Conditions Database for SHiP

User Manual

Authors:

J.J.M.J van der Heijden
N. D’Penha

ST2019
ST-2019-PI-CERN-UM v1.0 | 15.04.2020

Contents

Document Change Records

Chapter	Author(s)
1. Importing the Conditions Database API	J.J.M.J van der Heijden
2. Working with detectors	J.J.M.J van der Heijden
3. Working with conditions	J.J.M.J van der Heijden
4. Using MongoDB with the Conditions Database	N. D'Penha

Chapter 1: Importing the Conditions Database API

The Conditions Database API has been made extensible by using a Factory to instantiate an API instance. The full high level design is presented in Figure 1. FairSHiP scripts import the *CDB API Factory* and get an instance of an *CDB API* defined by the *CDB API interface.* To get an instance of the *MongoDBToCDBAPIAdapter* we must call the factory with the request for a "mongo" API. The steps to do this are described below.

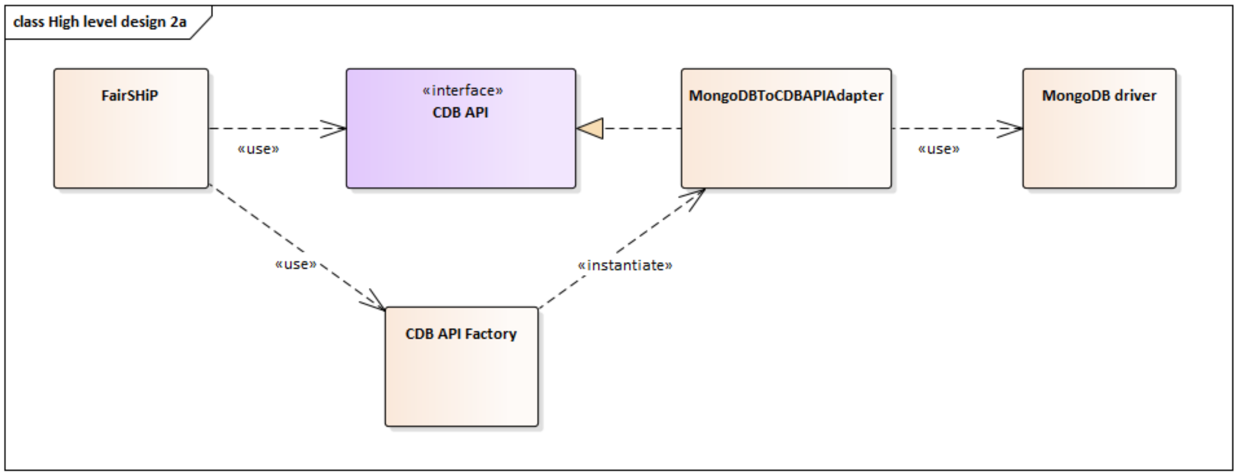
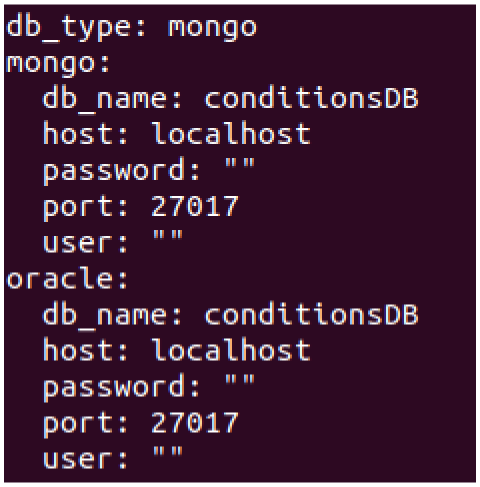


Figure 1 High Level Design of the Conditions Database API

Setup the configuration file

To configure the factory we must create a config.yml file which specifies connection properties for a database instance. A default config.yml file is distributed with the API in the folder **/conditionsDatabase/config.yml.** When opening this file we get these values:



```
db_type: mongo
mongo:
  db_name: conditionsDB
  host: localhost
  password: ""
  port: 27017
  user: ""
oracle:
  db_name: conditionsDB
  host: localhost
  password: ""
  port: 27017
  user: ""
```

First of all, we have the **db_type** this is the type of database for which the factory will return an API object. There can be multiple database types in the config file (in the example here we have mongo and oracle). By setting the **db_type** we choose which of these database types will be returned.

