

Blacknest Data System (BDS)

BdsMetadata Program – 2.2.6 - 2021-05-04

1. Introduction

The BdsMetadata program provides a command line control client for the BDS system. It allows the database MetaData of the BDS system to be added to or changed.

The BdsMetadata program implements a simple script language that allows for relatively easy manipulation of the BDS MetaData.

2. Usage

The BdsMetadata program can be run only by admin users. It accepts the following command line options:

-help	Help on command line parameters
-host <hostname>	BDS Server host name
-user <user:password>	The BDS user id and password
-name <server name>	The BDS server name
-c <command>	Run the given script command
-f <command file>	Run the commands from the given script file
-d variable=value	Set the given variable to the value given
-maxChanges <n>	Maximum number of data changes per statement. (Can also be set using the \$maxChanges variable).
-verbose	Verbose messages

The BdsMetadata program will read the BDS_HOST environment variable at start-up. This variable, if set, defines the default BdsServer host name to contact. The default is “localhost” if this is not set. The program will carry out the operations as defined in the single “command” argument or from the command file if given.

The “-name <server name>” flag has to be given with the name of the BDS server being used. This adds an additional safety check.

3. Operation

The BdsMetadata program will run a sequence of statements written in its special purpose script language. The entire set of statements are run as a single BDS transaction and so if any statement fails, then none of the changes are made. Where possible, the line number of the statement which caused the error is given.

The maxChanges parameter defines the maximum number of database changes a single statement can perform. By default this is set to 1. This is to protect from selection patterns matching more items than expected.

The BdsMetadata program operates through the BDS AdminAccess API and so all of the user and other restrictions and protections of this interface are in operation. The program operates at the BDS API level and thus has all of the features of this API.

4. Metadata Scripting Language

The Metadata scripting is a simple language defined for the task. This section provides a brief description of this language.

- Comments can be included using a '#' or “//” as the start of a line. Also 'C' style “/* */” comments are supported.
- Statements consist of a command followed by arguments to the command and terminated with the ';' character.
- Statements can be on one or multiple lines. There can also be multiple statements per line.
- String variables are supported using the '\$' character in front of the variables name.
- Sets of parameters are supported as a CSV list or name/value list within curly brackets.

4.1. Values

Values can be:

- Simple identifiers consisting of alpha-numeric characters starting with [A-Za-z].
- Strings consisting of characters within double quotes.
- Variables like \$startTime.

4.2. Parameter Lists

Parameter lists can be in CSV format or in name/value format. The name/value format is as follows:

```
{station=SA,type=array}
```

Normally this is used to set an objects members, but it can also be used for pattern matching. The field names are those as listed in the BDS API for the object in question. Unnamed members are set to default values or left unchanged

The CSV list is simply a comma separated list:

```
{,,SA,array}
```

Parameters are set based on the order within the object. If there is no value given, the members value is set to a default value or left unchanged.

4.3. Pattern Match Lists

Pattern matching using the standard BDS pattern matching scheme using a Selection object. The pattern can be defined using a CSV list or a name/value list. The parameters are as follows:

```
{startTime,endTime,network,station,channel,source}
```

Each item can be a simple string or a regular expression such as: “EKB.*”

When patterns are used for selection Digitisers or Sensors, the program will search for the channels that match the pattern and then the Digitisers and Sensors within the selection time.

4.4. Variables and Default values

The language supports string variables using the “\$” character. variables can be set using the “set” command and as the results of some commands. There is also a default set of parameters defined. When an object is added then the system will use the values of variables of the same member name as the defaults for parameters values not given.

By default the following variables are defined. These can be cleared using the “clear” command:

<i>Name</i>	<i>Value</i>
startTime	0001-01-01T00:00:00
endTime	9999-01-01T00:00:00
network	BN
source	Main
dataType	seismic
datum	WGS84
maxChanges	1

Thus by default when a Channel is created it will have the startTime and endTime as set above unless overridden.

All current variable settings can be listed with the “print;” command.

4.5. Commands

The following commands are currently available:

clear;

Clears all variables.

set \$<variable> = <value>;

Sets the named variable to the given value;

print \$<variable>;

Prints the value of the given variable;

print;

Prints all of the variables;

backup \$<variable>;

Backup the current Database and BdsServer configuration into the current BDS data store within the Backup directory. Each entry has a reference based on the time time backup was made to the nearest second. The backup reference is returned in the variable given as an argument.

BEAM

restore "<value>;"

Restores the database to the one saved with the given reference.

info <Type>;

List the members of the given type.

list <Type> {<pattern>;}

List all items of the given type that match the selection pattern.

find <Type> \$<variable> {<pattern>;}

Searches for a selection match and returns the id of the selected item. Returns an error if no matches or more than one match.

delete <Type> {<pattern>;}

Deletes the objects of the given type that match the pattern given. If a Channel is deleted then so will all of its Calibrations, Responses and Instruments.

add <Type> \$<variable> {parameters};

Add an object of the given type setting its members as given in the parameter list. The objects id is returned in the variable. The objects members are set to the values in the parameter list or the value of variables that have the same name as the members if not set.

Adding a Channel will also add the needed Station if it does not exist.

