-01-13 00:00:00.000",
         "UserSID": "S-1-5-21-1874068349-1688302950-636360099-52692"
      }},
      ...
   ]
}}
```

Each requested field is returned even if it has no value.

If a query returns a full page of results, a “continue” value is returned, which can be used with the “continue at” or “continue after” parameters to get a secondary page. Example:

```
{"results": {
   "docs": [
      ...
      {"doc": {
         "EventID": "673",
```

```
            "Timestamp": "2006-01-13 00:00:03.000",
            "UserSID": "S-1-5-18"
        }}
    ],
    "continue": "2006-01-13 00:00:03.000"
}}
```

5.6 Aggregate Query Command

Aggregate queries perform metric calculations across objects selected from the perspective table. Compared to an object query, which returns fields for selected objects, an aggregate query only returns the metric calculations. An aggregate query provides all parameters in the URI of a GET request. The REST command is:

```
GET /{application}/{table}/_aggregate?{params}
```

where `{application}` is the application name, `{table}` is the perspective table, and `{params}` are URI parameters separated by ampersands (&). The following parameters are supported:

- `m=`***`metric function`*** (optional): The metric function to calculate for selected objects. Currently, the only metric function supported by Doradus Logging is COUNT(*), so this parameter can be omitted. Example:

  ```
  m=COUNT(*)
  ```

- `q=`***`text`*** (optional): A DQL query expression that defines which objects to include in metric computations. If omitted, all objects are selected (same as `q=*`). Examples:

  ```
  q=*
  q=Level IN (warning, error)
  ```

- `f=`***`grouping expression`*** (optional): A grouping field expression, which divide computations into groups. When this parameter is omitted, the corresponding *global query* computes a single value for each metric function. When provided, the corresponding *grouped query* computes a value for each group value. Examples:

  ```
  f=Tags
  f=TOP(3,EventID)
  f=TRUNCATE(Timestamp,HOUR)
  ```

Below is an example aggregate query using URI parameters:

```
GET /LogDepot/AppLogs/_aggregate?q=Level=error&m=COUNT(*)&f=TOP(3,MessageType)
```

5.6.1 Global Aggregates

Without a grouping parameter, an aggregate query returns a single value: the metric function computed across all selected objects. Consider the following aggregate query URI REST command:

```
GET /LogDepot/AppLogs/_aggregate?m=COUNT(*)&q=EventID=540
```

This aggregate query returns counts all objects in the `AppLogs` table whose `EventID` is 540. A typical response in XML is:

```
<results>
   <aggregate metric="COUNT(*)" query="EventID=540"/>
   <totalobjects>309142</totalobjects>
   <value>309142</value>
</results>
```

In JSON:

```
{"results": {
   "aggregate": {
      "metric": "COUNT(*)",
      "query": "EventID=540"
   },
   "totalobjects": "309142",
   "value": "309142"
}}
```

As shown, the `aggregate` element lists the parameters used for the aggregate query, in this case the metric (`?m`) and query (`&q`) parameters. For a global aggregate, the metric value is provided in the `value` element. The number of objects scanned to compute the metric value is also returned in the `totalobjects` element.

5.6.2 Grouped Aggregates

When a grouping parameter is provided, objects are divided into sets based on the distinct values created by the grouping expression. A separate metric value is computed for each group. Example:

```
GET /LogDepot/AppLogs/_aggregate?m=COUNT(*)&q=UserSID='S-1-5-18'&f=EventID
```

This query searches for `AppLogs` objects whose `UserSID` field equals the value `S-1-5-18`. Results are divided intro groups based on the `EventID` field. A typical JSON result shown below:

```
{"results": {
   "aggregate": {
      "metric": "COUNT(*)",
      "query": "UserSID='S-1-5-18'",
      "group": "EventID"
   },
   "totalobjects": "172274",
   "summary": "172274",
   "totalgroups": "19",
   "groups": [
      {"group": {
         "metric": "1",
         "field": {"EventID": "512"}
      }},
      {"group": {
         "metric": "7",
         "field": {"EventID": "514"}
```

```
        }},
        ...
        {"group": {
            "metric": "1",
            "field": {"EventID": "806"}
        }}
    ]
}}
```

For grouped aggregate queries, the `results` element contains a `groups` element, which contains one `group` element for each group value. Each `group` contains the `field` name and value for that group and the corresponding `metric` value. The `totalobjects` value computes the number of objects selected in the computation. The `summary` value computes the metric value across all selected objects independent of groups, which, for `COUNT(*)`, is the same as the `totalobjects` value.

5.6.3 Grouping Field Aliases

You can rename the grouping field element in the query results by using an *alias*. The alias name follows the grouping expression prefixed with the keyword `AS`. For example:

```
GET /LogDepot/AppLogs/_aggregate?m=COUNT(*)&q=UserSID='S-1-5-18'&f=EventID AS ErrorNo
```

The alias `ErrorNo` replaces the grouping field `EventID` in the output:

```
{"results": {
    "aggregate": {
        "metric": "COUNT(*)",
        "query": "UserSID='S-1-5-18'",
        "group": "EventID AS ErrorNo"
    },
    "totalobjects": "172274",
    "summary": "172274",
    "totalgroups": "19",
    "groups": [
        {"group": {
            "metric": "1",
            "field": {"ErrorNo": "512"}
        }},
        {"group": {
            "metric": "7",
            "field": {"ErrorNo": "514"}
        }},
        ...
        {"group": {
            "metric": "1",
            "field": {"ErrorNo": "806"}
        }}
    ]
}}
```

5.6.4 Special Grouping Functions

This section describes special functions that provide enhanced behavior for the grouping parameter.

5.6.4.1 TOP and BOTTOM Functions

By default, all group values are returned, and the groups are returned in ascending order of the grouping field value. The `TOP` and `BOTTOM` functions can be used to return groups in the order of the metric value. Optionally, they can also be used to limit the number of groups returned to the *highest* or *lowest* metric values. The `TOP` and `BOTTOM` functions *wrap* a grouping field expression and specify a *limit* parameter. For example:

```
GET /LogDepot/AppLogs/_aggregate?m=COUNT(*)&q=UserSID='S-1-5-18'&f=TOP(3,EventID)
```

The first parameter to TOP/BOTTOM is the limit value; the second parameter is the grouping field. This aggregate query counts objects, grouped by `EventID`, but it only returns the groups with the three highest values. Typical results for the example above in JSON:

```
{"results": {
    "aggregate": {
       "metric": "COUNT(*)",
       "query": "UserSID='S-1-5-18'",
       "group": "TOP(3,EventID)"
    },
    "totalobjects": "172274",
    "summary": "172274",
    "totalgroups": "19",
    "groups": [
       {"group": {
          "metric": "119161",
          "field": {"EventID": "673"}
       }},
       {"group": {
          "metric": "30815",
          "field": {"EventID":"672"}
       }},
       {"group": {
          "metric": "4353",
          "field": {"EventID":"675"}
       }}
    ]
}}
```

The `BOTTOM` parameter works the same way but returns the groups with the lowest metric values. When either the `TOP` or `BOTTOM` function is used, the total number of groups that were actually computed is returned in the element `totalgroups`, as shown above.

When the limit parameter is 0, all groups are returned, but they are returned in metric-computation order. Using 0 for the limit parameter essentially means *unlimited*.

5.6.4.2 FIRST and LAST Functions

The FIRST and LAST functions are similar to the TOP and BOTTOM functions except that a limited set of groups are returned based on the group names instead of the metric values.

```
GET /LogDepot/AppLogs/_aggregate?m=COUNT(*)&q=UserSID='S-1-5-18'&f=FIRST(3,EventID)
```

This query returns the first 3 groups based on the alphabetical sorting of EventID values. LAST returns the last groups in descending sort order. These functions help with aggregate queries that must be "paged".

5.6.4.3 TRUNCATE Function

The TRUNCATE function truncates a timestamp field to a given granularity, yielding a value that can be used as a grouping field. Before the timestamp field is truncated, the TRUNCATE function can optionally *shift* the value to another time first. The syntax for the function is:

```
TRUNCATE(<timestamp field>, <precision> [, <time shift>])
```

For example:

```
GET /LogDepot/AppLogs/_aggregate?m=COUNT(*)&f=TRUNCATE(Timestamp,DAY,GMT-2)
```

This query counts all AppLogs objects. For each one, it subtracts 2 hours from the Timestamp value and then truncates ("rounds down") to the nearest day. The count of all objects for each truncated timestamp is computed in a separate group.

The <precision> value must be one of the following mnemonics:

Precision	Meaning
SECOND	The milliseconds component of the timestamp is set to 0.
MINUTE	The milliseconds and seconds components are set to 0.
HOUR	The milliseconds, seconds, and minutes components are set to 0.
DAY	All time components are set to 0.
WEEK	All time components are set to 0, and the date components are set to the Monday of the calendar week in which the timestamp falls (as defined by ISO 8601). For example 2010-01-02 is truncated to the week 2009-12-28.
MONTH	All time components are set to 0, and the day component is set to 1.
QUARTER	All time components are set to 0, the day component is set to 1, and the month component is rounded "down" to January, April, July, or October.
YEAR	All time components are set to 0 and the day and month components are set to 1.

The optional <time shift> parameter adds or subtracts a specific amount to each object's timestamp value before truncating it to the requested granularity. Optionally, the parameter can be quoted in single or double quotes. The syntax of the <time shift> parameter is:

```
<timezone> | <GMT offset>
```

Where <GMT offset> uses the same format as the NOW function:

```
GMT<sign><hours>[:<minutes>]
```

The meaning of each format is summarized below:

- `<timezone>`: A timezone abbreviation (e.g., `"PST"`) or name (e.g., `"America/Los_Angeles"`) can be given. Each object's timestamp value is assumed to be in GMT (UTC) time and adjusted by the necessary amount to reflect the equivalent value in the given timezone. The allowable values for a <timezone> abbreviation or name are those recognized by the Java function `java.util.TimeZone.getAvailableIDs()`.

- `GMT+<hour>` or `GMT-<hour>`: The term GMT followed by a plus or minus signed followed by an integer hour value adjust each object's timestamp up or down by the given number of hours.

- `GMT+<hour>:<minute>` or `GMT-<hour>:<minute>`: This is the same as the previous format except that each object's timestamp is adjusted up or down by the given hour and minute value.

Note that in the GMT versions, the sign ('+' or '-') is required, and in URIs, the '+' sign must be escaped as `%2B`.

When a grouping field uses the `TRUNCATE` function, the truncated value is used for the field value within each group. An example is shown below:

```
{"results": {
    "aggregate": {
       "metric": "COUNT(*)",
       "query": "*",
       "group": "TRUNCATE(Timestamp,DAY,GMT-2)"},
       "totalobjects": "2000090", "summary": "2000090",
       "totalgroups": "3",
       "groups": [
          {"group": {
             "metric": "105457",
             "field": {"TRUNCATE(Timestamp,DAY,GMT-2)": "2006-01-12"}
          }},
          {"group": {
             "metric": "605818",
             "field": {"TRUNCATE(Timestamp,DAY,GMT-2)": "2006-01-13"}
          }},
          {"group": {
             "metric": "288770",
             "field": {"TRUNCATE(Timestamp,DAY,GMT-2)":"2006-01-13"}
          }}
       ]
    }}
```

5.7 Task Status Command

Doradus Logging uses background tasks to perform data aging and batch merging. These tasks execute automatically at their defined frequency. On multi-node clusters, Doradus ensures that each task is

executed by only a single node at each scheduled occurrence. The following REST command is available to list the status of all known tasks:

```
GET /_tasks
```

This command returns a response such as the following:

```
<tasks>
   <task name="LogDepot/*/logs-merging">
      <Status>Completed</Status>
      <Executor>10.1.161.79</Executor>
      <FinishTime>2015-09-04 14:59:46</FinishTime>
      <StartTime>2015-09-04 14:59:46</StartTime>
   </task>
   <task name="LogDepot/*/logs-aging">
      <Status>Completed</Status>
      <Executor>192.168.1.12</Executor>
      <FinishTime>2015-09-03 15:48:57</FinishTime>
      <StartTime>2015-09-03 15:48:57</StartTime>
   </task>
</tasks>
```

In JSON:

```
{"tasks": {
   LogDepot/*/logs-merging": {
      "Status": "Completed",
      "Executor": "10.1.161.79",
      "FinishTime": "2015-09-04 14:59:46",
      "StartTime": "2015-09-04 14:59:46"
   },
   LogDepot/*/logs-aging": {
      "Status": "Completed",
      "Executor": "192.168.1.12",
      "FinishTime": "2015-09-03 15:48:57",
      "StartTime": "2015-09-03 15:48:57"
   }
}}
```

The status of each task is listed using the task name `<application>/*/data-aging`. The status, start time, and finish time is shown. The `Executor` is the IP address if the Doradus node that performed the task.

5.8 Browse Objects Command

A simple log object browser can be invoked via the URL:

```
GET /_lsapps
```

The initial page returns a table of Logging service applications within the database. Example:

Applications	Tables

LogDepot	
	AppLogs
LoggingEvents	
	Events
MyApp	
	MyLogs

You can click on an table name to display the first (newest) objects in the table. Each page has page controls to display the first, next, previous, and last page of data. This interface is useful for casual browsing of log objects.