These values should be adapted according to your database location and specification:

db_name: The name of the mongo database
host: The location where the database is hosted (localhost if the database is run locally)
password: User password if database is protected (empty string by default)

port: The port of the mongo database (27017 by default)
user: A username for the mongo database if the database is protected (empty string by default)

Instantiate the factory

To create an instance of the Conditions Database API we must first import the factory from the conditionsDatabase package.

```
from conditionsDatabase.factory import APIFactory
```

Then instantiate the factory.

```
factory = APIFactory()
```

Instantiate the Conditions Database API

With the instantiated factory in the previous step we can now ask for the database instance specified in the config.yml file.

To ask for a database API specified in the **default** conditionsDatabase/config.yml file. We can ask the api_factory to construct a database API without giving parameters

```
conditionsDB = factory.construct_DB_API()
```

Take into account that for calling this method without any parameters the conditionsDatabase package should be within the folder pointed to by $FAIRHOME.

You can also ask the factory to use another specified .yml file. For this specify a string with the path to the config.yml file. In the example below we use the path to the default config.yml location.

```
conditionsDB = factory.construct_DB_API("/FairShip/conditionsDatabase/config.yml")
```

Chapter 2: Working with detectors

This chapter covers managing detector data with the Conditions Database API.

List detectors by name

To get a list of detector names in the Conditions Database, use the API function list_detectors(name = None).

The parameter **name** represents the name of the parent detector where we want to list the subdetectors of. If this name parameter is kept empty it will default to None and will list the main detectors in the system.

To get all main detectors names in the database:

```
result = conditionsDB.list_detectors()
```

To get all subdetectors of detector "detectorX" in the database:

```
result = conditionsDB.list_detectors("detectorX")
```

"detectorX" should be replaced with any valid detector path in the Conditions Database.

Add detector

To add a detector to the Conditions Database, use the API function add_detector(name, parent_id = None).

The parameter **name** represents the name for the new detector.

The parameter **parent_id** should be specified if we want to add the new detector as a subdetector of a certain parent. The parent_id is optional, and if not specified the detector will be added as a main detector.

To add a main detector:

```
conditionsDB.add_detector("detector1")
```

To add a subdetector to detector "detectorX":

```
conditionsDB.add_detector("detector2", "detectorX")
```

"detectorX" should be replaced with any valid detector path in the Conditions Database.

Get detector

To get a detector from the Conditions Database, use the API function get_detector(detector_id). This function returns a dictionary representation of the detector queried.

The parameter detector_id is a string identifying the detector.

To get a main detector:

```
result = conditionsDB.get_detector(“detectorX”)
```

“detectorX” should be replaced with any valid detector path in the Conditions Database.

To get the subdetector “detectorY” of “detectorX”:

```
result = conditionsDB.get_detector(“detectorX/detectorY”)
```

“detectorX/detectorY” should be replaced with any valid detector path in the Conditions Database.

Remove detector

Important: Use this function with care, otherwise data will get lost

To remove a detector from the Conditions Database, use the API function remove_detector(detector_id). Take care that while removing a detector you delete all of it’s conditions and subdetectors as well.

The parameter **detector_id** is a string identifying the detector.

To remove a main detector:

```
conditionsDB.remove_detector(“detectorX”)
```

This removes main detectorX and all of its subdetectors and conditions.

To remove a subdetector:

```
conditionsDB.remove_detector(“detectorX/detectorY”)
```

Chapter 3: Working with conditions

This chapter covers managing the conditions data with the Conditions Database API.

Add conditions

To add a condition to the Conditions Database, use the API function add_condition(detector_id, name, tag, values, type=None, collected_at=datetime.now(), valid_since=datetime.now(), valid_until=datetime.max).

The parameter **detector_id** represents the detector path of the detector this condition should be added to.
The parameter **name** represents the name for the new condition. Must not be unique. E.g "strawPositions".
The parameter **tag** a string identifying the condition tag.
The parameter **values** represent the condition values, they are of a "mixed" type, which means we can put any datatype in values.
The parameter **type** is a string representing the condition type. If unset this will default to None.
The parameter **collected_at** is a timestamp at which the condition is collected. This parameter can be entered as a datetime Object, or a String. If unset, this will default to the current date and time. Certain time accuracy can be omitted, if so the time will be set to 0's.
The parameter **valid_since** is a timestamp from which the condition is valid. This parameter can be entered as a datetime Object, or a String. If unset this will default to the current date and time. Certain time accuracy can be omitted, if so the time will be set to 0's.
The parameter **valid_until** is a timestamp until which the condition is valid. This parameter can be entered as a datetime Object, or a String. If unset this will default to the maximum datetime (for infinite validity). Certain time accuracy can be omitted, if so the time will be set to 0's.

