gds documentation

Release 1.6.0

Software and Controls Group

Sep 12, 2019

Contents

1 Introduction **1**

2 Local and global environment **2**

3 Working with bundles and modules **3**

4 The gds command line tool **4**

4.1 gds init . . . 5
4.2 gds new [options] <module> . . . 5
4.3 gds env . . . 6
4.4 gds gen [options] [element] . . . 6
4.5 gds info [options] . . . 9
4.6 gds build [options] [module] . . . 9
4.7 gds install [options] [module] . . . 10
4.8 gds clone [options] [module] . . . 10
4.9 gds test [options] [module] . . . 11
4.10 gds validate [options] [module] . . . 12
4.11 gds options . . . 12

5 Starting to use gds **14**

CHAPTER

ONE

INTRODUCTION

This document describes the GMT OCS development environment and the use of the *gds* tool.

CHAPTER

TWO

LOCAL AND GLOBAL ENVIRONMENT

In order to support the deployment of completed modules and the development of new modules or new versions of existing ones, the GMT SDK defines two different scopes in the file system:

1. The global scope includes modules that are available for all the users in an SDK installation. The default location is `/opt/gmt` although it can be configured to a different location setting the shell environment variable `GMT_GLOBAL`. The global scope directories and files are usually owned by an administration and deployment account and not by an individual user account. The default account is root.
2. The local scope includes modules that are being actively developed or different versions of existing ones for testing purposes. The location of the local scope in the file system is defined by the `GMT_LOCAL` environment variable. The variable is set individually for every developer.

CHAPTER

THREE

WORKING WITH BUNDLES AND MODULES

A Module represents the basic unit of development, integration and test in the GMT SDK. Modules are grouped in Bundles.

By default the SDK defines two types of bundles, *global* and *local*, for each development scope. The global bundle include modules procured by GMT as part of the SDK. These modules cannot be altered, but must be available in order to build and run the system. The local bundle(s) define the modules in which developers will be working.

Subsystems defined in the GMT model are mapped into Modules for the purpose of development, testing and integration. As explained in the Software and Controls Standards, Subsystems are modeling entities and are decomposed in Packages and Components. The Module file system structure is based in the Subsystem file system structure too.

An example of a local bundle is included in the following code segment:

```
module.exports =
   name:       "local"
   desc:       "List of local development modules"
   elements:
      isample_dcs: { active: true, test: false, developer: 'gmto', domain:
↪'idcs' }
      hdk_dcs:     { active: true, test: false, developer: 'gmto', domain:
↪'idcs' }
```

Bundle definitions are saved in `$GMT_LOCAL/etc/`. The file with this definition is saved as `local_bundle.coffee`. This step has to be performed only when a new module is created.

The bundle includes the definition of a module (isample_dcs) that is part of the idcs (Instrument Device Control System) domain.

The gds utility allows managing the life-cycle of each module as a unit. If we want to create a new module we will execute:

```
> gds new isample_dcs
```

CHAPTER

FOUR

THE GDS COMMAND LINE TOOL

The GMT SDK includes the gds tool to assist in the development of GMT software.

Tip: In order to see the commands available for the gds tool write in `gds --help` in the shell command line.

```
> gds --help


Usage: gds [options] [command]

Options:

  -V, --version                output the version number
  -l, --logging <level>        Activates <level> logging
  -i, --interactive            Activates interactive mode
  -f, --file_log <filename>    Streams the log output to filename
  -s, --sim_mode               Executes commads in simulation mode
  -h, --help                   output usage information

Commands:

  init                         Initializes development environment
  new [options] <module>       Creates a new module
  env                          Prints the GMT environment variables
  gen [options] [element]      Generates code for the local bundle or the␣
↪optional element
  info [options] [module]      List information of the local bundles or the␣
↪optional module
  build [options] [module]     Builds the local bundles or the optional␣
↪module
  install [options] [module]   Installs the local bundle modules or the␣
↪optional module
  clone [options] [module]     Clones the local bundle modules or the␣
↪optional module
  test [options] [module]      Run the test of the local bundle modules or␣
↪the optional module
  validate [element]           Validates the model element
```

