The Implementation

Sven van der Meer

Table of Contents

for SKB-Framework v0.0.1, November 16, 2018

1. skb-framework

The `skb-framework` is the main entry point to the framework. It realizes two things in one. First, it is an application itself. Second, it is a start script that can be used by other applications to start the framework.

At the start, it checks the setting `__FW_LOADER_FLAVOR` (line 1 in the source block below). If this variable is set, then another application wants to start the framework. Otherwise, the `skb-framework` is the application. In the later case, the script sets required variables for the loader (lines 3-5 below):

- `__FW_LOADER_FLAVOR` - the flavor of the application, here `SF`
- `__FW_LOADER_SCRIPTNAME` - the name of the script (application)
- `__FW_LOADER_APPNAME` - the application name

The next step is to find the framework installation. The script tries the variable `SF_HOME` first, `readlink` first (lines 8-13), if that fails `dirname` (lines 15-19). If all attempts fail, the script terminates with an error (lines 20-24). Otherwise it set `FW_HOME`.

```
if [[ -z ${__FW_LOADER_FLAVOR:-} ]]; then
    ## we should load the framework itself, so SF
    export __FW_LOADER_FLAVOR="SF"
    export __FW_LOADER_SCRIPTNAME="$0"
    export __FW_LOADER_APPNAME="SKB Framework"

    ## try readline to find where we are
    if [[ -z ${SF_HOME:-} ]]; then
        SF_HOME=$(readlink -f $0)
        SF_HOME=${SF_HOME%/*}
        SF_HOME=${SF_HOME%/*}
        export SF_HOME
    fi
    ## try dirname to find where we are
    if [[ -z ${SF_HOME:-} ]]; then
        SF_HOME=$(dirname $0)
        SF_HOME=$(cd $SF_HOME/..; pwd)
        export SF_HOME
    fi
    if [[ -z ${SF_HOME:-} ]]; then
        printf "  unable to set home \$SF_HOME (tried environment, readlink, and
dirname \$0)\n"
        printf "  please set SF_HOME\n\n"
        exit 10
    fi
    export FW_HOME=$SF_HOME
```

If `skb-framework` is used by another application to start the framework, the script only tries to find the framework installation. The mechanism here is the same as explained above: try `FW_HOME` first, then `readlink`, then `dirname`. IF all fails, exit with an error.

```
else
    ## try readline to find where we are
    if [[ -z ${FW_HOME:-} ]]; then
        FW_HOME=$(readlink -f $0)
        FW_HOME=${FW_HOME%/*}
        FW_HOME=${FW_HOME%/*}
        export FW_HOME
    fi
    ## try dirname to find where we are
    if [[ -z ${FW_HOME:-} ]]; then
        FW_HOME=$(dirname $0)
        FW_HOME=$(cd $FW_HOME/..; pwd)
        export FW_HOME
    fi
    if [[ -z ${FW_HOME:-} ]]; then
        printf "  unable to set framework home \$FW_HOME (tried environment, readlink,
and dirname \$0)\n"
        printf "  please set FW_HOME\n\n"
        exit 10
    fi
```

One the framework installation has been found, the script tests if the loader exists.

```
if [[ ! -x $FW_HOME/bin/loader/loader.sh ]]; then
    printf "  did find/set \$FW_HOME, but did not find loader\n"
    printf "  tried $FW_HOME/bin/loader/loader.sh\n\n"
    exit 11
fi
```

When all conditions are satisfied, the script executes the loader handing over all arguments unprocessed.

```
$FW_HOME/bin/loader/loader.sh $*
exit $?
```

2. The Loader

The loader is the script `$FW_HOME/bin/loader/loader.sh`. It is responsible for all initial configuration, loading elements, testing settings and dependencies, processing command line arguments (options), and execute task, scenarios, or the shell. The load process has multiple steps, starting with some initial settings and finished with a cleanup.

