Doradus Administration

1. Overview

This document provides configuration and operation guidelines for the Doradus database management system. This document is relevant to the v2.4 release (currently under development). It provides information on the following topics:

- Overview of the Doradus architecture
- Deploying Doradus in a minimal environment
- Expanding the deployment to accommodate additional capacity
- Configuring Doradus multi-tenant operation
- Security and general configuration parameters
- Monitoring Doradus via JMX

1.1. Recent Changes

The following changes were made to reflect new features for the v2.4 release:

- A new configuration parameter `secondary_dbhost` allows failover to a secondary host or list of hosts when all hosts in the primary list, specified by `dbhost`, are unreachable.

- A new command-line parameter `-param_override_filename` allows parameter overrides to be placed in a separate file instead of modifying `doradus.yaml` directly.

- Tenant management has been enhanced to support user-specific permissions.

- Doradus can be hosted as an Apache Tomcat application, using it to handle the REST API. Documentation has been added to describe the building and use of the `doradus-tomcat` project.

- A new parameter has been added to allow fine-tuning of OLAP compression overhead: `olap_compression_level`.

- A new parameter has been added to allow use of multiple threads in multi-shard OLAP queries: `olap_search_threads`.

- A new `GET /_config` command retrieves the Doradus server's current version and configuration parameters.

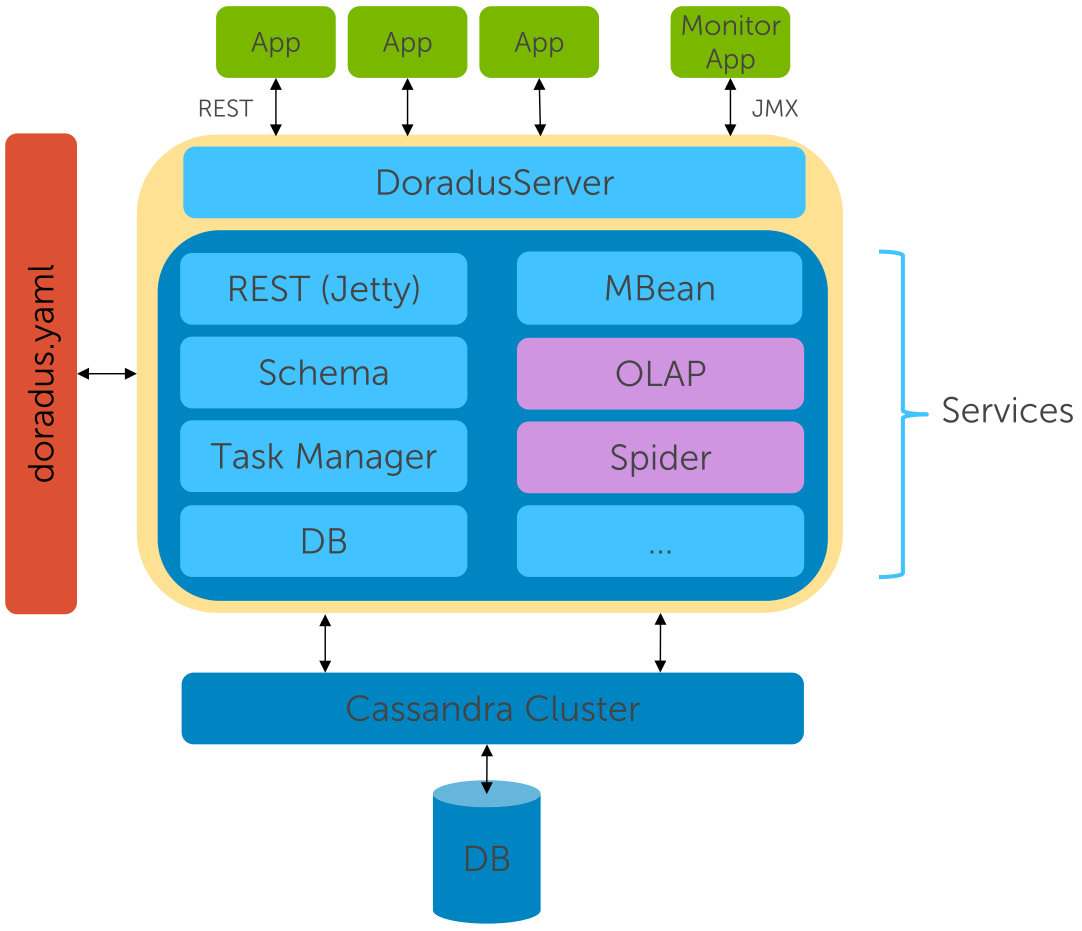
- A new `GET /_dump` command returns a snapshot stack trace of all current Doradus server threads.

- A new `GET /_logs` command returns the most recent logs recorded in the Doradus server's `MemoryAppender`, which typically contains DEBUG log entries not present in the file log.

2. Architecture Overview

Doradus is a Java server application that leverages and extends the Cassandra NoSQL database. At a high level, it is a REST service that sits between applications and a Cassandra cluster, adding powerful features to—and hiding complexities in—the underlying database. This allows applications to leverage the benefits of NoSQL such as horizontal scalability, replication, and failover while enjoying rich features such as full text searching, bi-directional relationships, and powerful analytic queries.

An overview of Doradus architecture is depicted below:



Key components of this architecture are summarized below:

- **Apps**: One or more applications access a Doradus server instance using a simple **REST** API. A **JMX** API is available to monitor Doradus and perform administrative functions.

- **DoradusServer**: This core component controls server startup, shutdown, and services. Entry points are provided to run the server as a stand-alone application, as a Windows service (via procrun), or embedded within another application.

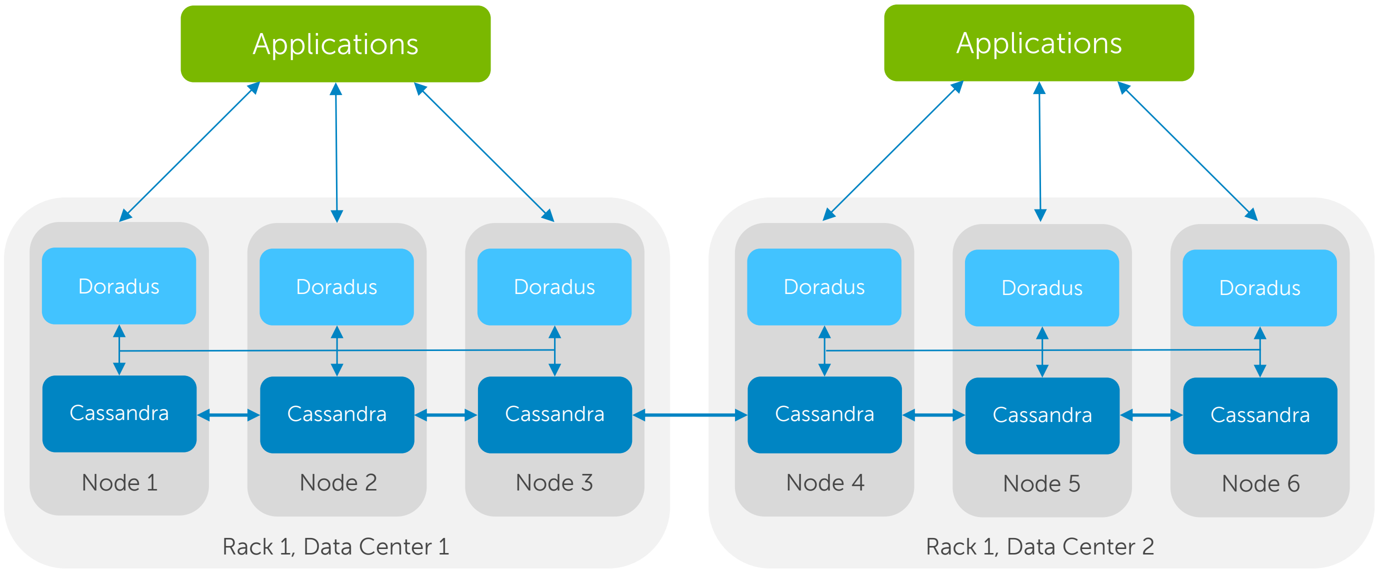
- **Services**: Doradus' architecture encapsulates functions within service modules. Services are initialized based on the server's **doradus.yaml** configuration. Services provide functions such as the **REST** API (an embedded Jetty server), **Schema** processing, and physical **DB** access. A special class of *storage services* provide storage and access features for specific application types. Doradus currently provides multiple storage services, include the OLAP, Spider, and Logging services. Doradus can be configured to use multiple storage services in a single instance.

- **Cassandra Cluster**: Doradus currently uses the Apache Cassandra NoSQL database for persistence. Future releases are intended to use other data stores. Cassandra performs the "heavy lifting" in terms of persistent, replication, load balancing, replication, and more.

By default, Doradus operates in *single-tenant* mode, which means that all applications are stored in a single Cassandra keyspace. In *multi-tenant* mode, named tenants own one or more applications stored in a separate keyspace. Multi-tenant mode allows multiple applications to share a common Doradus cluster while providing data isolation and security. Full details on configuring and operating multi-tenant mode are described in the **Doradus Administration** documentation.

The minimal deployment configuration is a single Doradus instance and a single Cassandra instance running on the same machine. On Windows, these instances can be installed as services. The Doradus server can also be embedded in the same JVM as an application.

Multiple Doradus and Cassandra instances can be deployed to scale a cluster horizontally. An example of a Doradus/Cassandra multi-node cluster is shown below:



This example demonstrates several deployment features:

- One Doradus instance and one Cassandra instance are typically deployed on each node.