The gds tool implements a set of commands that allow to operate with bundles, modules and module elements as a whole. The following sections describe each of the commands. Each command has its own help that can be displayed by executing:

```
> gds <command> --help
```

4.1 gds init

Description

Initializes the file system structure of the local development environment (defined by the shell variable `GMT_LOCAL`). This operation is usually only needed when a developer uses the GMT SDK for the first time.

```
> gds init
```

The local file structure created contains the following directories:

```
$GMT_LOCAL/
    |-- bin/
    |-- db/
    |-- doc/
    |-- etc/
    |-- examples/
    |-- include/
    |-- lib/
    |-- test/
    |-- var/
```

This structure is similar to the one in `GMT_GLOBAL`. The deployable files of the local modules will be installed in these directories and will have preference over the ones installed in `GMT_LOCAL`. This is specially useful for developers working on new versions of modules that are part of the SDK distribution.

4.2 gds new [options] <module>

Description

The `gds new` command creates a default file structure for a new module. Each module is mapped into the file system using the following structure:

```
<module>/
   |-- model
   |     |-- <module>.coffee
   |     |-- <module>_def.coffee
   |     |-- <module>.rst
   |     |-- <module>_ld.coffee
   |     |-- <module>_types.coffee
   |     |-- <pkg_1_pkg>/
   |     |         |-- <component_1>
   |     |         |-- ...
   |     |         |-- <component_n>
   |     |-- ...
   |     |-- <pkg_n_pkg>
   |     |-- webpack.config.coffee
   |-- src
   |     |-- coffee/
   |     |-- py/
   |     |-- cpp/
```

```
|    |-- etc/
|
|-- docs/
```

See the Model Specification Guide Document for a description of the `model` file structure. The `src` file structure depends on the target programming language. The details of each language mapping are described in the corresponding document.

```
> gds new --help

Usage: new [options] <module>

Creates a new module

Options:

-o, --override  override existing files
-p, --preserve  preserve existing files
-h, --help      output usage information
```

Options

`-o, --override` If the command is run more than once, this flag will write over the existing files

`-p, --preserve` If the command is run more than once, this flag will preserve a copy of the existing files with the extension *.as_is*

`-h, --help` Display the help message

4.3 gds env

Description

The `gds env` command displays the values of the environment variables used by gds.

4.4 gds gen [options] [element]

Description

The `gds gen` command implements a set of generators that use as an input the model specifications and applies to them different transformations. The output of this transformations produces one or several files that are stored in the module file system.

```
> gds gen --help

Usage: gen [options] [element]

Generates code for the local bundle or the optional element

Options:

   -m, --model  [name]  model containing element (default: model)
   -t, --target [name]  code|scaffold|doc|test (default: code)
   -h, --help           output usage information
```

Options

`element`

Specifies the model element used as source for the transformations. The model element could be an aggregate or simple. If the model element is an aggregate, e.g. SubSystem or Package, gds would apply the transformations to all the elements contained in the aggregate.

```
> gds gen isample_dcs
```

In this example gds will generate artifacts for all the packages part of the DCS isample_dcs. As Packages are also aggregates it would generate artifacts for all the Components in each Package.

```
> gds gen isample_ctrl_pkg
```

In this example gds will generate artifacts for all the Components part of the isample_ctrl_pkg.

```
> gds gen isample_filter_wheel_ctrl
```

In this example gds will generate artifacts for the Component isample_filter_wheel_ctrl.

The behavior of the generator can be configured per component basis in the module definition file. The following attributes are available per Component:

`language` List of language transformations to be used by the generators. The values of language depend on which target is being used. See description of the option '–target'

`build` Specifies how the generated code should be build. Possible values are:

`obj` When the language is compiled the generator will produce Makefile targets that build an object library with the Component code. The library file will be installed in $GMT_LOCAL/<deployment_destination>/lib/<platform>. Where platform is 'so' or 'js'. The value of <deployment_destination> depends on the attribute 'deploy'.

`app` When the language is compiled the generator will produce Makefile targets that build an executable. The executable file will be installed in $GMT_LOCAL/<deployment_destination>/bin. The value of <deployment_destination> depends on the attribute 'deploy'.