The initial settings are shown in the source block below. Line 1 restricts *bash*, providing a safer execution environment. Line 2 allows for extended globbing (finding files recursively with wildecards such as `*/.adoc`). Line 3 takes the current time. This information is later used to calculate how long the load process did take. Line 4 removes an environment setting that prints rather annoying messages when running Java.

```
set -o errexit -o pipefail -o noclobber -o nounset
shopt -s globstar
_ts=$(date +%s.%N)
unset JAVA_TOOL_OPTIONS
```

2.1. Dependencies

The first real action in the load process is the test for core dependencies. The source block below shows how the loader tests for:

- *BASH* version 4 - needed to use associative arrays (or maps)
- GNU *Getopt* - extensively used in the loader and tasks to parse command lines
- *bc* - a calculator used for calculating execution time of tasks and scenarios
- *mktemp* - used to create names for temporary directories, required to store runtime configuration information

```
if [[ "${BASH_VERSION:0:1}" -lt 4 ]]; then
    printf " ==> no bash version >4, required for associative arrays\n\n"
    exit 12
fi
! getopt --test > /dev/null
if [[ ${PIPESTATUS[0]} -ne 4 ]]; then
    printf " ==> getopt failed, require for command line parsing\n\n"
    exit 13
fi
if [[ ! $(command -v bc) ]]; then
    printf " ==> did not find bc, require for calculations\n\n"
    exit 14
fi
if [[ ! $(command -v mktemp) ]]; then
    printf " ==> did not find mktemp, require to create temporary files and
directories\n\n"
    exit 15
fi
```

2.2. Core and default Settings

Once all dependencies are satisfied, the loader realizes core settings:

- If not set, then set `$FW_HOME` (lines 1-4)
- Source the loaders declaration files (line 6). This will create the main configuration map called `CONFIG_MAP` along with the map `CONFIG_SRC`.
- Set core variables in the configuration map (lines 7-25):
  - *FW_HOME* - the home directory of the framework
  - *RUNNING_IN* - set to loader, this will later be changed the `shell` or `task` by the shell and tasks
  - *SYSTEM* - to know in which system the framework is running
  - *CONFIG_FILE* - the SKB configuration file
  - *STRICT* - set strict mode to `off`, at least initially
  - *APP_MODE* - set the default application mode to `use`
  - *PRINT_MODE* - set the default print mode to `ansi` (for ANSI formatted text with colors and effects)
  - Levels - set the levels for loader, shell, and tasks initially to `error`
  - Quiet - set quiet mode for loader, shell, and tasks to `off`, i.e. they are *not* quiet
  - *SCENARIO_PATH* - create an empty path for scenarios
  - *SHELL_SNP* - activate shell prompt

```
if [[ -z ${FW_HOME:-} ]]; then
    FW_HOME=$(dirname $0)
    FW_HOME=$(cd $FW_HOME/../.. && pwd)
fi

source $FW_HOME/bin/loader/declare/_include
CONFIG_MAP["FW_HOME"]=$FW_HOME                      # home of the framework
export FW_HOME
CONFIG_MAP["RUNNING_IN"]="loader"                   # we are in the loader,
shell/tasks will change this to "shell" or "task"
CONFIG_MAP["SYSTEM"]=$(uname -s | cut -c1-6)        # set system, e.g. for Cygwin path
conversions
CONFIG_MAP["CONFIG_FILE"]="$HOME/.skb"              # config file, in user's home
directory
CONFIG_MAP["STRICT"]=off                            # not strict, yet (change with
--strict)
CONFIG_MAP["APP_MODE"]=use                          # default application mode is use,
change with --all-mode, --build-mode, --dev-mode
CONFIG_MAP["PRINT_MODE"]=ansi                       # default print mode is ansi,
change with --print-mode

CONFIG_MAP["LOADER_LEVEL"]="error"                  # output level for loader, change
with --loader-level, set to "debug" for early code debugging
CONFIG_MAP["SHELL_LEVEL"]="error"                   # output level for shell, change
with --shell-level
CONFIG_MAP["TASK_LEVEL"]="error"                    # output level for tasks, change
with --task-level

CONFIG_MAP["LOADER_QUIET"]="off"                    # message level for loader, change
with --lq
CONFIG_MAP["SHELL_QUIET"]="off"                     # message level for shell, change
with --sq
CONFIG_MAP["TASK_QUIET"]="off"                      # message level for tasks, change
with --tq

CONFIG_MAP["SCENARIO_PATH"]=""                      # empty scenario path, set from
ENV or file (parameter)
CONFIG_MAP["SHELL_SNP"]="off"                       # shell shows prompt, change with
--snp
```