- Doradus instances are *peers*, hence an application can submit requests to any Doradus instance in the cluster.

- Each Doradus instance is typically configured to use all *network near* Cassandra instances. This allows it to distribute requests to local Cassandra instances, providing automatic failover should a Cassandra instance fail.

- Cassandra can be configured to know which nodes are in the same *rack* and which racks are in the same *data center*. With this knowledge, Cassandra uses replication strategies to balance network bandwidth and recoverability from node-, rack-, and data center-level failures.

3. Installing and Running Doradus

Doradus and Cassandra are both pure Java applications and run on Windows, Linux, and OS X. Doradus can be run in a variety of ways as a standalone application or embedded within another application. Download, installing, and running Doradus are described in this section.

3.1. Downloading Cassandra

Instructions for downloading Cassandra are available on the Apache Cassandra web site:

http://cassandra.apache.org/

Doradus works with 2.0 versions and higher. It has tested the most with the 2.0.x branch.

3.2. Downloading Doradus

Doradus is a Github project whose source code and build scripts can be found at the following link:

https://github.com/dell-oss/Doradus

There are two basic ways to download Doradus:

1. Download the entire file tree as a single zip file using this link:

   https://github.com/dell-oss/Doradus/archive/master.zip

   Unzip this file, which will create a parent folder called "Doradus-master" containing build, source, and document files. Or:

2. Clone the entire project using a Git client to your working space. For example, using a command line client:

   ```
   git clone https://github.com/dell-oss/Doradus.git
   ```

   This will create a local Git repository in a parent folder called "Doradus".

3.3. Compiling Doradus

Compiling Doradus requires JDK 1.8 and either Ant or Maven. Below are instructions for both Ant and Maven. With both approaches, the folder called `/doradus-server` is referred to as `{doradus_home}`.

3.3.1. Compiling with Maven

From the parent folder, enter the following Maven command:

```
mvn clean install dependency:copy-dependencies -DskipTests=true -Dgpg.skip=true
    -Dmaven.javadoc.skip=true
```

This command compiles all Doradus components: `doradus-client`, `doradus-common`, `doradus-server`, `doradus-jetty`, etc. Class and jar files are created under `target` subdirectories of each component, and dependent jar files are copied to `target/dependency` folders under each component. Therefore, you should see a directory structure similar to this:

```
./Doradus (or whatever your root directory is)
    /doradus-client
        ...
    /doradus-common
        ...
    /doradus-server
        /config
        /script
        /src
        /target
            /classes
                /com
                doradus.yaml
                log4j.properties
            /dependency
            ...
    /doradus-jetty
        ...
```

In general, resource files are copied to `target/classes` folder. Jar files needed for runtime reside in the `target` and `target/dependency` folders.

3.3.2. Compiling with Ant

To compile Doradus using Ant, just enter `ant` from the home directory. All components (doradus-common, doradus-server, etc.) are built. The Ant build places binaries in different locations than Maven. You should see a folder structure such as this:

```
./Doradus (or whatever your root directory is)
    /doradus-client
        ...
    /doradus-common
        ...
    /doradus-server
        /bin
        /config
            doradus.yaml
            log4j.properties
        /lib
        ...
    /doradus-jetty
        ...
```

Resource files are copied to the `config` folder. Generated and dependent jar files are placed in the `lib` folder.

3.4. Running Doradus Standalone

Java 1.8 is required to run Doradus. The easiest way to run Doradus is as a stand-alone application using its embedded Jetty server. If you build Doradus using Maven, from the directory `{doradus_home}/doradus-jetty` use a command such as the following

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

If you compiled Doradus with Ant, use a command such as the following:

```
java -cp ./lib/*:../doradus-server/config:../doradus-server/lib/*
    com.dell.doradus.core.DoradusServer
```

Parameters are passed to Doradus as follows:

- Configurable parameters are defined in the file `doradus.yaml`. In Github, it resides in the folder `doradus-server/src/main/resources`. During builds, this file is copied to the folder `doradus-server/target/classes` (Maven) or `doradus-server/config` (Ant). This appropriate folder should be in your `classpath`. You can specify an alternate location with the following Java parameter:

  ```
  -Ddoradus.config=file:/Doradus/my-doradus.yaml
  ```

- Logging options are defined in the file `log4j.properties`, which resides in the same folder as `doradus.yaml`. Specify an alternate location with the parameter:

  ```
  -Dlog4j.configuration=file:/Doradus/mylog4j.properties
  ```

- Most `doradus.yaml` parameters can be overridden in command line arguments by prepending a "-" to the parameter name. For example, to override `restport`, add the command line argument:

  ```
  -restport 5711
  ```

  List-valued parameter values should be separated with commas and no spaces. Example:

  ```
  -tls_cipher_suites TLS_ECDH_RSA_WITH_AES_128_CBC_SHA,TLS_ECDH_RSA_WITH_AES_256_CBC_SHA
  ```

- Instead of modifying doradus.yaml or passing command line arguments, you can place all your non-default configuration parameters in their own file and specify the file's location with a single command-line argument `-param_override_filename`. Example:

  ```
  -param_override_filename /Users/Me/myoverrides.yaml
  ```

3.5. Running Doradus using Tomcat

Instead of using the embedded Jetty server to serve its REST API, you can host Doradus in the Apache Tomcat server. The folder `doradus-tomcat` contains interface source files, a `web.xml` file, and a `pom.xml` build file. (Currently, this project only supports Maven builds.) To use the Tomcat interface, use the following steps:

1) Download and install Tomcat. Instructions for Tomcat 7 can be found here: https://tomcat.apache.org/tomcat-7.0-doc/appdev/installation.html.

2) Copy the latest `doradus.yaml` file from `./doradus-server/src/main/resources` to `./doradus-tomcat/src/main/resources`.

3) Modify the `doradus.yaml` and comment-out the `webserver_class` parameter. Example:

   ```
   #webserver_class: com.dell.doradus.server.JettyWebServer
   ```

   Commenting-out this line prevents Doradus from trying to initialize an embedded web server. Make any additional changes you need to this file at the same time.

4) The provided `web.xml` file allows Doradus REST commands to respond to ROOT URIs (no prefix) or the special prefix `_api`. For example, if Doradus runs as ROOT, the "list applications" command will be:

   ```
   GET /_applications
   ```

   If Doradus is assigned the URI prefix `_api`, the "list applications" command will be:

   ```
   GET /_api/_applications
   ```

   If you'd like to use a different API prefix, modify `web.xml` and add a section such as the following:

   ```
   <!--Doradus running with URI prefix "Fuzzy" -->
   <servlet-mapping>
       <servlet-name>DoradusRestServlet</servlet-name>
       <url-pattern>/Fuzzy/*</url-pattern>
   </servlet-mapping>
   ```

5) Generate the `war` file by compiling `doradus-tomcat` using `pom.xml` in the root folder. If Doradus will use no URI prefix, use the following command:

   ```
   $ mvn compile war:war -Dwar.warName=ROOT
   ```

   If Doradus will use a URI prefix, change `ROOT` to the appropriate prefix name.

6) Copy the war file (e.g., `ROOT.war` or `_api.war`) from the `target` folder to `<TOMCAT_HOME>/webapps`.

7) Make any other Tomcat changes you need by adjusting the corresponding configuration files. For example, the default port number (8080) is defined in the file `<TOMCAT_HOME>config/server.xml`.

8) Start Tomcat from the `<TOMCAT_HOME>/bin` folder, e.g., with `./catalina.sh run` on Mac or Linux or just `catalina` for Windows. The doradus-tomcat war file will be expanded automatically when Tomcat starts.

Note that the doradus.log file should appear in the `<TOMCAT_HOME>/bin` folder.

3.6. Running Doradus as a Windows Service

On Windows, Doradus can also be installed and managed as a service using the Apache Commons Daemon component. See http://commons.apache.org/proper/commons-daemon/procrun.html for more information.

3.7. Running Doradus as a Daemon

On Linux, Doradus can be executed as a daemon that is launched at system startup. Example scripts are provided in the doradus-server's scripts folder:

```
doradus-server
   /scripts
      doradusserver
      /init.d
         doradusserver
```

The `doradusserver` script under `/init.d` can be placed in the system's `/etc/init.d` folder. See the comments in the header on modifying the file and setting permissions. The `doradusserver` file under `/scripts` is an example run file for the Doradus server. Modify it to pass parameters and JVM arguments that should be used when Doradus is started by the `init.d` script.

Doradus can also be run as a daemon on OS X using the recommended `launchd` manager. See the documentation for creating launch daemons here:

> https://developer.apple.com/library/mac/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingLaunchdJobs.html

3.8. Embedding Doradus In Another Application

Some applications can benefit from embedding Doradus in the same JVM process. Such applications call internal Doradus service methods directly instead of using the REST API. This offers higher performance since the overhead of the REST API is avoid, hence it is ideal when performance is critical such as for bulk load applications.

To embed Doradus in the same JVM, include the Doradus server jar files in the `classpath`. There are two ways to start Doradus depending on if the calling wants to block:

3.8.1. DoradusServer.main() and DoradusServer.startServer()

This static method is passed runtime arguments, if any as a `String[]`. Example:

```
import com.dell.doradus.core.DoradusServer;
...
String[] args = new String[] {"-restport", "5711"};
DoradusServer.main(args);   // blocks until shutdown
```

This example overrides the parameter `restport`, setting it to `5711`. When `main()` is called, Doradus starts all internal services and the `storage_services` configured in `doradus.yaml`. However, `main()` does not return until the process receives a shutdown signal (e.g., Ctrl-C or `System.exit()` is called).