To add a condition:

```
conditionsDB.add_condition("detector3/subdetector1", "conditionName1", "SampleTag", value_array,
"testType", "2020-03-21 18:14", "2020-03-21 18:12", "2020-05-20")
```

To add a condition, leaving out type, collected_at, valid_since and valid_until:

```
conditionsDB.add_condition("detector3/subdetector1", "conditionsName1", "SampleTag3",
value_array)
```

This will set the condition to the values set, further Type will be set to None, collected_at and valid_since will be set to the current time and date and valid_until to the maximum time and date.

Get all conditions of a detector

To get all conditions of a detector from the Conditions Database, use the API function get_conditions(detector_id). This function returns a list with all conditions, represented as Python dictionaries, associated with the detector queried.

The parameter **detector_id** is a string identifying the detector for which the conditions must be retrieved.

To get all conditions of a main detector:

```
conditionsDB.get_conditions(“detectorX”)
```

To get the subdetector “detectorY” of “detectorX”

```
conditionsDB.get_conditions(“detectorX/detectorY”)
```

Get conditions by name

To get all conditions of a detector by their condition name, use the API function get_conditions_by_name(detector_id, name). This function returns a list with conditions, represented as Python dictionaries, associated with the conditions queried.

The parameter **detector_id** is a string identifying the detector for which the conditions must be retrieved.
The parameter **name** is a string identifying the condition name for which the conditions must be retrieved.

To get conditions by name:

```
conditionsDB.get_conditions_by_name(“detectorX”, “conditionName”)
```

Get conditions by tag

To get all conditions of a detector by their condition tag, use the API function get_conditions_by_tag(detector_id, tag). This function returns a list with conditions, represented as Python dictionaries, associated with the tag queried.

The parameter **detector_id** is a string identifying the detector for which the conditions must be retrieved.
The parameter **tag** is a string identifying the condition tag for which the conditions must be retrieved.

To get conditions by tag:

```
conditionsDB.get_conditions_by_tag(“detectorX”, “conditionTag”)
```

Get a condition by name and tag

To get a condition of a detector by its condition name and tag, use the API function get_conditions_by_name_and_tag(detector_id, name, tag). Since the name and tag combination is unique, this function returns a single condition, represented as a Python dictionary.

The parameter **detector_id** is a string identifying the detector for which the condition must be retrieved.
The parameter **name** is a string identifying the condition name for which the condition must be retrieved.
The parameter **tag** is a string identifying the condition tag for which the conditions must be retrieved.

To get a condition by name and tag:

```
conditionsDB.get_condition_by_name_and_tag("detectorX", "conditionName", "conditionTag")
```

Get conditions by validity

To get conditions of a detector by its validity time, use the API function get_conditions_by_name_and_validity(detector_id, name, start_date, end_date=None). Returns all the conditions, represented as dictionaries, with **name** as condition name and are valid from **start_date** up to and including **end_date.**

The parameter **detector_id** is a string identifying the detector for which the conditions must be retrieved.
The parameter **name** is a string identifying the conditions name for which the condition must be retrieved.
The parameter **start_date** is a timestamp from which the conditions queried should be valid. This parameter can be entered as a datetime, or a String.
The parameter **end_date** is a timestamp until which the conditions queried should be valid. This parameter can be entered as a datetime, or a String. This parameter is optional, if not entered the function will return everything that is valid on **start_date**.

To get conditions by validity interval:

```
conditionsDB.get_conditions_by_name_and_validity("detectorX", "detectorName", "2020-03-01",
"2020-05-01")
```

This will return all conditions of "detectorX" with name "detectorName" which were valid from 2020-03-01 00:00:00 until 2020-05-01 00:00:00

To get conditions by validity timestamp:

```
conditionsDB.get_conditions_by_name_and_validity("detectorX", "detectorName", "2020-03-01")
```

This will return all conditions of "detectorX" with name "detectorName" which were valid at 2020-03-01 00:00:00.