`deploy` Specifies where the artifacts resulting from building the generated code will be installed. The possible values are:

`dist` The build artifacts will be installed in the base module path: $GMT_LOCAL/

`test` The build artifacts will be installed in $GMT_LOCAL/test

`example` The build artifacts will be installed in $GMT_LOCAL/example

`codegen` If **true** the codegen transformations will be applied to the element, otherwise will be ignored.

`active` If **false** the element will be ignored by any gds command.

The following paragraph includes an excerpt from the isample_dcs_def.coffee module definition file.

```
module.exports =
   elements:
      isample_ctrl_pkg:
            elements:

               isample_ctrl_super:
                  language:    ['cpp','coffee']
```

```
            build:       'obj'
            deploy:      'dist'
            codegen:     true
            active:      true

         isample_filter_wheel_ctrl:
            language:    ['cpp']
            build:       'obj'
            deploy:      'dist'
            codegen:     true
            active:      true

         isample_hw_adapter:
            language:    ['cpp']
            build:       'obj'
            deploy:      'dist'
            codegen:     true
            active:      true

         isample_ctrl_app:
            language:    ['cpp']
            build:       'obj'
            deploy:      'dist'
            codegen:     true
            active:      true

         isample_main_app:
            language:    ['coffee']
            build:       'app'
            deploy:      'dist'
            codegen:     true
            active:      true

   isample_example_pkg:
         elements:

         isample_ex_app:
            language:    ['coffee']
            build:       'app'
            deploy:      'example'
            codegen:     true
            active:      false

   isample_doc_pkg:
         elements:

         isample_manual_doc:
            language: ['rst']
            build: 'pdf'
            deploy: 'doc'
            codegen: true
            active: false

         isample_notes_rpt:
            language: ['rst']
            build: 'pdf'
            deploy: 'doc'
            codegen: true
```

```
                active: false
```

`--model`

Specifies the model layer to use as source for the generator, The possible values are model, metamodel or metametamodel.

Each model layer has a different set of generators. For code generation the source model layer is 'model'. The different transformations for code generation can be consulted in the corresponding mapping document:

- Model to c++ Mapping Specification Document
- Model to nodejs Mapping Specification Document. ref: TBD
- Model to python Mapping Specification Document. ref: TBD

`--target`

Specifies the family of generators to be use. The possible values are:

code The transformations from model to code will be applied to the specified model element. The language(s) for each component are defined in the module definition file (see above example).

scaffold This target value takes as an input the module definition file and applies metamodel transformations to the model elements defined in the module definition file. The output of the transformations is saved in the file system and includes skeleton files for the specification of the module model elements. The types of each model element is inferred from the termination of the names on the model elements. The appendix Class n-grams in the Glossary defines the predefined termination and their corresponding mapping.

doc This target value generates documentation taking as an input the specification of the module elements. Documents have to be specified in the module definition file (see example above). The language transformation output values can be specified on a per document basis.

test This target value generates conformance tests skeletons taking as an input the specification of the module elements. The test skeletons are generated in <module>/test directory.

4.5 gds info [options]

Description

The `gds info` command display information of the modules that the gds tool is processing.

```
> gds info --help

Usage: info [options]

List information of the active modules

Options:

  -h, --help            output usage information
```

4.6 gds build [options] [module]

Description

The `gds build` command builds the specified module. If no argument is passed it will build the modules of the local bundle.

```
> gds build --help

Usage: build [options] [module]

Builds the local bundles or the optional module

Options:

  -b, --bundle <name>  uses the bundle <name> instead of the local bundles␣
↪(default: )
  -h, --help           output usage information
```

Options

`--bundle <name>` The command will build the modules contained in the specified bundle.

4.7 gds install [options] [module]

Description

The `gds install` command installs the deployable artifacts of the specified module. If no argument is passed it will install the artifacts of the modules in the local bundle.

```
> gds install --help

Usage: install [options] [module]

Installs the local bundle modules or the optional module

Options:

  -b, --bundle <name>  uses the bundle <name> instead of the local bundles␣
↪(default: )
  -h, --help           output usage information
```

Options

`--bundle <name>` The command will install the modules contained in the specified bundle.

> **Warning:** In the latest release the use of the `install` command is not recommended as the installation will be performed as part of the invocation of the make command.