3.8.2. DoradusServer.startEmbedded()

This method returns as soon as all internal services are initialized and started. In addition to runtime arguments (like `main()`), this method accepts a second `String[]` that specifies optional services. Example:

```
String[] args = new String[] {"-restport", "5711"};
String[] services = new String[] {OLAPService.class.getName()};
DoradusServer.startEmbedded(args, services); // returns when services are started
```

The `services` parameter overrides the doradus.yaml file parameter `default_services`, starting only the given optional services. The example above starts Doradus with the OLAP storage service. But the REST, Task Manager, and other optional services are not initialized. Note that Doradus always initializes the DB and Schema services, which are required. See the `doradus.yaml` file `default_services` option for a description of optional services that may be requested.

The `services` parameter must include full package name of each service. At least one storage service must be initialized otherwise `startEmbedded()` will throw a `RuntimeException`. If multiple storage services are provided, the first one becomes the default for new applications created via the embedded Doradus instance that do not explicitly declare a storage service.

Although `startEmbedded()` returns when all requested and requires services have started, a service may not be immediately ready to use. For example, the internal database service is not ready until a connection to Cassandra is established. Until it does, the schema service and all storage services also will not be ready. To ensure that a needed service is ready to use, the application can call the `waitForFullService()` method. For example, to wait for the `OLAPService` to be ready:

```
OLAPService.instance().waitForFullSerice();  // wait until OLAP is ready.
```

Note that multiple Doradus instances can be launched for the same Cassandra cluster with different service sets. For example, a bulk load application could embed Doradus, initializing only the storage service that it requires, while another standalone instance of Doradus can be executed with full services.

When Doradus is started with the `startEmbedded()` method, it returns as soon as all requested services are initialized and running. Doradus can be gracefully shutdown by calling the `shutDown` method. Example:

```
DoradusServer.shutDown();      // gracefully shutdown and return
```

Alternatively, Doradus can be gracefully shutdown and terminate the JVM process by calling `stopServer`. Example:

```
DordusServer.stopServer(null); // gracefully shutdown and call System.exit()
```

The parameter passed to `stopServer()` is a `String[]`, but it is ignored.

4. Administrative REST Commands

Several REST commands are available for monitoring and managing Doradus instances. Below is a summary of general purpose REST commands:

Command	REST URI	Comments
System Commands		
List configuration	`GET /_config`	• Lists Doradus version and all configuration parameters. • Privileged command in multi-tenant mode.
Thread dump	`GET /_dump`	• Takes a snapshot stack trace of all threads. • Privileged command in multi-tenant mode.
Memory log	`GET /_logs`	• Returns latest logs in memory appender; DEBUG log entries by default. • Privileged command in multi-tenant mode.
List Commands	`GET /_commands`	• Lists all REST commands based on currently-loaded services.
Tenant Commands		
Create new tenant	`POST /_tenants`	• Create a new tenant in multi-tenant mode. • Privileged command.
Modify tenant	`PUT /_tenants/{tenant}`	• Modify an existing tenant in multi-tenant mode. • Privileged command.
Delete tenant	`DELETE /_tenants/{tenant}`	• Delete a tenant and all applications in multi-tenant mode. • Privileged command.
List all tenants	`GET /_tenants`	• List all tenants in multi-tenant mode. • Privileged command.
List tenant	`GET /_tenants/{tenant}`	• Lists tenant details in multi-tenant mode. • Privileged command.
Task Commands		
List tasks	`GET /_tasks`	• List all tasks for the default or a specific tenant.

Tenant commands are described in more detail under **Multi-Tenant Configuration**. In addition to these system commands, each Doradus storage service supports application-specific REST commands.

5. Deployment Guidelines

This section provides an overview of options for deploying and configuring Doradus.

5.1. Deployment Options

The basic components used by Doradus are illustrated below:

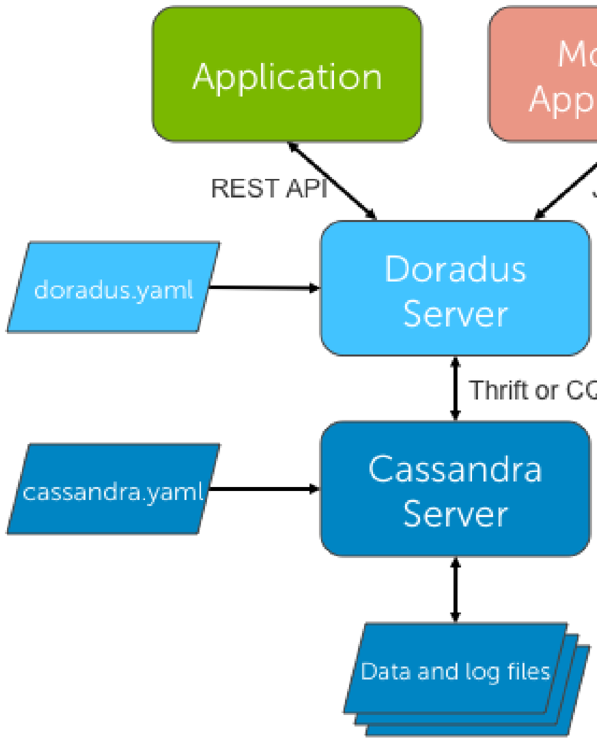


Figure 1 - Basic Doradus Components

These components are:

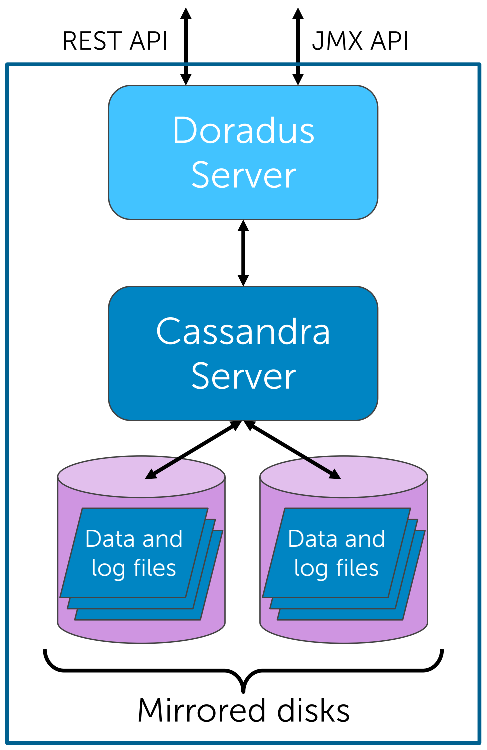
- **Application**: One or more client applications perform schema changes, update objects, and submit queries using the Doradus **REST API** when Doradus executes as a standalone process. However, applications can also embed Doradus in the same JVM and call internal services directly.

- **Monitor Application**: Doradus uses the **JMX API** to monitor server resource usage. A JMX application such as JConsole can be used to invoke JMX functions.

- **Doradus Server**: This is the core Doradus component, which processes commands and maps requests to Cassandra using either the **Thrift API** or **CQL API**. The `doradus.yaml` file is the primary Doradus configuration file.

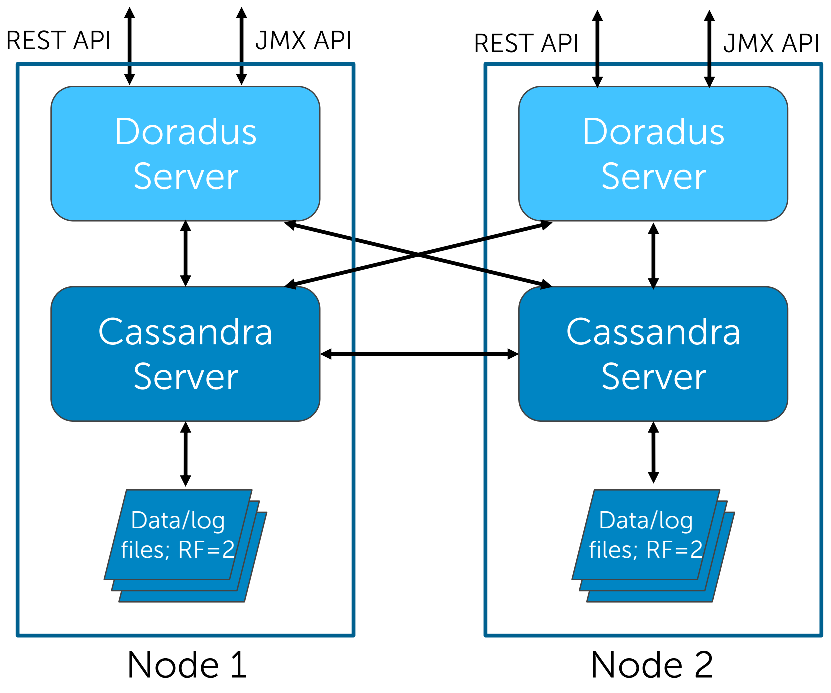
- **Cassandra Server**: This is the core Cassandra server, which provides persistence, replication, elasticity, and other database services. Cassandra's primary configuration comes from the `cassandra.yaml` file. Cassandra stores data in various data and log files.