The dates in these functions can be cut off at any point, which means you can be as specific as you want. You can search for, for example, 2020-03 or 2020-05-06 12:00. For any of these dates, everything unspecified will get set to the first month/day at 00:00:00. This means the two examples will get set to 2020-03-01 00:00:00 and 2020-05-06 12:00:00 respectively.

This also means that ranges are calculated from the specific valued described above. An example, if we search for **valid_since**: "2020-01" **valid_until:** "2020-04-03 11:12". We will return all conditions which have a valid_since value larger or equal to 2020-01-01 00:00:00 and have a valid_until value less or equal than 2020-04-03 11:12:00

Get condition by collection date

To get a condition of a detector by its collection date, use the API function get_conditions_by_name_and_collection_date(detector_id, name, collected_at). Returns all the condition with **name** as condition name and are collected at **collected_at.** Since the name and collection date combination are unique it will always return one condition dictionary

The parameter **detector_id** is a string identifying the detector for which the condition must be retrieved.
The parameter **name** is a string identifying the condition name for which the condition must be retrieved.
The parameter **collected_at** is a timestamp at which the condition is collected. This parameter can be entered as a datetime, or a String

To get a condition using datetime for collected_at:

```
Import datetime

conditionsDB.get_conditions_by_name_and_collection_date(“detectorX”, “detectorName”,
datetime.datetime(2020-03-01))
```

To get a condition using string for collected_at:

```
conditionsDB.get_conditions_by_name_and_collection_date(“detectorX”, “detectorName”, “2020-03-
01”)
```

Update condition validity time and type

To update a conditions validity time and/or type, use the API function update_condition_by_name_and_tag(detector_id, name, tag, type=None, valid_since=None, valid_until=None). This function updates the type, valid_since, and valid_until of the condition specified.

The parameter **detector_id** is a string identifying the detector for which the condition must be retrieved.
The parameter **name** is a string identifying the condition name for which the condition must be updated.
The parameter **tag** is a string identifying the condition tag for which the conditions must be updated.
The parameter **type** is a string identifying the **new** type of the condition. This parameter is optional, if not filled in, the type will not be updated.
The parameter **valid_since** is a timestamp identifying the **new** starting date of the validity time of the condition. This parameter is optional, if not filled in, the starting date will not be updated.
The parameter **valid_until** is a timestamp identifying the **new** end date of the validity time of the condition. This parameter is optional, if not filled in, the end date will not be updated.

To update the only the type of a condition:

```
conditionsDB.update_condition_by_name_and_tag("detectorX", "conditionName", "conditionTag",
"testType2")
```

To update only the validity interval of a condition:

```
conditionsDB.update_condition_by_name_and_tag("detectorX", "conditionName", "conditionTag",
None, “2020-03-04”, “2020-05-04”)
```

Chapter 4: Using MongoDB with the Conditions Database

MongoDB

MongoDB is a document database with the scalability and flexibility that you want with high querying and indexing capabilities.

Docker

What is Docker?

“an open-source project that automates the deployment of software applications inside containers by providing an additional layer of abstraction and automation of OS-level virtualization on Linux.”

Basically, Docker is a tool that allows developers, sys-admins etc. to easily deploy their applications in a sandbox (called containers) to run on the host operating system i.e. Linux. The key benefit of Docker is that it allows users to package an application with all of its dependencies into a standardized unit for software development. Unlike virtual machines, containers do not have high overhead and hence enable more efficient usage of the underlying system and resources.

To Install Docker, you can refer to the detailed documentation for the OS you are using:

Linux, Windows, Mac

Note: Depending on how you've installed docker on your system, you might see a permission denied error after running the above command. If you're on a Mac, make sure the Docker engine is running. If you're on Linux, then prefix your docker commands with sudo. Alternatively, you can create a docker group to get rid of this issue.

Note: Docker works with Docker images, which are read-only templates with instructions for creating a Docker container. The current image uses CentOS as its base image, then it installs all the necessary dependencies for the proper usage of FairShip. These dependencies can be found in a different git repo, which is being maintained by CERN. Furthermore, the current image will automatically install mongoDB as well.

In order to build a new Docker image with the latests dependencies, you will need to

1. clone git repo.
2. Cd into the folder that contains Dockerfile.
3. docker image build.

Refer to the documentation here to execute docker commands.