2.3. Core Includes

Next, some core files from the framework's API are loaded. To makes things simple, the provided `include files are used to load all API functions. Error and warning counters are reset, i.e. set to _0`. Finally, the loaders own function for parsing the command line is loaded (line 6). This only loads the function, it does not actually parse the command line.

```
source $FW_HOME/bin/api/_include
ConsoleResetErrors
ConsoleResetWarnings

source $FW_HOME/bin/api/describe/_include
source $FW_HOME/bin/loader/init/parse-cli.sh
```

2.4. Application Flavor and Name

The next step is to set the flavor and application name/script. The flavor is any prefix used by the application to identify parameters. It must be provided by the application (which starts the loader) as the setting `__FW_LOADER_FLAVOR`. Once flavor and application settings are realized, the application map and version are loaded (lines 44-51).

```
if [[ -z ${__FW_LOADER_FLAVOR:-} ]]; then
    ConsoleFatal " ->" "interal error: no flavor set"
    printf "\n"
    exit 16
else
    CONFIG_MAP["FLAVOR"]=$__FW_LOADER_FLAVOR
    CONFIG_SRC["FLAVOR"]="E"
    if [[ -z ${CONFIG_MAP["FLAVOR"]} ]]; then
        ## did not find FLAVOR
        ConsoleFatal " ->" "internal error: did not find setting for flavor"
        printf "\n"
        exit 16
    fi

    FLAVOR_HOME="${CONFIG_MAP["FLAVOR"]}_HOME"
    CONFIG_MAP["APP_HOME"]=${!FLAVOR_HOME:-}
    CONFIG_SRC["APP_HOME"]="E"
    if [[ -z ${CONFIG_MAP["APP_HOME"]:-} ]]; then
        ConsoleFatal " ->" "did not find environment setting for application home,
tried \$${CONFIG_MAP["FLAVOR"]}_HOME"
        printf "\n"
        exit 17
    elif [[ ! -d ${CONFIG_MAP["APP_HOME"]} ]]; then
        ## found home, but is no directory
        ConsoleFatal " ->" "\$${CONFIG_MAP["FLAVOR"]}_HOME set as
${CONFIG_MAP["APP_HOME"]} does not point to a directory"
        printf "\n"
        exit 18
    fi
fi

if [[ -z ${__FW_LOADER_SCRIPTNAME:-} ]]; then
    ConsoleFatal " ->" "interal error: no application script name set"
    printf "\n"
```

```
    exit 20
else
    CONFIG_MAP["APP_SCRIPT"]=${__FW_LOADER_SCRIPTNAME##*/}
fi
if [[ -z "${__FW_LOADER_APPNAME:-}" ]]; then
    ConsoleFatal " ->" "interal error: no application name set"
    printf "\n"
    exit 21
else
    CONFIG_MAP["APP_NAME"]=$__FW_LOADER_APPNAME
fi
source $FW_HOME/bin/loader/declare/app-maps.sh
if [[ -f ${CONFIG_MAP["APP_HOME"]}/etc/version.txt ]]; then
    CONFIG_MAP["VERSION"]=$(cat ${CONFIG_MAP["APP_HOME"]}/etc/version.txt)
else
    ConsoleFatal " ->" "no application version found, tried
\$APP_HOME/etc/version.txt"
    printf "\n"
    exit 22
fi
```

2.5. Temporary Directory

The next step is to test if the temporary directory can be created. This is important for two reasons:

- The directory uses the application flavor in the same, to separate several SKB applications that might be running on the same host
- In the directory, the loader will safe to runtime configuration. This is required because associative arrays (or hash maps, or maps), used to store all configuration data, cannot be exported into the *bash* environment. Using temporary files is the only way for the loader to share the configuration with the shell, and for the shell to share it with tasks.

```
if [[ ! -z ${TMP:-} ]]; then
    TMP_DIRECTORY=${TMP}/${CONFIG_MAP["APP_SCRIPT"]}
else
    TMP_DIRECTORY=${TMPDIR:-/tmp}/${CONFIG_MAP["APP_SCRIPT"]}
fi
if [[ ! -d $TMP_DIRECTORY ]]; then
    mkdir $TMP_DIRECTORY 2> /dev/null
    __errno=$?
    if [[ $__errno != 0 ]]; then
        ConsoleFatal " ->" "could not create temporary directory $TMP_DIRECTORY,
please check owner and permissions"
        printf "\n"
        exit 23
    fi
fi
if [[ ! -w $TMP_DIRECTORY ]]; then
    ConsoleFatal " ->" "cannot write to temporary directory $TMP_DIRECTORY, please
check owner and permissions"
    printf "\n"
    exit 24
fi
```

2.6. Sneak Preview of CLI Arguments

Next, the loader does a sneak preview inside the command line arguments. This is done to determine the application mode that might be requested from the command line. We need to know the requested mode here, before we actually do parse the arguments later. This is important to load the correct declarations for all elements.

```
case "$@" in
    *"-D"* | *"--dev-mode"*)
        CONFIG_MAP["APP_MODE"]="dev"
        CONFIG_SRC["APP_MODE"]="O"
        ;;
    *"-B"* | *"--build-mode"*)
        CONFIG_MAP["APP_MODE"]="build"
        CONFIG_SRC["APP_MODE"]="O"
        ;;
    *"-A"* | *"--all-mode"*)
        CONFIG_MAP["APP_MODE"]="all"
        CONFIG_SRC["APP_MODE"]="O"
        ;;
esac
```

2.7. Parameter Declarations

With the application mode set, the loader can load the parameter declarations. These declarations

can only be loaded from source, i.e. not from an optional cache, since one of the parameters actually sets the cache directory. Once loaded (line 1), all loaded parameters are processed (line 4). This function will load values from the environment, then from an optional configuration file, and finally from potentially declared default values. This function only loads settings, it does not test any of these settings.

```
DeclareParameters
if ConsoleHasErrors; then printf "\n"; exit 25; fi
source $FW_HOME/bin/loader/init/process-settings.sh
ProcessSettings
```

2.8. Option Declarations

Next is the declaration of command line options. This declaration can be done from cache (if cached) or source (if no cache exists). Errors here indicate bugs or runtime problems. Once declared, all options are set to `false` in the CLI map (lines 8-11). This allows to test which options have been used as actual arguments when parsing the command line later.

```
if [[ -f ${CONFIG_MAP["CACHE_DIR"]}/opt-decl.map ]]; then
    ConsoleInfo "-->" "declaring options from cache"
    source ${CONFIG_MAP["CACHE_DIR"]}/opt-decl.map
else
    DeclareOptions
    if ConsoleHasErrors; then printf "\n"; exit 26; fi
fi
declare -A OPT_CLI_MAP
for ID in ${!DMAP_OPT_ORIGIN[@]}; do
    OPT_CLI_MAP[$ID]=false
done
```

2.9. Parse Command Line Arguments

Now the loader can safely parse the command line arguments. The parse function does parse for most options, only set options that have no further side effect. This means that this function does not execute on any option. This is done later in the load process. The print mode is immediately set and tested (lines 3-12) to make sure we have a valid setting for it.

```
ParseCli $@
if ConsoleHasErrors; then printf "\n"; exit 27; fi
case "${CONFIG_MAP["PRINT_MODE"]:-}" in
    ansi | text | adoc)
        ConsoleInfo "-->" "found print mode '${CONFIG_MAP["PRINT_MODE"]}'"
        ;;
    *)
        CONFIG_MAP["PRINT_MODE"]=ansi
        CONFIG_SRC["PRINT_MODE"]=
        ConsoleWarn "-->" "unknown print mode '${CONFIG_MAP["PRINT_MODE"]}', assuming
'ansi'"
        ;;
esac
```

2.10. Realize early Exit Options

At this stage, the loader can realize early exit options. Those are options that do not require any further actions by the loader. These options include:

- Clean the cache (lines 1-5),
- Print the help screen (lines 6-9), and
- Print the application version (lines 10-13)