A minimal deployment consists of a single Doradus instance and a single Cassandra instance. Both can execute on the same node. For a minimal level of protection, the data and log files should reside on mirrored (RAID 1) disks. This is depicted below:



Single Node Configuration

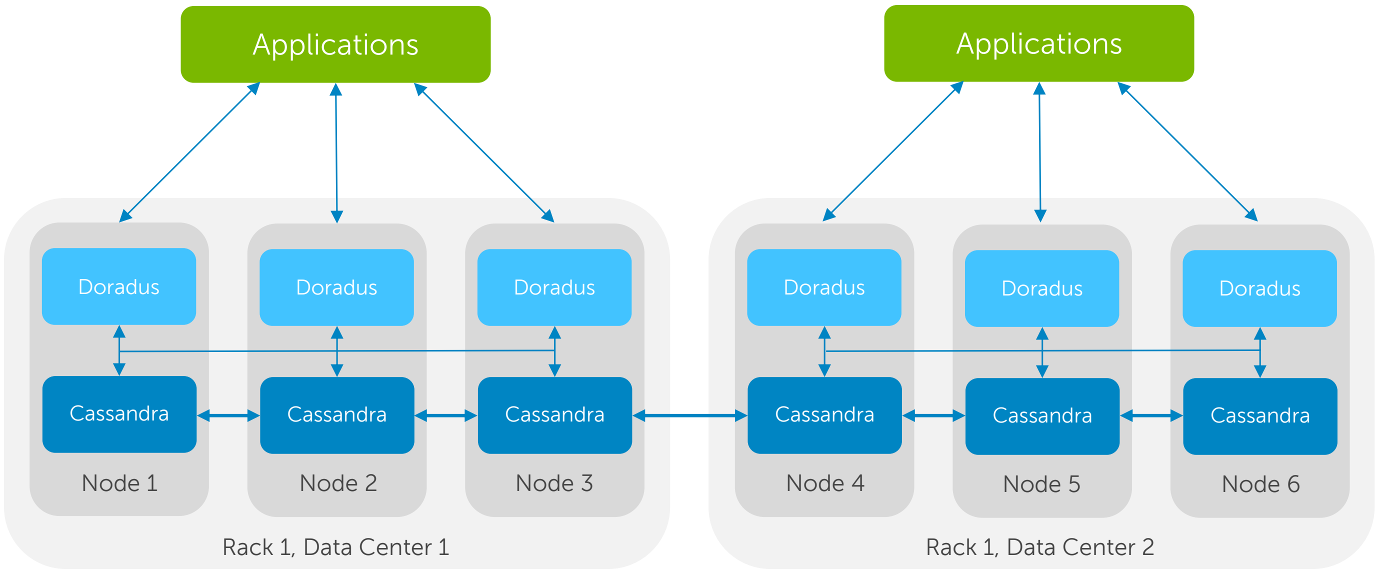
A more robust deployment uses two nodes and a replication factor of 2 (RF=2). Although only one Doradus instance is required, running Doradus on both nodes provides complete failover in case either node fails. This configuration is illustrated below:



Fully Redundant 2 Node Cluster: RF=2

When multiple Doradus instances are used in the same cluster, they are *peers*: any request can be sent to any instance. If a node fails, applications can redirect requests to any available instance. Doradus instances also communicate with each other to distribute background worker tasks and coordinate schema changes.

Cassandra also supports multi-data center deployments. Each Cassandra node is configured with a specific *rack* and *data center* assignment. A rack is usually network-near to other racks in the same data center but independently powered. Data centers are geographically dispersed. With rack and data center awareness, Cassandra can use a replication policy that ensures maximum availability should a node, rack, or entire data center fail. An example of a multi-rack/multi-data center deployment is shown below, using the same model of deploying a Doradus instance on each node:



Multi-Data Center Deployment

Advanced configurations such as these are beyond the scope of this document, but information about Cassandra configuration is publicly available.

5.2. Hardware Recommendations

Server class hardware is recommended for Doradus cluster nodes, though "commodity x86" based servers are sufficient, as opposed to high-end servers. The recommended hardware for each node is summarized below:

- Dual quad core Xeon processors with HT (8 cores, 16 virtual CPUs)
- 32GB of memory
- 2 x 1Gbit NICs
- 4 x hard disks (or more)

Software configuration for effectively using CPUs is described later. Two NICs are recommended, configured as follows:

- Internode traffic: Each node should have one NIC dedicated for traffic to other nodes. Cassandra will use this connection for replication and coordination. This NIC should be given a static IP address since each node will be configured to know the IP address of other nodes.

- External traffic: The other NIC should be dedicated for outbound traffic with applications.

A minimum of 4 hard disks are recommended, used as follows:

- Disk 1: Operating system and software (i.e., C: drive)
- Disk 2: Cassandra commit log
- Disks 3 and 4: Cassandra data files (2)

The software and commit log disks do not have to be large. The size and number of disks used for Cassandra data files depend on the volume of data expected. A minimum of two disks is recommended to allow parallel I/Os and improved bandwidth. When more data capacity is required for a node, added disks should be allocated to additional data files. Because resiliency is provided by internode replication, configuring disks with RAID is unnecessary.

Virtualized hosts are possible but some Cassandra experts suggest that virtualization adds a 5-15% performance penalty.

5.3. Security Best Practices

A summary of best practices for securing Doradus and Cassandra configuration files, protocols, and data is provided below:

- **Doradus REST API**: The REST API can be configured to use TLS (SSL), and it can restrict connections to those that provide a specific client-side certificate.
- **Doradus JMX API**: The JMX API can be secured with authentication and/or TLS.
- **Doradus configuration files**: The primary Doradus configuration file is `doradus.yaml`. This file and the `log4j.properties` file are stored in the folder `{doradus_home}/config`. These files should be considered sensitive and secured from unauthorized access.
- **Cassandra API**: There are two application protocols for Cassandra: Thrift and CQL. Doradus can use either protocol and supports TLS for both protocols.
- **Cassandra JMX API**: The Cassandra JMX protocol can be configured to require authorization and/or to encrypt data using TLS.
- **Cassandra Gossip API**: This is inter-node communication protocol can be configured to use TLS for encryption. However, because of the high-volume nature of this protocol, encryption is not recommended except for cross-data center communication.
- **Cassandra configuration files**: The primary Cassandra configuration file is `cassandra.yaml`. This file and other secondary configuration files are stored in the folder `{cassandra_home}/conf`. These files should be considered sensitive and secured from unauthorized access.
- **Cassandra data files**: All application data is stored in Cassandra data files, which reside in various folders as described in the section Setting Cassandra Data File LocationsError! Reference source not found.. The data in these files is unencrypted and should be secured from unauthorized access.

6. Multi-Tenant Configuration

Doradus can be configured to support multiple *tenants*, which can be thought of as independent customers sharing the same cluster. Each tenant defines its own applications, whose data is physically separated from other tenants. User applications access specific tenants with tenant-specific credentials.

The procedures for configuring and managing multi-tenant features are described below.

6.1. Tenant Mapping to Cassandra

Each tenant is mapped to a Cassandra keyspace, meaning a keyspace is created using the tenant name. All data and metadata are stored in ColumnFamilies within the keyspace. Three ColumnFamilies are shared by all applications owned by the tenant:

- `Applications`: This ColumnFamily holds metadata for all applications owned by the tenant including schemas and, for non-default tenants, tenant options.
- `Tasks`: This ColumnFamily holds status and synchronization records used by the Doradus Task Manager service to perform background tasks such as data aging.
- `OLAP`: This ColumnFamily holds data for all Doradus OLAP applications, if any, owned by the tenant.

If the tenant owns any Doradus Spider applications, data is stored in application-specific ColumnFamilies. Two kinds of ColumnFamilies are currently used for Spider applications:

- *`application_table`*: This ColumnFamily holds object data for a specific Spider application and table.
- *`application_table`*`_terms`: This ColumnFamily holds indexing data for a specific Spider application and table.

Each tenant possesses one or more user IDs that can be used to access applications. Each user ID is mapped to a Cassandra user, prefixed with the tenant name. For example, if the tenant named `HelloKitty` owns a user called `Katniss`, the corresponding Cassandra user ID is `HelloKitty_Katniss`. This allows different tenants to independently define the same user ID. In Doradus REST commands, the tenant user ID (not the Cassandra user ID) and password are provided using basic auth.

Each tenant can also define the replication factor (RF) to use for its applications. This value cannot exceed the number of underlying Cassandra nodes. RF values greater than 1 ensure that data is replicated to 2 or more nodes, thus providing high availability in case of node failure.

6.2. Single-Tenant Operation

By default, Doradus operates in *single-tenant* mode, which means all applications are created in a single keyspace with a default name. Single-tenant mode is affected primarily by two doradus.yaml file options:

```
multitenant_mode: false
keyspace: 'Doradus'
```

When `multitenant_mode` is `false`, all applications are stored in the *default* keyspace defined by the `keyspace` option.

Single-tenant mode does not require security to be enforced within Cassandra. However, you can configure Cassandra to enforce user/password authentication by setting the following options in each Cassandra node's `cassandra.yaml`:

```
authenticator: PasswordAuthenticator
authorizer: CassandraAuthorizer
```

When Cassandra first starts with these options, it creates a default *super user* account with the user ID and password `cassandra`/`cassandra`. You should change the super user ID and/or password to something more secure. Doradus must use a super user account for access, hence set the following doradus.yaml options to a valid super user account:

```
dbuser: cassandra
dbpassword: cassandra
```

You can also pass these options into Doradus as runtime as arguments "`-dbuser xxx -dbpassword yyy`".