This image can be run immediately by the following command

```
docker run <image_id>
```

A different process that can be followed would be to have an official docker image that will be maintained on docker hub. You could pull this image instead of creating your own version. To pull the docker image from docker hub, use the pull command:

```
docker pull <image_id>
```

The pull command fetches the fairship image from the Docker registry and saves it to our system. You can use the docker images command to see a list of all images on your system.

```
docker images
```

Once you are done installing Docker, test your Docker installation by running the following

```
docker run <image_id>
```

Now we are in the Docker environment. The last thing we have to do is initialize our Mongo Database and we are good to go. We do this by running the following two commands.

```
chmod +x <fairship root folder>/conditionsDatabase/start_mongodb_locally.sh
```

```
<fairship root folder>/conditionsDatabase/start_mongodb_locally.sh
```

MongoDB is now set up and good to go.

Installing MongoDB

Note: Skip this step if you are using the Docker image with MongoDB installed.
To install MongoDB on a non-debian based system follow the instructions here.
This tutorial installs MongoDB 4.2 Community Edition.

1. Import the public key used by the package management system.

From a terminal, issue the following command to import the MongoDB public GPG Key from https://www.mongodb.org/static/pgp/server-4.2.asc:

```
wget -qO - https://www.mongodb.org/static/pgp/server-4.2.asc | sudo apt-key add -
```

The operation should respond with an *OK*.

However, if you receive an error indicating that *gnupg is not installed*, you can:

Install gnupg and its required libraries using the following command:

```
sudo apt-get install gnupg
```

Once installed, retry importing the key:

```
wget -qO - https://www.mongodb.org/static/pgp/server-4.2.asc | sudo apt-key add -
```

2. Create a list file for MongoDB.

Create the list file */etc/apt/sources.list.d/mongodb-org-4.2.list* for your version of Ubuntu.

```
echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu bionic/mongodb-org/4.2
multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-4.2.list
```

3. Reload the local package database.

Issue the following command to reload the local package database:

```
sudo apt-get update
```

4. Install the MongoDB packages.

To install the latest stable version, issue the following

```
sudo apt-get install -y mongodb-org
```

Additional Information

ulimit Considerations

Most Unix-like operating systems limit the system resources that a session may use. These limits may negatively impact MongoDB operation. See UNIX ulimit Settings for more information.

Directories

If you installed via the package manager, the data directory /var/lib/mongodb and the log directory /var/log/mongodb are created during the installation.

By default, MongoDB runs using the mongodb user account. If you change the user that runs the MongoDB process, you must also modify the permission to the data and log directories to give this user access to these directories.

Configuration File

The official MongoDB package includes a configuration file (/etc/mongod.conf). These settings (such as the data directory and log directory specifications) take effect upon startup. That is, if you change the configuration file while the MongoDB instance is running, you must restart the instance for the changes to take effect.

Run MongoDB

Follow these steps to run MongoDB Community Edition on your system. These instructions assume that you are using the official mongodb-org package – not the unofficial mongodb package provided by Ubuntu – and are using the default settings.

To run and manage your mongod process, you will be using your operating system's built-in init system. Recent versions of Linux tend to use systemd (which uses the systemctl command), while older versions of Linux tend to use System V init (which uses the service command).
This manual is written assuming your system uses systemd init system.

If you are unsure which init system your platform uses, run the following command:

```
ps --no-headers -o comm 1
```

1. Start MongoDB

You can start the mongod process by issuing the following command:

```
sudo systemctl start mongod
```

If you receive an error similar to the following when starting mongod:

Failed to start mongod.service: Unit mongod.service not found.
Run the following command first:

```
sudo systemctl daemon-reload
```

Then run the start command above again.

2. Verify that MongoDB has started successfully.

```
sudo systemctl status mongod
```

You can optionally ensure that MongoDB will start following a system reboot by issuing the following command:

```
sudo systemctl enable mongod
```

3. Stop MongoDB.
As needed, you can stop the mongod process by issuing the following command:

```
sudo systemctl stop mongod
```

4. Restart MongoDB.
You can restart the mongod process by issuing the following command:

```
sudo systemctl restart mongod
```

5. Begin using MongoDB
Start a mongo shell on the same host machine as the mongod. You can run the mongo shell without any command-line options to connect to a mongod that is running on your localhost with default port 27017:

```
Mongo
```

Additionally, You can follow the state of the process for errors or important messages by watching the output in the /var/log/mongodb/mongod.log file.