When adding an "array" the following syntax should be used:

add Station \$<variable> {parameters} {station/channel pair list};

This special syntax is used to add an array. The Station's "type" field should be set to "array". The station/channel pair list should be a simple comma separated list such as: {S1,C1_A1,S2,C1_A1}.

add Response \$<variable> {parameters};

This adds a single response as the "Overall" stage 0 response.

add Responses \$<variable> {parameters};

This special syntax creates a set of responses based on the multiple response variable given. The responses will share the same time range etc, but will have different stage,name and type as defined in the responses stored in the given variable. It will also add the stage 0,Sensor response based on the 1,instrument response.

read <PoleZero,FAP> \$<variable> { "<filename>"};

Read a response from the given file. The response is stored in the variable given and this can be used on a subsequent "add Response ..." command. The read command supports SAC and IDC file formats. Response types supported include PoleZero,FAP and FIR. From these the first response is only imported and used as the stage 0,Sensor response. The command also supports the type "Responses". When this is use a complete set of multi-stage responses are read. The one that is for stage 1,instrument is duplicated as the stage 0,Sensor response. The multiple response read cad be stored using the "add Responses" command.

change <Type> \$<variable> {pattern} {parameters};

Changes all the objects of the given type that match the given pattern using the parameters given. Returns the id of the last object affected.

split <Type> \$<variable> {pattern} <time> {parameters};

Splits the object of the type given that matches the pattern into two separate ones split at the time given. The second object has its members changed to those given in the parameters list or as given by default variables.

clone <Type> \$<variable> {pattern} {parameters} [{arguments}];

Clones the object of the type given that matches the pattern to create a copy which has its members changed to those given in the parameters list or as given by default variables.

In the case of a Channel type, the optional arguments list is used. Normally cloning a Channel will just clone the Channel. The arguments field is a comma separated list of names that allows the command to also clone the calibrations, responses and instruments. The argument fields supported are:

- calibration Clone the calibrations
- response Clone the Responses
- instrument Clone the instruments
- digitiser Normally cloning instruments will share the Digitiser of the cloned channel. Setting this parameter will create a clone of the Digitiser for this new Channel.
- sensor Normally cloning instruments will share the Sensor of the cloned channel. Setting this parameter will create a clone of the Sensor for this new Channel.

When cloning a Channel with its calibrations, responses, instruments etc the program will perform the following:

- Will select the calibrations, responses, instruments based on the pattern. This may select more than one.
- If one item is selected, then its start/end times will be set to that of the new channel as defined in the parameters.
- If more than one item is selected (multiple Calibrations etc) then all of these that are within the time range of the “new” Channel are cloned keeping the intermediate times unchanged. If the items startTime is before than the new Channels start time it will be changed to match the new Channels startTime. Similarly the items endTime will be set to the new Channels endTime if it is after the Channels endTime.

4.6. Example Script

```
#####  
#      BDS MetaData command line manipulation script  
#      Adding channels  
#####  
#
```

BEAM

```
# Admin, backs up database
backup $backup;

# Sets default variables
set $startTime = 0001-01-01;
set $endTime   = 9999-01-01;
set $network    = BN;
set $station    = "";
set $channel    = "";
set $source     = Main;
set $dataType   = seismic;

# Info on a Type
info Calibration;

# Delete all existing channels and stations that start with "H"
delete Station {station="H.*"};
delete Channel {station="H.*"};

# Add some Pole/Zero responses
add PoleZero $pz0 {0,1,2,3} {10,11,12,13};
add PoleZero $pz1 {0,1,2,4} {10,11,12,13};

# Add a channel with calibration, responses and instrument
set $station    = "H08C1";
set $channel    = "LEA_US";
add Channel $c {};
add Calibration $cal {,,,,,Cal1,100,1,1,m,0,0,0};
add Response $r {,,,,,0,"Overall","PoleZero",,,,,,$pz0};
split Response $r {station=H08C1,channel=LEA_US} 2000-00-00
{,,,,,0,"Overall","PoleZero",,,,,,$pz1};
add Digitiser $d {,,,"20 DMOD","Type0",1,0,40,0,0};
# add Sensor $s {,,,"EKA DSP Willmore","Mk11",,,,,};
clone Sensor $s {startTime=2008-02-02T00:00:00, endTime=2008-02-04T00:00:00,
station="TSB01", channel="BHZ_01", source="Main"}{serialNumber=112291};
add ChannelInstrument $i {,,,$c,Main,$d,$s};

# Clone this channel to a set of channels
clone Channel $c {station=H08C1,channel=LEA_US} {station=H08C1,channel=LEV_US}
{calibration,response,instrument};
clone Channel $c {station=H08C1,channel=LEA_US} {station=H08C2,channel=LEA_US}
{calibration,response,instrument,digitiser};
clone Channel $c {station=H08C1,channel=LEA_US} {station=H08C2,channel=LEV_US}
{calibration,response,instrument,digitiser,sensor};

# Find the ID of a Sensor
find Sensor $s {,,,"EKB1","SHZ",EkaDig2};
print $s;

#Add a set of stage responses
read Responses $r0 { "idc_all.response" };

delete Response {,,TT,TSB09,BHZ_01,Main};
add Responses $r {,,TT,TSB09,BHZ_01,Main,0,"","",,,,,,$r0};
```

```
list Response {,,TT,TSB09,BHZ_01,Main};
```

5. Types

The bdsMetadata program operates through the BDS API on the BDS API data types. The fields of these are documented in the bdsApi manual at:

<https://portal.beam.ltd.uk/support/blacknest/files/bds/doc/bdsApi/html/index.html>

There are some special field names that are used with some data types:

Response: The field name “**response**” is used to set the actual PoleZero, FAP or FIR table values. A BdsMetadata variable that has been set to one of these “PoleZero,FAP,FIR” data types is used to set this fields value.

6. Return Value

The program will return a status value of 0 if all was Ok. It will return a non zero value, the BDS error number, on error together with a message output on stderr.

7. Further Information

For further information please look at the BDS system documentation at:

<https://portal.beam.ltd.uk/support/blacknest>.