6.3. Multi-Tenant Operation

Multi-tenant operation is enabled by setting `multitenant_mode` to `true`. Multi-tenant mode requires the use of the CQL API, hence `use_cql` must also be set to `true`, and `dbport` should be set to Cassandra's CQL port. Finally, multi-tenant mode requires Cassandra to enforce user ID/password authentication, and Doradus must be configured to use a super user. Therefore, enabling multi-tenant mode requires setting all of the following doradus.yaml options:

```
multitenant_mode: true
use_cql: true
dbport: 9042
dbuser: cassandra
dbpassword: cassandra
```

As described in the previous section, you can pass `dbhost` and `dbpassword` as runtime arguments to prevent storing them in the doradus.yaml file.

When multi-tenant mode is enabled, all application-specific REST commands that do not identify a specific tenant are directed to the default tenant. Such commands do not require tenant credentials, hence the default tenant can be considered a "public" tenant. You can disable the default tenant by setting the following doradus.yaml option:

```
disable_default_keyspace: true
```

When this option is set, all application-specific REST commands must be directed to a specific tenant and provide valid credentials for that tenant.

When a new tenant is defined, Doradus creates the corresponding Cassandra keyspace using the following doradus.yaml file options as a template:

```
ks_defaults:
    - strategy_class: SimpleStrategy
    - strategy_options:
        - replication_factor: "1"
    - durable_writes: true
```

This options can be changed to use different defaults for new keyspaces. In a multi-node cluster, replication_factor should be increased to ensure data availability in case of a node failure. Note a tenant can be creating while overriding some of these default keyspace options.

6.4. Managing Tenants

The REST commands to create, modify, and delete tenants are considered privileged operations, hence they must be accompanied with valid super user credentials. REST commands use *basic authorization* (basic auth), which means credentials are passed using the `Authorization` header:

```
Authorization: Basic xxx
```

Where xxx is the user ID and password, separated by a colon, and then base64 encoded. For example, if the super user and password are `cassandra:cassandra`, the header would look like this:

```
Authorization: Basic Y2Fzc2FuZHJhOmNhc3NhbmRyYQo=
```

A summary of tenant-management commands is given below:

Command	REST URI	Comments
Create new tenant	POST /_tenants	• Requires <tenant> document. • Fails if tenant already exists.
Modify tenant	PUT /_tenants/{tenant}	• Requires <tenant> document.
Delete tenant	DELETE /_tenants/{tenant}	• All applications and data are deleted. • ColumnFamilies are moved to “snapshot” files.
List all tenants	GET /_tenants	• Lists tenant and application names only.
List tenant	GET /_tenants/{tenant}	• Lists tenant details.

The following sections describe each command in detail.

6.4.1. Creating a New Tenant

A new tenant is created with the REST command:

```
POST /_tenants
```

This command must include a message body that minimally identifiers the new tenant’s name. Optionally, the command can define one or more tenant users and/or tenant options. An example in XML is shown below:

```
<tenant name="HelloKitty">
   <options>
      <option name="ReplicationFactor">3</option>
   </options>
   <properties>
      <property name="Owner_Organization">Cats, Inc.</property>
   </properties>
   <users>
      <user name="Katniss" password="Everdeen"/>
      <user name="Asparagus" password="Gus">
         <permissions>APPEND</permissions>
      </user>
      <user name="Skimbleshanks" password="Skanks">
         <permissions>READ</permissions>
```

```
        </user>
    </users>
</tenant>
```

In JSON:

```
{"HelloKitty": {
    "options": {
        "ReplicationFactor": "3"
    },
    "properties": {
        "Owner_Organization": "Cats, Inc."
    },
    "users": {
        "Katniss": {
            "password": "Everdeen"
        },
        "Asparagus": {
            "password": "Gus",
            "permissions": "APPEND"
        },
        "Skimbleshanks": {
            "password": "Skanks",
            "permissions": "READ"
        }
    }
}}
```

Only the new tenant name (`HelloKitty`) is required. Options, properties, and/or users can also be defined with the following semantics:

- `options`: This group sets system-defined tenant options. Currently, the only option supported is `ReplicationFactor`, which overrides the `replication_factor` defined within the `ks_defaults` parameter in `doradus.yaml`. This option sets the replication factor for data stored within the tenant. It should not be larger than the number of nodes in the underlying Cassandra cluster.

- `properties`: This group is for storing information about the tenant such as company, organization, or license keys. Any property name and value can be used, however, property names cannot begin with an underscore (_).

- `users`: This group is used to define an initial set of users. If this group is not present, an initial user is created with a random user ID and password and `ALL` permissions. When users are defined, a password must be provided. Optionally, a permissions can be specified, which is a comma-separated list of permission mnemonics. If no permissions are defined, `ALL` is assigned by default. Currently, the following permissions are supported:

    - `ALL`: This permission allows the user to perform all commands on all applications within the tenant.

    - `APPEND`: This permission restricts the user to POST commands.

    - `UPDATE`: This permission allows the user to perform PUT and POST commands.

    - `READ`: This permission allows the user to perform GET commands.

When a `POST /_tenants` command is received, the following occurs:

- A new tenant is created with the given name.
- If no tenant users are specified, a new user is created for the tenant with a random user ID, random password, and `ALL` permissions.
- A new Cassandra keyspace is created using the tenant name. If the option `ReplicationFactor` is specified, it is used for the new keyspace. All other defaults come from the `doradus.yaml` parameter `ks_defaults`.
- CQL commands are issued to create all new tenant user IDs/passwords. Each Cassandra user is named `{tenant}_{user}` so that user names are unique across tenants. The Cassandra user can be used for direct access to Cassandra.
- The keyspace is initialized with Doradus metadata ColumnFamilies (`Applications` and `Tasks`).
- The tenant's definition is written to the `Applications` CF so it can be recovered if needed.

The POST command returns the new tenant's full definition, including name; user IDs, passwords, and permissions; properties; and options, using the same document format shown above. The response also includes the system-assigned `_CreatedOn` property, which is a timestamp documenting when the tenant was created. Example:

```
<tenant name="HelloKitty">
   <options>
      <option name="ReplicationFactor">3</option>
   </options>
   <properties>
      <property name="Owner_Organization">Cats, LLC</property>
      <property name="_CreatedOn">2015-06-17 15:41:40</property>
   </properties>
   <users>
      <user name="Katniss" password="Everdeen">
         <permissions>APPEND</permissions>
      </user>
      <user name="Asparagus" password="Gus">
         <permissions>APPEND</permissions>
      </user>
      <user name="Skimbleshanks" password="Skanks">
         <permissions>READ</permissions>
      </user>
   </users>
</tenant>
```

If the tenant is modified, the `_CreatedOn` property can be specified but must match the existing value.

If a create tenant command is received but the given tenant name already exists, a `409 Conflict` response is returned. This prevents the accidental creation of a new tenant using a name already used.

6.4.2. List All Tenants

This command lists all existing tenants:

```
GET /_tenants
```

This command returns each tenant's name and the list of applications owned by each tenant. An example response in XML is shown below:

```
<tenants>
   <tenant name="Doradus">
      <applications>
         <value>SmokeTest</value>
         <value>App_Default</value>
      </applications>
   </tenant>
   <tenant name="Bibliotecha">
      <applications>
         <value>App_Biblio</value>
      </applications>
   </tenant>
</tenants>
```

As with all REST commands, a JSON response can be requested by adding `?format=json` to the URI. Example:

```
GET /_tenants?format=json
```

Which returns:

```
{"tenants": {
   "Doradus": {
      "applications": [
       "SmokeTest",
       "App_Default"
      ]
   },
   "Bibliotecha": {
      "applications": [
       "App_Biblio"
      ]
   }
}}
```

6.4.3. List Tenant

Details of a specific tenant can be listed with the following REST command:

```
GET /_tenants/{tenant}
```

Where {tenant} is the tenant name. This command returns all tenant details includes its name, properites, options, and all tenant user IDs, passwords, and permissions. An example in XML is shown below:

```
<tenant name="Bibliotecha">
   <properties>
      <property name="_CreatedOn">2015-06-17 15:51:20</property>
   </properties>
   <users>
      <user name="Up0ipvcy6ogtp" password="l1ysppf2l6zu">
         <permissions>ALL</permissions>
      </user>
   </users>
</tenant>
```

6.4.4. Modify Tenant

An existing tenant can be modified via the following REST command:

```
PUT /_tenants/{tenant}
```

Where {tenant} is the tenant name. All changes are allowed including adding, modifying, and deleting properties, options, and users. (However, the value of the `_CreatedOn` property cannot be changed.) The command must include an XML or JSON document with the same format used to create a tenant. The tenant document is interpreted as follows:

- Any options, properties, or users not in the current tenant definition are added.
- Any options, properties, or users in the current tenant definition but not repeated in the modified tenant definition are deleted. However:
  - The `_CreatedOn` property cannot be deleted.
  - If the `ReplicationFactor` option is deleted, the keyspace's replication factor is not modified.
- Existing options, properties, and users can be modified by specifying a new option value, property value, user password, etc.
  - If the `ReplicationFactor` option is modified, the keyspace's replication factor modified accordingly. This action should be followed by the administrative command "nodetool repair <keyspace>" to redistribute data according to the keyspace's new replication factor.
- An existing user's permissions can be modified without respecifying the user's current password. In this case, the password is not changed.

Example:

```
{"HelloKitty": {
    "options": {"ReplicationFactor": "3"},
    "properties": {"Owner_Organiztion": "Cats, Inc."},
    "users": {
        "Asparagus": {"permissions": "APPEND"},
        "Skimbleshanks": {"permissions": "READ"},
        "Katniss": {"password": "MockingJay", "permissions": "ALL"},
        "Rumpleteazer": {"password": "Mongojerrie"}
    }
}}
```

In this example, the password for user `Katniss` is modified and a new user `Rumpleteazer` is added.