If any of these options was requested, the loader will exit with code *0*, success.

```
if [[ ${OPT_CLI_MAP["clean-cache"]} != false ]]; then
    ConsoleInfo "-->" "cleaning cache and exit"
    source ${CONFIG_MAP["FW_HOME"]}/bin/loader/options/clean-cache.sh
    exit 0
fi
if [[ ${OPT_CLI_MAP["help"]} != false ]]; then
    source ${CONFIG_MAP["FW_HOME"]}/bin/loader/options/help.sh
    exit 0
fi
if [[ ${OPT_CLI_MAP["version"]} != false ]]; then
    source ${CONFIG_MAP["FW_HOME"]}/bin/loader/options/version.sh
    exit 0
fi
```

2.11. Declarations for Commands and Exit Status Codes

Now the declarations of shell commands (lines 1-7) and exit codes (lines 8-14) can be loaded, either from cache or source. Errors here indicate a framework bug or runtime problem, since commands and exit codes are core features and rather static.

```
if [[ -f ${CONFIG_MAP["CACHE_DIR"]}/cmd-decl.map ]]; then
    ConsoleInfo "-->" "declaring commands from cache"
    source ${CONFIG_MAP["CACHE_DIR"]}/cmd-decl.map
else
    DeclareCommands
    if ConsoleHasErrors; then printf "\n"; exit 28; fi
fi
if [[ -f ${CONFIG_MAP["CACHE_DIR"]}/es-decl.map ]]; then
    ConsoleInfo "-->" "declaring exit-status from cache"
    source ${CONFIG_MAP["CACHE_DIR"]}/es-decl.map
else
    DeclareExitStatus
    if ConsoleHasErrors; then printf "\n"; exit 29; fi
fi
```

2.12. Dependency Declarations

The next step is to load dependency declarations, either from cache or from source. Errors here can point to a bug in the framework, a problem in the application, or a problem with a dependency declaration itself. In this step, dependencies are only declared, but not tested, since the loader still does not know which of them are required by tasks.

```
if [[ -f ${CONFIG_MAP["CACHE_DIR"]}/dep-decl.map ]]; then
    ConsoleInfo "-->" "declaring dependencies from cache"
    source ${CONFIG_MAP["CACHE_DIR"]}/dep-decl.map
else
    ConsoleInfo "-->" "declaring dependencies from source"
    DeclareDependencies
    if ConsoleHasErrors; then printf "\n"; exit 30; fi
fi
```

2.13. Task Declarations

This step is probably the most complicated and most important step in the load process. First, the declarations of tasks are loaded, either from cache or from source (lines 1-8). Errors here can point to a bug in the framework, a problem in the application, or a problem with a task declaration itself. Next, the loaded tasks are processed (line 9-11). This function will take the loaded tasks and test or validate all parameters and dependencies they require. This process can take some time, especially for testing external dependencies. Errors tend to indicate configuration or dependency problems, not internal or declaration problems.

Some tasks might declare requirements as *optional*. Those dependencies only throw warnings in a normal application run. Only if an application is run in *strict* mode will those problems throw errors.

```
if [[ -f ${CONFIG_MAP["CACHE_DIR"]}/task-decl.map ]]; then
    ConsoleInfo "-->" "declaring tasks from cache"
    source ${CONFIG_MAP["CACHE_DIR"]}/task-decl.map
else
    ConsoleInfo "-->" "declaring tasks from source"
    DeclareTasks
    if ConsoleHasErrors; then printf "\n"; exit 31; fi
fi
source $FW_HOME/bin/loader/init/process-tasks.sh
ProcessTasks
if ConsoleHasErrors; then printf "\n"; exit 32; fi
```

2.14. Scenario Declarations

When tasks are declared, the loader can declare all scenarios from the framework and application home as well as the optional scenario path.

```
ConsoleInfo "-->" "declaring scenarios from source"
DeclareScenarios
if ConsoleHasErrors; then printf "\n"; exit 33; fi
source $FW_HOME/bin/loader/init/process-scenarios.sh
ProcessScenarios
if ConsoleHasErrors; then printf "\n"; exit 34; fi
```