Uninstall MongoDB Community Edition

To completely remove MongoDB from a system, you must remove the MongoDB applications themselves, the configuration files, and any directories containing data and logs. The following section guides you through the necessary steps.

WARNING

<u>This process will completely remove MongoDB, its configuration, and all databases. This process is not reversible, so ensure that all of your configuration and data is backed up before proceeding.</u>

1. Stop MongoDB.

Stop the mongod process by issuing the following command:

```
sudo service mongod stop
```

2. Remove Packages.

Remove any MongoDB packages that you had previously installed.

```
sudo apt-get purge mongodb-org*
```

3. Remove Data Directories.

Remove MongoDB databases and log files.

```
sudo rm -r /var/log/mongodb

sudo rm -r /var/lib/mongodb
```

Importing Data

You might want to import data into your database, this can be achieved using the mongoimport tool. The mongoimport tool is part of the MongoDB tools package. Consult the installation guide for your platform for instructions on how to install the tools package as part of your MongoDB installation. The below text explains how to import data into a specific database using a JSON formatted file.
From your system shell or command prompt, use the mongoimport tool to insert the documents into the collection in the conditionsDB database.

By default, mongoimport will import data into an instance of MongoDB on localhost, port 27017. To import data into a MongoDB instance running on a different host or port, specify the hostname or port by including the --host and --port options.

Use the --drop option to drop the collection if it already exists. This ensures that the collection will only contain the data you are importing.

```
mongoimport --db conditionsDB --collection detector_wrapper  \
      --authenticationDatabase admin --username <user> --password <password> \
      --drop --file ~/Downloads/inventory.crud.json
```

The above query can be simplified and used with minimum options as follows:

```
mongoimport --db conditionsDB --collection detector_wrapper  --file <path/fileName.json>
```

Additional information about mongoimport can be found here.

Exporting Data

mongoexport is a command-line tool that produces a JSON or CSV export of data stored in a MongoDB instance.

Run mongoexport from the system command line, not the mongo shell.

Syntax:

```
mongoexport --collection=<coll> [options]
```

Example:
To connect to a local MongoDB instance running on port 27017, you do not have to specify the host or port.
For example, to export the detector_wrapper collection to the specified output file from a local MongoDB instance running on port 27017:

```
mongoexport --collection=detector_wrapper  --db=conditionsDB  --out=detectors_exported_data.json
```

The detector_wrapper collection contains all detector and condition data. All data would exist in the detector_wrapper collection.

Additional information about mongoexport can be found here.

Creating Binary Exports

The mongodump tool is a utility for creating a binary export of the contents of a database. mongodump can export data from either mongod or mongos instances; i.e. can export data from standalone, replica set, and sharded cluster deployments.
The mongodump tool is part of the MongoDB tools package.
For standalone or a replica set, mongodump can be a part of a backup strategy with mongorestore for partial backups based on a query, syncing from production to staging or development environments, or changing the storage engine of a standalone.

The mongodump tool excludes the content of the local database in its output. The mongodump output only captures the documents in the database and does not include index data. mongorestore or mongod must then rebuild the indexes after restoring data.

To connect to a local MongoDB instance running on port 27017 and use the default settings to export the content, run mongodump without any command-line options:

```
mongodump
```

To specify a host and/or port of the MongoDB instance, you can use the following:
Specify the hostname and port in the --uri connection string:

```
mongodump --uri="mongodb://mongodb0.example.com:27017"  [additional options]
```

Specify the hostname and port in the --host:

```
mongodump --host="mongodb0.example.com:27017"   [additional options]
```

Specify the hostname and port in the --host and --port:

```
mongodump --host="mongodb0.example.com" --port=27017 [additional options]
```

For more information on the options available, click here.

Restore from a Binary Database Dump or Standard Input

The mongorestore program loads data from either a binary database dump created by mongodump or the standard input (starting in version 3.0.0) into a mongod or mongos instance.

The mongorestore tool is part of the MongoDB tools package.
Run mongorestore from the system command line, not the mongo shell.

```
mongorestore [options] [<directory>/<BSON file>]
```

For example, to restore from a dump directory to a local mongod instance running on port 27017:

```
mongorestore  dump/
```

As mongorestore restores from the dump/ directory, it creates the database and collections as needed and logs its progress:

For more examples, see Examples.
For more information on the options and arguments, see Options.