4.8 gds clone [options] [module]

Description

The `gds clone` command will clone the Github repositories corresponding to the specified module. The option `--developer` is used to determine which Github account is to be used as upstream repository. If no argument is passed all the modules of the local bundle will be cloned.

```
> gds clone --help

Usage: clone [options] [module]

Clones the local bundle modules or the optional module

Options:

  -b, --bundle    <name>  uses the bundle <name> instead of the local␣
↪bundles (default: )
  -d, --developer <name>  overrides the developer defined in the bundle file␣
↪(default: )
  -h, --help              output usage information
```

Options

bundle The command will clone the modules contained in the specified bundle.

developer The name of the Github repository account.

4.9 gds test [options] [module]

Description

The gds test allows the automated execution of module tests. Module tests are created in the corresponding module test directory. The name of the files containing the tests must conform to the syntax: <element>_<test_number>_<test_type>_test.coffee

```
> gds test --help

Usage: test [element]

Run the tests for thee model element

Options:

   --scope <scope name>    scope for running the test (default: local)
   -n, --number <number>   number of the test (default: 01)
   --type <test_type_name> runs only the tests of the specified type␣
↪(interface|functional|performance|all) (default: all)
   --recursive             runs the test recursively if the element is an␣
↪aggregate
   -h, --help              output usage information
```

The following code shows an example of executing a functional test.

```
> gds test isample_dcs -n 01 --type functional

# -> will run the test $GMT_GLOBAL/test/isample_dcs/isample_dcs_01_
↪functional_test.coffee
```

Options

type The command will execute the test of the specified type

number Only the test matching the number will be executed.

scope The command will search for tests in the specified scope. The default value is *global*

suite Only the test matching the suite identifier will be executed. This allow to organize tests in different categories that can be invoked incrementally (e.g. integration or performance tests after unit tests).

recursive The command will search for tests in the specified scope. The default value is *global*

4.10 gds validate [options] [module]

Description

The `gds validate` performs a series of validations on model specifications.

```
> gds validate --help

Usage: validate [element]

Validates the model element

Options:

   -h, --help              output usage information
```

4.11 gds options

The following options can be used in combination with the previous commands

-V, --version The `gds` command will print the version number

-l, --logging <level> Activates <level> logging. The following table describes the different values available for the `logging` option

<level>	Abbr	Description
fatal	FTL	fatal - errors which make the application unusable
error	ERR	error - errors that preclude to achive an specific request
warning	WRN	warning - problems that may caused that the result achieved may not be the expected
info	INF	info - information about the general execution of the application
debug	DBG	debug - information to provide an understanding of the internal of the application
trace	TRC	trace - information that may server to identify a potential problem
metric	MET	metric - information to record performance metrics relative to the execution of the application

-i, --interactive Activates interactive mode. In this mode, the gds tool enters in a interactive session in which the model can be navigated and plugin methods can be invoked.

> **Warning:** In the current release, after invoking `gds -i` entering the `return` key is needed in order for the gds interactive prompt to appear.

-f, --file_log <filename> Streams the log output produced during the execution of a gds command to <*filename*>

-s, --sim_mode Executes commands in simulation mode. In this mode, commands are executed but changes to the file system are prevented

> **Tip:** Use this option in combination with `--logging trace` for troubleshooting.

-h, --help output usage information

CHAPTER

FIVE

STARTING TO USE GDS

1. Confirm with your system administrator that the GMT SDK has been installed in your machine. The default installation location is /opt/gmt
2. Add the GMT environment variables to your .profile

```
export GMT_LOCAL=<your local development directory>
export GMT_GLOBAL=<the global installation of the OCS SDK>
source $GMT_GLOBAL/bin/gmt_env.sh
```

3. Close and open a new terminal session so the new environment becomes available
4. Execute the following commands to verify that the GMT environment variables are correctly set

```
> gds --help
> gds env       # displays GMT environment variables
```

5. Change to your local GMT development directory:

```
> cd $GMT_LOCAL
```

6. If it's the first time using the GMT SDK, initialize your file system structure

```
> gds init
```

   gds init will initialize the file system structure of GMT_LOCAL. It will create the $GMT_LOCAL file structure, where the deployable files of your local development are saved.

   From this moment the gds commands can be invoked to perform the required development tasks.