2.15. Set Levels

The next step is to set the correct levels (log levels) for the loader (the remaining load steps), the shell, and the tasks. This is done for each level type in separation.

```
case "${CONFIG_MAP["LOADER_LEVEL"]}" in
    off | all | fatal | error | warn-strict | warn | info | debug | trace)
        ;;
    *)
        ConsoleError "-->" "unknown loader-level: ${CONFIG_MAP["LOADER_LEVEL"]}"
        printf "    use: off, all, fatal, error, warn-strict, warn, info, debug,
trace\n\n"
        exit 35
        ;;
esac
case "${CONFIG_MAP["SHELL_LEVEL"]}" in
    off | all | fatal | error | warn-strict | warn | info | debug | trace)
        ;;
    *)
        ConsoleError "-->" "unknown shell-level: ${CONFIG_MAP["SHELL_LEVEL"]}"
        printf "    use: off, all, fatal, error, warn-strict, warn, info, debug,
trace\n\n"
        exit 36
        ;;
esac
case "${CONFIG_MAP["TASK_LEVEL"]}" in
    off | all | fatal | error | warn-strict | warn | info | debug | trace)
        ;;
    *)
        ConsoleError "-->" "unknown task-level: ${CONFIG_MAP["TASK_LEVEL"]}"
        printf "    use: off, all, fatal, error, warn-strict, warn, info, debug,
trace\n\n"
        exit 37
        ;;
esac
```

2.16. Do (Exit) Options

At this stage, the loader can process most the remaining exit options. Those are command line options that request some behavior and then should cause the loader to exit.

If such options are requested, the loader might provide some information about its execution time (lines 5-11).

```
source $FW_HOME/bin/loader/init/do-options.sh
DoOptions
if ConsoleHasErrors; then printf "\n"; exit 38; fi

if [[ $DO_EXIT == true ]]; then
    _te=$(date +%s.%N)
    _exec_time=$_te-$_ts
    ConsoleInfo "-->" "execution time: $(echo $_exec_time | bc -l) sec"
    ConsoleInfo "-->" "done"
    exit 0
fi
```

2.17. Create Runtime Configuration File

Now, the loader needs to create the temporary runtime configuration file. The remaining actions are either to run a execute a task, run a scenario, or execute the interactive shell. All of these actions require the configuration to be available. The configuration file is created in the already tested directory, usually located in `/tmp/`. The name of the configuration files includes a time stamp of its creation and an arbitrary, random string. `mktemp` is used to create the file name. Since the name is unique, the same application can be executed many times simultaneously.

```
CONFIG_MAP["FW_L1_CONFIG"]=$(mktemp "$TMP_DIRECTORY/$(date +"%H-%M-%S")-
${CONFIG_MAP["APP_MODE"]}-XXX")
export FW_L1_CONFIG=${CONFIG_MAP["FW_L1_CONFIG"]}
WriteRuntimeConfig
```

2.18. Execute Task or Scenario

The final exit options are to execute a task or to run a scenario. The loader will try either of them.

```
__errno=0
if [[ "${OPT_CLI_MAP["execute-task"]}" != false ]]; then
    echo ${OPT_CLI_MAP["execute-task"]} | $FW_HOME/bin/shell/shell.sh
    __et=$?
    __errno=$((__errno + __et))
fi
if [[ "${OPT_CLI_MAP["run-scenario"]}" != false ]]; then
    echo "execute-scenario ${OPT_CLI_MAP["run-scenario"]}" |
$FW_HOME/bin/shell/shell.sh
    __et=$?
    __errno=$((__errno + __et))
    DO_EXIT_2=true
fi
```

2.19. Start Shell

This is the final step of the load process. If no task or scenario was requested to be executed, the loader will start the interactive shell. This shell will run until a user caused it or exit, or until an error terminated the whole application.

```
if [[ ${DO_EXIT_2} == false ]]; then
    $FW_HOME/bin/shell/shell.sh
    __errno=$?
fi
```