6.4.5. Delete Tenant

An existing tenant can be deleted via the following REST command:

```
DELETE /_tenants/{tenant}
```

Where {tenant} is the tenant name. No message body is provided in the command. The tenant's keyspace, all of its applications, and of its data are deleted. (Note that Cassandra creates a snapshot of a CF when it is deleted, so the data is still recoverable for a while.)

6.5. Application Commands in Multi-Tenant Mode

Application REST commands are those that define or modify application schemas, update data, query data, or perform other application-specific commands supported by the managing storage service. When Doradus is operating in multi-tenant mode, all such commands must identify the target tenant by appending `?tenant={tenant}` to the URI. For example, the following commands are directed to the tenant named `HelloKitty`:

Examples:

```
POST /_applications?tenant=HelloKitty       // create a new application
POST /foo/bar?tenant=HelloKitty             // add data to application foo
GET /_tasks?tenant=HelloKitty               // list tasks for all applications
GET /_olapp?tenant=HelloKitty               // display the OLAP browser
```

If no `?tenant` parameter is provided, the command is directed to the default tenant and no credentials are required. (Note that access to the default tenant may be disabled if the option `disable_default_keyspace` is set to `true`.) When a REST command uses other URI query parameters, the `tenant` parameter can be given anywhere in the query string. URI query parameters are separated by the ampersand (&). The following commands are equivalent, sending a query to the application `foo` in the `HelloKitty` tenant:

```
GET /foo/bar/_query?q=*&tenant=HelloKitty
GET /foo/bar/_query?tenant=HelloKitty&q=*
```

Tenant-specific application commands must be accompanied with valid credentials for that tenant. Credentials are passed using a *basic authorization* (basic auth) header `Authorization`. Example:

```
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.

Note that super user credentials can be used to access any tenant, hence they must be kept secured.

6.6. System Commands in Multi-Tenant Mode

REST commands that are not application-specific are called *system commands* and are not directed to a specific tenant and hence do not use the `?tenant` parameter. This includes the following URIs:

```
/_tenants/...                        // Tenant manager command
/_logs                               // Dump logs command
/_dump                               // Dump threads command
```

In multi-tenant mode, system commands must also be authenticated via basic auth using super user credentials.

7. Doradus Configuration and Operation

This section provides recommendations for runtime and configuration options for the Doradus server.

7.1. Setting Runtime Memory

Doradus is generally stateless. Most of the memory it requires are for the following scenarios:

1. Large update batches, which must be parsed and held in memory until written to the database.
2. Queries that accumulate large result sets, which must be aggregated or sorted before returning.
3. Immutable query results, such as OLAP query clauses and data arrays, which are cached on an LRU basis for "hot" data.

For most cases, Doradus is happy with 1 to 4 GB of memory. Since tests have shown that *growing* a Java process's memory can be very disruptive, so set both the minimum and maximum memory parameters to the same value. Example:

```
java -Xms4G -Xmx4G -cp ... com.dell.doradus.core.DoradusServer
```

7.2. Setting Doradus Logging Options

Doradus logs messages about its operation using the log4j logging facility. Logging options are defined in the file `log4j-server.properties`. The high-level logging options are shown below:

```
log4j.rootLogger=DEBUG, console, file, memory
log4j.logger.org.eclipse.jetty=INFO
log4j.appender.console.Threshold=INFO
log4j.appender.file.Threshold=INFO
log4j.appender.memory.Threshold=DEBUG
```

This sets the global logging level to DEBUG and defines multiple "appenders": console (stdout), a disk file, and memory. Although DEBUG-level log records are generated, only INFO-level entries (and above) are sent to the console and file appenders. DEBUG entries are only collected by the memory appender. This minimizes the output to the console- and file-based logs, but it allows the most recent DEBUG records to be retrieved via the REST command: `GET /_logs`.

If Doradus is run as a service, the output to console is unnecessary, so this parameter should be removed. Additionally, the lines that begin with `log4j.appender.console` should be deleted or commented-out.

Because the memory appender only retains the most recent log entries, in diagnostic situations, you might want to capture DEBUG entries in the file log as well. Note that this causes log files to grow more quickly. DEBUG entries can be captured in the file appender by changing the following line:

```
log4j.appender.console.Threshold=DEBUG
```

The `log4j-server.properties` file also defines the following option:

```
log4j.appender.file.File=doradus.log
```

This option sends the file log appender to a log file called `doradus.log` in the Doradus server's runtime directory. If you wish log files to be stored with another name and/or in another location, modify this option with a new relative or absolute file name.

Two other options to consider:

```
log4j.appender.file.maxBackupIndex=50
log4j.appender.file.maxFileSize=20MB
```

These options cause the log file to grow up to 20MB, at which point it is renamed with a numeric extension (.1, .2, etc.) and a new file is started. Up to 50 files are retained (up to 1GB total log data), at which point the oldest file is removed. Modify these options to increase or decrease the number and size of log files stored.

7.3. Securing the Doradus REST API

By default, the Doradus REST API uses unencrypted HTTP. Because Doradus provides no application-level security, any process that connect to the Doradus REST port is allowed to perform all schema, update, and query commands. The REST API can be secured by enabling TLS (SSL), which encrypts all traffic and uses mutual authentication to restrict access to specific clients. Optionally, client authentication can be enabled to restrict connections to only those whose certificates have been registered at the server. The process for securing the REST port with TLS is defined below:

1) Enable TLS by setting the `tls` parameter in the `doradus.yaml` file to true. Example:

   ```
   tls: true
   ```

2) Create a certificate for use by the Doradus server and store it in a keystore file. You can use the `keytool` utility included with the JRE. An overview of the process to create a self-signed certificate is outline here:

   http://docs.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html#CreateKeystore

3) Set the `keystore` parameter in the `doradus.yaml` file to the location of the **keystore** file, and set the `keystorepassword` parameter to the file's password. Example:

   ```
   keystore: config/keystore
   keystorepassword: mykspassword
   ```

4) If client authentication will be used, create a certificate for each client application and import them into a **truststore** file. (See the same article referenced above.) Set the `truststore` parameter in the `doradus.yaml` file to the location of the keystore file, and set the `truststorepassword` parameter to the file's password. Example:

   ```
   truststore: config/truststore
   truststorepassword: mytspassword
   ```

5) To require client authentication, set the `clientauthentication` parameter in the `doradus.yaml` file. This requires REST API connections to use mutual authentication. Example:

   ```
   clientauthentication: true
   ```

6) The cipher algorithms allowed by the REST API when TLS is enabled is controlled via the `tls_cipher_suites` parameter. The default list includes the algorithms recommended for FIPS

compliance. The actual algorithms allowed by REST API is a subset of the listed algorithms and those actually available to the JVM in which Doradus is running. The cipher algorithm list can be tailored, for example, to only allow 256-bit symmetrical encryption. Example:

```
tls_cipher_suites:
   - TLS_DHE_DSS_WITH_AES_256_CBC_SHA
   - TLS_DHE_RSA_WITH_AES_256_CBC_SHA
   - TLS_RSA_WITH_AES_256_CBC_SHA
   - TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
   - TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
   - TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA
   - TLS_ECDH_RSA_WITH_AES_256_CBC_SHA
```

Custom algorithms can also be used as long as installed with the JVM used to run Doradus.

With the steps above, Doradus will use TLS for its REST API port, optionally requiring mutual authentication. Clients must connect to the REST port using TLS. If client authentication is enabled, they must submit a certificate that was added to the truststore. Each client must also support one of the configured cipher algorithms.

7.4. Securing the Doradus JMX API

Doradus supports the Java Management Extensions (JMX) protocol for monitoring Doradus and performing certain administrative commands. JMX can be secured with authorization and/or encryption, however, security JMX security is disabled in the `{doradus_home}/bin/doradus-server.bat` file included with Doradus. The common JMX options are defined in variables at the top of the file as shown below:

```
set JMX_PORT=9999
set JMX_SSL=false
set JMX_AUTHENTICATE=false
```

These options are summarized below:

- **`JMX_PORT`**: This variable sets the `com.sun.management.jmxremote.port` define in the JVM, which is the port number that the remote JMX clients must use. As shown, port 9999 is the default used for Doradus.

- **`JMX_SSL`**: This variable sets the `com.sun.management.jmxremote.ssl` define in the JVM. When set to `true`, it requires remote clients to use SSL (TLS) to connect to the JMX port. When SSL is enabled, additional options are available to require remote clients to have a client-side certificate.

- **`JMX_AUTHENTICATE`**: This variable sets the `com.sun.management.jmxremote.authenticate` define in the JVM. When set to true, it requires remote clients to provide a user ID and password in order to connect. Additional parameters are required to define allowed user IDs and passwords and the locations of the corresponding files.

Because JMX is a standard and external documentation is available for securing JMX, details for using SSL and/or authentication are not covered here. See for example the following:

http://docs.oracle.com/javase/6/docs/technotes/guides/management/agent.html

Though not shown above, here is another useful option: On multi-homed systems, `java.rmi.server.hostname` can be set to a specific IP address, causing JMX to bind to that address instead of the default one. For example:

```
set JAVA_OPTS=
...
 -Djava.rmi.server.hostname=10.1.82.121^
...
```

This causes the JMX port to bind to address 10.1.82.121.

7.5. Configuring Doradus for Clusters

Multiple instances of Cassandra and/or Doradus can be used in the same cluster. As described in the section Deployment Guidelines, a recommended practice is to deploy an instance of both Doradus and Cassandra on each node. Then, each Doradus instance should be configured to contact all Cassandra nodes. This provides load balancing and fault tolerance in case a Cassandra node fails.

The Cassandra instance(s) to which each Doradus instance connects is defined in the `dbhost` parameter in its `doradus.yaml` file. This parameter should be set to a comma-separated list of host names or (preferably) IP addresses of Cassandra instance hosts. For example:

```
dbhost: 10.1.82.146, 10.1.82.147, 10.1.82.148
```

How this parameter is used depends on whether Thrift or CQL is used:

7.5.1. Thrift API

When Thrift is used (`use_cql` is `false`), all hosts specified in dbhost are used in a round-robin manner. If a host fails, it is skipped and periodically retried. This achieves load balancing across a set of servers.

Alternatively, you can set dbhost to a primary Cassandra node or set of nodes and failover to a secondary list of nodes only the primary node(s) are all unavailable. To do this, specify one or more Cassandra nodes using the `secondary_dbhost` parameter. Example:

```
dbhost: 10.1.82.146
secondary_dbhost: 10.1.82.147, 10.1.82.148
```

This example tells Doradus to use 10.1.82.146 exclusively (perhaps because network-near) and failover to 10.1.82.147 and 10.1.82.148 only if the primary node becomes available.

7.5.2. CQL API

When CQL is used (`use_cql` is `true`), all Cassandra nodes are automatically discovered and used. However, `dbhost` can still be set to multiple nodes in case one of them is down at the time of startup.

7.6. `doradus.yaml` Configuration Parameters

This section describes the parameters that can be set in the `doradus.yaml` file.

7.6.1. Multi-tenant Mode Parameters

See the section Multi-Tenant Configuration for details on configuring and operating Doradus single- and multi-tenant modes. Below is a reference of the parameters that affect multi-tenant operation:

```
multitenant_mode: false
```

This parameter controls whether Doradus operates in single-tenant mode (false) or multi-tenant mode (true).

```
disable_default_keyspace: false
```

When Doradus is operating in multi-tenant mode, this parameter controls whether access to the default tenant is allowed.

```
keyspace: Doradus
```

This parameter defines the name of the default tenant.

```
ks_defaults:
    - strategy_class: SimpleStrategy
    - strategy_options:
        - replication_factor: "1"
    - durable_writes: true
```

This parameter group is used when creating both the default tenant and new tenants in multi-tenant mode.

Note that when multi-tenant mode is enabled, `use_cql` must be set to `true`, and `dbuser` and `dbpassword` must be set to the credentials for a Cassandra super user account. These parameters are described later.

7.6.2. default_services Parameter

Default services define the internal Doradus services that will be started when Doradus is run in stand-alone mode. (When Doradus is started in embedded mode, the default services are passed to the `DoradusServer.startEmbedded()` method.) Example:

```
default_services:
    -com.dell.doradus.mbeans.MBeanService
    -com.dell.doradus.service.db.DBService
    -com.dell.doradus.service.rest.RESTService
    -com.dell.doradus.service.schema.SchemaService
    -com.dell.doradus.service.taskmanager.TaskManagerService
```

These services are in addition to storage services, which are defined separately. Note that `DBService` and `SchemaService` are required and always started even if not listed here. Other services are optional and can be disabled when not needed. Custom services can also be added to this list. Each line must contain the full package name of the service class. The optional services are:

```
com.dell.doradus.mbeans.MBeanService
```

Provides monitoring and management services via custom Doradus MBeans. Can be disabled if custom JMX commands are not needed.

```
com.dell.doradus.service.rest.RESTService
```

Provides the REST API via an embedded Jetty server, which listens to the server and port defined by `restaddr` and `restport`. It only makes sense to disable this when running Doradus as an embedded service.

```
com.dell.doradus.service.taskmanager.TaskManagerService
```

Provides background task management for data aging, statistics refreshing (Spider), and other scheduled tasks. Also manages on-demand tasks such as removing obsolete data (Spider). Can be disabled if background tasks are not needed for this Doradus instance.

7.6.3. `storage_services` Parameter

The `storage_services` parameter defines which storage services are initialized when Doradus is started as a standalone application (e.g., as a service or console app). At least one storage service must be defined. The first storage service listed becomes the default for new applications created without explicitly declaring a storage service. Example:

```
storage_services:
    - com.dell.doradus.service.spider.SpiderService
    - com.dell.doradus.service.olap.OLAPService
```

This initializes both the Spider and OLAP services, making the Spider service the default for new applications.

7.6.4. `cf_defaults` Parameters

The `cf_defaults` parameter is used when Doradus creates a new ColumnFamily. All values recognized by Cassandra are allowed and are passed "as is". Parameters must begin with a dash (-), and sub-parameters should be further indented. Empty values should be specified with a pair of quotes. Example:

```
cf_defaults:
    - compression_options:
        - sstable_compression: ""   # use empty string for "none"
    - gc_grace_seconds: 3600
```

After a ColumnFamily is created, Doradus does not modify its parameters to match values in `doradus.yaml`. It is safe to use Cassandra tools such as `cassandra-cli` to modify ColumnFamily parameters when needed.

See Cassandra documentation for information about ColumnFamily parameters.

7.6.5. Cassandra Connection Parameters

These parameters control connections to the Cassandra database using the Thrift or CQL APIs:

```
dbhost: 'localhost'
```

This parameter configures the host name(s) or address(es) of the Cassandra node(s) that the Doradus server connect to. Static IP addresses are preferred over host names. If a comma-separated list of hosts is provided, Doradus will connect to each one in round-robin fashion, allowing load balancing and failover.

```
dbport: 9160
```

This is the Cassandra Thrift or CQL API port number. Change this value if Cassandra's port is using something other than 9160. (Cassandra should use the same port on all nodes.)

`jmxport: 7199`

Change this value if Cassandra's JMX port is using something other than 7199. (Cassandra should use the same port on all nodes.)

`db_timeout_millis: 60000`

This is the socket-level Cassandra database connection timeout in milliseconds. It is applicable to both the Thrift and CQL APIs. This time is used both when connecting to Cassandra and when performing read/write operations. If a connect, read, or write request is not received within this time, the connection is closed and a retry is initiated. Note, however, that Cassandra has its own RPC timeout value, which defaults to 10 seconds. This means that read and write operations will typically based on Cassandra's value and not this one.

`db_connect_retry_wait_millis: 30000`

This is the Cassandra database (Thrift) initial connection retry wait in milliseconds. It is applicable to both the Thrift and CQL APIs. If an initial connection to Cassandra times-out (see `db_timeout_millis` above), Doradus waits this amount of time before trying again. Doradus keeps retrying indefinitely under the assumption that Cassandra may take a while to start.

`max_commit_attempts: 10`

This is the maximum number of times Doradus will attempt to perform a Cassandra update operation before aborting the operation. (See also `retry_wait_millis`).

`max_read_attempts: 3`

This is the maximum number of times Doradus will attempt to perform a read operation before aborting the operation. (See also `retry_wait_millis`).

`max_reconnect_attempts: 3`

This is the maximum number of times Doradus will attempt to reconnect to Cassandra after a failure. A reconnect is performed if a read or update call fails. (See also `retry_wait_millis`).

`retry_wait_millis: 5000`

This is the time in milliseconds that Doradus waits after a failed update, read, or reconnect request call to Cassandra. This time is multiplied by the attempt number, which means Doradus waits a little longer after each successive failed call. (See also `max_commit_attempts`, `max_read_attempts`, and `max_reconnect_attempts`.)

`use_cql: false`

By default, Doradus uses the Thrift API to communicate with Cassandra. When this option is set to `true`, it uses the CQL API instead. CQL must be enabled in each Cassandra instance that Doradus connects via the `cassandra.yaml` file option `start_native_transport: true`. Additionally, the `doradus.yaml` file option `dbport` must be set to the CQL API's port (the default is `9042`).

```
dbtls: false
```

Set to true to use TLS/SSL for connections to Cassandra. This option works for both CQL and Thrift. When `dbtls` is enabled, the options `keystore` and `keystorepassword` are used for certificate information.

```
dbtls_cipher_suites: [TLS_RSA_WITH_AES_128_CBC_SHA]
```

Set to a list of one or more cipher suites to be used with SSL/TLS to Cassandra. This option is only meaningful if `dbtls` is set to `true`. Cassandra must be configured to allow at least one of the same cipher algorithms in order to establish a successful connection. The list can be comma-separated on a single line within square brackets (e.g., [X, Y, Z]), or each cipher suite can be listed on its own line, indented and prefixed with a "-". The cipher suite `TLS_RSA_WITH_AES_128_CBC_SHA` is generally always allowed by Cassandra.

```
dbuser: cassandra
dbpassword: cassandra
```

If authentication/authorization is configured for Cassandra, these parameters must be set to a super user that Doradus can use. When Cassandra's authenticator is set to `PasswordAuthorizer`, it initially creates the super id/password `cassandra`/`cassandra`. A better password and/or a different super user should be created for Doradus. Remember that these values can be passed-in dynamically as run-time arguments. The `dbuser` and `dbpassword` parameters can be used for both the Thrift and CQL APIs and for both single- and multi-tenant operation.

7.6.6. Doradus REST API Parameters

These parameters configure the Doradus REST API.

```
restaddr: 0.0.0.0
```

By default, the Doradus REST API binds to address 0.0.0.0, which means it accepts connections directed to any of the host's IP addresses (including "localhost"). Change this value if you want the REST API to accept connections on a specific IP address instead.