2.20. Clean Up

Finally, the loader can cleanup and prepare to terminate. Cleanup basically means to remove the temporary configuration file. If the temporary directory is no longer needed, i.e. no other configuration exists in it, it can be removed as well.

```
if [[ -f $FW_L1_CONFIG ]]; then
    rm $FW_L1_CONFIG >& /dev/null
fi
if [[ -d $TMP_DIRECTORY && $(ls $TMP_DIRECTORY | wc -l) == 0 ]]; then
    rmdir $TMP_DIRECTORY
fi
```

2.21. Done

The loader is done, all actions have been taken. It now can safely exit and return the execution to the `skb-framework` or the calling application. A final message is displayed to mark this point in the process.

```
ConsoleMessage "\n\nhave a nice day\n\n\n"
exit $__errno
```

3. The Shell

The loader is the script `$FW_HOME/bin/shell/shell.sh`. It provides an interactive shell to either execute shell commands or tasks. Most shell commands are realized by tasks, so the actual shell implementation is rather simple.

The initial settings are shown in the source block below. Line 1 restricts *bash*, providing a safer execution environment.

```
set -o errexit -o pipefail -o noclobber -o nounset
```

3.1. Test Parent and and load Configuration

The shell then tests if it is run from the right parent (i.e. the loader) and it loads the temporary configuration. The parent test is rather simple: if `FW_HOME` or `FW_L1_CONFIG` are not set, it is very likely that the shell has not been started by the loader. Otherwise, the temporary configuration can be loaded (line 5). The configuration is set for *running in the shell* (line 6). This information is used by the API functions in the console to determine what output (log) level and what error/warning counters to use.

```
if [[ -z ${FW_HOME:-} || -z ${FW_L1_CONFIG-} ]]; then
    printf " ==> please run from framework or application\n\n"
    exit 40
fi
source $FW_L1_CONFIG
CONFIG_MAP["RUNNING_IN"]="shell"
```

3.2. Include

Next, the shell includes the framework's API functions (lines 1-2) and the functions to maintain its command history (line 3). Then the error and warning counters are reset (set to *0*), and a simple new line is printed (if messages are allowed).

```
source $FW_HOME/bin/api/_include
source $FW_HOME/bin/api/describe/task.sh
source $FW_HOME/bin/shell/history.sh

ConsoleResetErrors
ConsoleResetWarnings
ConsoleMessage "\n"
```

3.3. Settings

Now the shell realizes its core settings. These settings are:

- *SCMD* - used to store shell input
- *SARG* - used to store arguments (only used for history now)
- *STIME* - used to set the time when a command was entered, and later for time calculations
- *RELOAD_CFG* - a flag to indicate if the temporary configuration should be reloaded. This flag is required for instance when the task `set` alters settings.
- *HISTORY* - a map with the history of commands #end::settings[]

```
SCMD=                           # a shell-command from input
SARG=                           # argument(s), if any, for a shell command
STIME=                          # time a command was entered
RELOAD_CFG=false                # flag to reload configuration, e.g. after a change of
settings
declare -A HISTORY              # the shell's history of executed commands
HISTORY[-1]="help"              # dummy first entry, size calculation doesn't seem to
work otherwise
```

3.4. Inner Loop: FWInterpreter

The inner loop is an interpreter for all input the shell wants to add to its history. This input might be the request to execute a task, or a command that the shell realizes using a task, or a simple command. The inner loop is defined as a function.

```
FWInterpreter() {
    case "$SCMD" in
        # ...
    esac
}
```

Inside this function, the inner loop, the shell tests first for all input that it can associate to a command. If none of these tests is satisfied, it assumes that the input is actually a task to be executed with parameters.

3.4.1. Execute a Scenario

The first command tested is to execute a scenario. This command starts with either `execute-scenario` or `es`. No argument means error. Some argument means the name of a scenario. In this case, execute the scenario and put the command into the history.

```
        execute-scenario | es)
            printf "\n    execute-scenario/rs requires a scenario as argument\n\n"
            ;;
        "execute-scenario "*)
            SARG=${SCMD#*execute-scenario }
            ExecuteScenario $SARG
            ShellAddCmdHistory
            ;;
        "es "*)
            SARG=${SCMD#*es }
            ExecuteScenario $SARG
            ShellAddCmdHistory
            ;;
```

3.4.2. Clear Screen

This a simple command without arguments. The clear screen functionality is realized by printing the ANSI escape sequence for clear screen.

```
        clear-screen | "clear-screen "* | cls | "cls "*)
            printf "\033c"
            ShellAddCmdHistory
            ;;
```

3.4.3. Time

Print the current time.

```
        time | "time "* | T | "T "*)
            printf "\n    %s\n\n" "$STIME"
            ShellAddCmdHistory
            ;;
```

3.4.4. Print Configuration

Execute the task `list-configuration` with default settings, which displays a list with the current configuration.

```
        configuration | "configuration "* | c | "c "*)
            ${DMAP_TASK_EXEC["list-configuration"]}
            ShellAddCmdHistory
            ;;
```

3.4.5. Print Statistics

Execute the task `statistics` with default settings, which displays an overview of statistic information.

```
        statistic | "statistic "* | s | "s "*)
            ${DMAP_TASK_EXEC["statistics"]}
            ShellAddCmdHistory
            ;;
```

3.4.6. List Tasks

Execute the task `list-tasks` with default settings, which displays an list of loaded tasks.

```
        tasks | "tasks "* | t | "t "*)
            ${DMAP_TASK_EXEC["list-tasks"]}
            ShellAddCmdHistory
            ;;
```

3.4.7. Comments

Do nothing if the input starts with the comment character `#` in any variation.

```
        "" | "#" | "#"* | "# "*)
            ;;
```

3.4.8. All other Input

All other input is interpreted as the request to execute a task with optional arguments.

```
        *)
            SARG="$SCMD"
            ExecuteTask "$SARG"
            ShellAddCmdHistory

            case "$SCMD" in
                "set "* | "setting "*) RELOAD_CFG=true;;
            esac
            ;;
```

3.5. Outer Loop: FWShell

The outer loop is the shell's main loop reading input from standard input and interpreting it. While input is read (a line finished with an enter), the line is read into the variable `SCMD` and a time stamp is written into `STIME`. Then `SCMD` is evaluated.

```
FWShell() {
    while read -a args; do
        SCMD="${args[@]:-}" <&3
        STIME=$(date +"%T")
        case "$SCMD" in
            # ...
        esac

        if [[ $RELOAD_CFG == true ]]; then
            source $FW_L1_CONFIG
            CONFIG_MAP["RUNNING_IN"]="shell"
            RELOAD_CFG=false
        fi
        if ConsoleIsPrompt; then ConsoleMessage "${CONFIG_MAP["SHELL_PROMPT"]}"; fi
    done
}
```

Inside this function, the outer loop, the shell tests first for shell commands. If none of these tests is satisfied, it calls the inner loop to deal with the input.

Once finished with the input, the value of `RELOAD_CFG` to see if the configuration has to be reloaded. Finally, a new prompt is displayed.

3.5.1. Help

If help is requested, display the command help file for the current print mode.

```
            help | h | "?")
                cat
${CONFIG_MAP["FW_HOME"]}/etc/help/commands.${CONFIG_MAP["PRINT_MODE"]}
                ;;
```

3.5.2. History

History means to print the history or to run a commend stored in the history. Both functionalities are provided by the history function.

```
            !*)
                SARG=${SCMD#*!}
                ShellCmdHistory
                ;;
            history*)
                SARG=${SCMD#*history}
                ShellCmdHistory
                ;;
```

3.5.3. Exit

If exit is requested, the shell leaves the outer loop.

```
            exit | quit | q | bye)
                break
                ;;
```

3.5.4. All other Input

All other input is forwarded to the inner loop for further evaluation.

```
            *)
                FWInterpreter
                ;;
```

3.6. Run the Shell

The final lines run the actual shell. First, input is redirected to #3 while the shell is running (line 1). This allows to read lines from the standard input into the shell. Next, the first prompt is displayed (line 2). Then the outer loop is called (line 3). When the outer loop is finished, the redirection is reverted (line 4). Now the shell is finished and the process returns to the loader.

```
exec 3</dev/tty || exec 3<&0
if ConsoleIsPrompt; then ConsoleMessage "${CONFIG_MAP["SHELL_PROMPT"]}"; fi
FWShell
exec 3<&-
```