```
restport: 1123
```

This is the REST port that Doradus listens to. Change this to accept REST API connections on a different port.

```
maxconns: 200
```

This is parameter is used to configure the maximum threads that the internal Jetty server allows. More than this number of connections may be received, but this value controls the maximum number of parallel REST commands that may be in progress at one time.

```
tls: false
tls_cipher_suites:
   - TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
   ...
clientauthentication: false
keystore: config/keystore
keystorepassword: changeit
truststore: config/truststore
```

```
truststorepassword: password
```

As described in the section Doradus Security Options, these parameters should be set to configure TLS/SSL for the Doradus REST API and to restrict connections to authenticated clients.

```
max_request_size: 52428800
```

This parameter determines the maximum size of an input requested that is accepted. Requests larger than this are rejected and the socket is closed. The default is 50MB.

7.6.7. Doradus Query Parameters

These parameters configure defaults used for Doradus object and aggregate query parsing and execution.

```
search_default_page_size: 100
```

This value is the default query page size used when an object query does not specify the &s (page size) parameter.

```
aggr_separate_search: false
aggr_concurrent_threads: 0
dbesoptions_entityBuffer: 1000
dbesoptions_linkBuffer: 1000
dbesoptions_initialLinkBuffer: 10
dbesoptions_initialLinkBufferDimension: 1000
dbesoptions_initialScalarBuffer: 30
l2r_enable: true
```

These parameters are set by the Doradus team and normally should not be changed.

7.6.8. OLAP Parameters

These parameters control the operation of the OLAP service:

```
olap_cache_size_mb: 100
```

This parameter controls the memory size of the most-recently-used field cache. If this value is exceeded, older fields will be un-loaded from memory.

```
olap_file_cache_size_mb: 100
```

When this value is non-zero, it enables OLAP “file” caching and sets the total size, in megabytes, of the cached data. OLAP uses virtual files to hold raw scalar values such as text values. Caching this data prevents round-trips to the database for certain types of queries. This value does not affect shard caching defined by other parameters.

```
olap_query_cache_size_mb: 100
```

When this value is non-zero, it enables recent query result caching and sets the total size, in megabytes, of the cached search results. Each cached result takes 1 bit per each object in the table.

```
olap_cf_defaults:
    - compression_options:
        - sstable_compression:  "" # use empty string for "none"
    - gc_grace_seconds: 3600
```

This parameter is used to create the OLAP ColumnFamily. All parameters recognized by Cassandra are accepted and passed "as is". Parameters must be indented and begin with a "-"; sub-parameters must be further indented. The OLAP ColumnFamily is created when the server is first started for a new database. If the OLAP ColumnFamily already exists, it is not modified to match these parameters. See Cassandra documentation for details about these values. Note that Doradus automatically compresses data before storing in Cassanda, hence compression should be disabled for the OLAP ColumnFamily.

```
olap_merge_threads: 0
```

This parameter controls the number of threads used to merge data within a shard. The default value of 0 means that a single thread is used to merge all data. When this value is > 0, multiple threads are used to merge shard data in parallel. Increasing this value can significantly decrease the time needed to merge shards. However, it must be balanced with the number of processors available on the machine.

```
olap_compression_threads: 0
```

This parameter controls the number of threads used to compress data before storing in Cassandra. This parameter interacts with `olap_merge_threads` as follows:

- If both `olap_merge_threads` and `olap_compression_threads` = 0, then a single thread is used to merge and store all data for each thread.
- If `olap_merge_threads` > 0 and `olap_compression_threads` = 0, then each merge thread both merges data and compresses all segments before writing to Cassandra.
- If `olap_compression_threads` > 0, then the merge thread(s) create data segments that are then queued for compression in the specified number of asynchronous compression threads.

Setting `olap_compression_threads` > 0 can significantly decrease the time needed to merge large OLAP shards. However, the value used must be balanced with `olap_merge_threads` and the number of cores available on the system.

```
olap_compression_level: -1
```

This parameter controls the GZIP compression level used to compress data in OLAP before storage. The default value, -1, is the same as value 6, which is a good balance between speed and space usage. Value 0 means "no compression", which speeds shard merge time but requires the most amount of disk space. Value 9 means "best compression", which requires the most CPU but uses the least amount of disk space. Tests suggest that levels 2, 4, and 6 are good choices.

```
olap_search_threads: 0
```

This parameter controls the number of threads used to perform multi-shard queries. When this value is > 0, up to the configured number of threads can be used by a single query to search different shards in parallel. Increasing this value can significantly improve performance for certain threads. However, it must be balanced with the number of processors available on the machine.

One parameter worth highlighting is the OLAP ColumnFamily's `gc_grace_seconds`. This value controls how long deleted rows called "tombstones" are retained with data tables (called *sstables*) before they are truly deleted during a compaction cycle. The default is 864,000 seconds or 10 days, which provides lots of time for a failed node to recover and learn about deletions it may have missed. However, the OLAP service deletes many rows when it merges a shard, and these rows consume disk space. Moreover, they consume memory because active sstables are memory-mapped (mmap-ed) by Cassandra. This can cause excessive memory usage.

A much smaller `gc_grace_seconds` value is recommended for the OLAP CF, somewhere between 3600 (a hour) and 86400 (1 day). This causes tombstones to be deleted more quickly, freeing-up disk space and reducing memory usage.

7.6.9. Spider Parameters

These parameters control the operation of the Spider server:

```
batch_mutation_threshold: 10000
```

Defines the maximum number of mutations a batch update will queue before flushing to the database. After the mutations are generated for each object in a batch update, Spider checks to see if this threshold has been met and flushes the current mutation batch if it has. Larger values improve performance by reducing Cassandra round trips, but they require larger API buffer sizes. The default is 10,000.

7.7. Managing Doradus via JMX

Doradus extends the Java JMX API with monitoring and administration features. JMX functions can be accessed with off-the-shelf JMX applications such as JConsole, which is bundled with the Java SDK. JMX functions are available from two `MXBean` objects:

- `com.dell.doradus.ServerMonitor`: This `MXBean` provides information about the Doradus server including its configuration, start time, and statistics about REST commands.

- `com.dell.doradus.StorageManager`: This `MXBean` provides information used by the Doradus Task Manager to conduct background tasks. It also provides operations for modifying tasks and to execute Cassandra backup and recovery tasks.

JMX functions only collect and return information when the Doradus MBean service (`com.dell.doradus.mbeans.MBeanService`) has been initialized. This is always the case when the server is started as a standalone process but optional when Doradus is embedded within another application.

7.7.1. ServerMonitor

`ServerMonitor` provides the following attributes:

Attribute Name	Description
`AllRequests`	Provides a composite set of values about REST API requests such as the total number of requests, the number of failed requests, and the reason of the last request failure.
`ConnectionsCount`	The current number of REST API connections.
`DatabaseLink`	Indicates if the Cassandra database being accessed is local (1), remote (2), or not yet known (0).
`RecentRequests`	A composite value set similar to `AllRequests` but for REST API requests made since the last time `RecentRequests` was accessed.

`ReleaseVersion`	The Doradus Server release version.
`ServerConfig`	Lists configuration doradus.yaml settings, including value overridden by runtime arguments.
`StartTime`	Provides the start time of the Doradus server as a `Date.getTime()` value.
`Throughput`	Provides a histogram of REST API request response times in seconds for various sampling rates (e.g., 1 minute, 5 minutes, 15 minutes.)
`WorkingDirectory`	The working directory of the Doradus server.

7.7.2. StorageManager

`StorageManager` is operative only when the Task Manager service has been initialized (`com.dell.doradus.service.taskmanager.TaskManagerService`). This is always the case when the server is started as a standalone process but optional when Doradus is embedded within another application. `StorageManager` provides the following attributes:

Attribute Name	Description
`AppNames`	The Doradus application names known to the Task Manager.
`GlobalDefaultSettings`	The global task schedule settings, which are used for tasks that do not have a more specific (e.g., application- or table-specific) schedule default. The global schedule is usually "never", meaning no default schedule is applied to tasks.
`JobList`	The list of tasks currently executing.
`NodesCount`	The number of Cassandra nodes in the cluster.
`OS`	The operating system of the Doradus server.
`OperationMode`	The operation mode of the first connected Cassandra mode (NORMAL, LEAVING, JOINING, etc.)
`RecentJob`	Attributes about the most recent job executed through the `StorageManager` interface.
`ReleaseVersion`	The Doradus server release version.
`StartMode`	The execution mode of the first connected Cassandra node: 1 (FOREGROUND), 2 (BACKGROUND), or 0 (UNKNOWN).

The `StorageManager MXBean` also provides operations that perform basic administrative functions, summarized below:

Operation Name	Description
`getJobByID`	Get the status of a job with a specific ID.
`startCreateSnapshot`	Start a task to create a backup snapshot of the first Cassandra node connected to the Doradus server, assigning the snapshot a name.
`startDeleteSnapshot`	Delete a backup snapshot with a given name.
`startRestoreFromSnapshot`	Restore the Cassandra node connected to the Doradus server from a given snapshot game.
`sendInterruptTaskCommand`	Interrupt an executing task with a given name, belonging to a given application.
`sendSuspectSchedulingCommand`	Suspend the scheduling of a task with a given name, belonging to a given application.
`sendResumeSchedulingCommand`	Resume the scheduling of a task with a given name, belonging to a given application.
`getAppSettings`	Get the default task schedule settings for a given application name.
`getTaskStatus`	Get the status of a task with a given name, belonging to a given

	application.