WorkBench Documentation

Release 0.0.1

Praveen G Shirali

Feb 27, 2019

Getting Started:

WorkBench is a hierarchical environment manager for *nix shells. It sources shell-code distributed across multiple levels of a folder hierarchy and invokes environments with the combination. Code could thus be implemented to operate at different scopes, allowing clear overrides at each folder depth and easy overall maintenance while managing several hundred environments.

WorkBench is a minimalistic framework. It is extendable and configurable, and can adapt to a variety of use-cases. It is implemented as a single bash script, and designed to work with minimal dependencies even on vanilla *nix systems.

CHAPTER 1

Installation

WorkBench is under active development. A package based installer is currently not available.

1.1 Install from source

WorkBench (`wb`) is a single bash file. You can use `curl` or `wget` to fetch the bleeding edge `wb` script from the WorkBench repository.

1.1.1 Using curl

```
curl -fsSL https://raw.githubusercontent.com/pshirali/workbench/master/wb > wb
```

1.1.2 Using wget

```
wget https://raw.githubusercontent.com/pshirali/workbench/master/wb
```

Ensure that `wb` is placed in a folder which is in your `PATH`. Set the execute bit for `wb` by running `chmod +x <path/to/wb>`

CHAPTER 2

Configuration

WorkBench accepts its configuration from environment variables which have the prefix `WORKBENCH_`. WorkBench comes with sane defaults and external configuration is optional.

2.1 Configuration using rcfile

WorkBench can also source an *rcfile* on invocation. The default location for the *rcfile* is `$HOME/.workbenchrc`. If a file at the default location exists, then it is automatically sourced.

A custom *rcfile* can be specified using the environment variable `WORKBENCH_RC` pointing to a file that already exists.

The *rcfile* can be used to define multiple configuration parameters at once.

Note: The rcfile overrides environment variables defined in the shell.

A full list of configurable parameters are available in subsequent chapters.

2.2 The WorkBench Home directory

WorkBench operates on files inside a directory defined by `WORKBENCH_HOME`. If `WORKBENCH_HOME` is undefined, the default home directory `$HOME/.workbench` is used. WorkBench automatically creates the necessary folder(s) on invocation.

CHAPTER 3

Completion

A bash completer for WorkBench is available in the `completion` subdirectory of the WorkBench repo.

To deploy it:

1. Download the file `completion/wb_complete.bash`
2. Add the following line to your `.bashrc` or `.bash_profile` where <*path/to*> is the directory where `wb_complete.bash` is located

```
source "<path/to>/wb_complete.bash"
```

CHAPTER 4

Introduction to subshells

Note: Skip this section if you are already familiar with bash subshells

Consider a file `abcd` with the contents below:

```
export ABCD=10
show_abcd () {
    echo "The value of ABCD is ${ABCD}"
}
alias c=clear
```

A bash subshell could be invoked using:

```
bash --rcfile ./abcd
```

While the prompt remains the same, a new interactive shell is now active. In this state, the following behavior can be observed:

```
>> echo $ABCD           # value from the environemnt variable is printed
10

>> show_abcd           # a bash function is invoked
The value of ABCD is 10

>> c                   # alias for `clear`. Clears the screen.

>> exit                # exits the subshell
```

On *exit* all context from the subshell is lost. It may be observed that executing the same commands in the parent shell does not result in the same behavior as what was seen in the subshell.

An environment is a subshell initialised with environment variables, functions or aliases which caters specifically to a project or a task at hand.

By using environments:

1. The parent shell's namespace remains free of project-specific declarations
2. Declarations are local to each environment. Commands and variables by the same name could be declared in each environment, which perform operations unique to that environment.
3. It is easy to exit from the subshell and unload the entire environment at once.

CHAPTER 5

WorkBench Concepts

5.1 Introduction

WorkBench makes it easy to work with a large number of custom shell environment scripts, each of which could be tailor-made for a project or task.

WorkBench sources shell code spread across different depths of a directory tree to construct an environment automatically. Code could thus be implemented in parts, residing in files at different directory depths, and without any hardcoded references.

WorkBench operates only on files present inside a directory as defined by `WORKBENCH_HOME`. It uses two abstract terms to refer to parts of a to-be-assembled environment; namely `Shelf` and `Bench`.

5.2 Shelves and Benches

EXAMPLE: Consider a `WORKBENCH_HOME` with the following structure:

```
WORKBENCH_HOME
    ├── ash.bench               # BENCH
    ├── bar/
    │   ├── baz/
    │   │   ├── maple.bench     # BENCH
    │   │   └── wb.shelf                        # SHELF
    │   └── birch.bench         # BENCH
    ├── foo/
    │   ├── pine.bench          # BENCH
    │   └── wb.shelf                            # SHELF
    └── wb.shelf                                # SHELF
```

5.2.1 Shelf

A `shelf` is `WORKBENCH_HOME`, or any subdirectory inside it, which contains the file as defined by `WORKBENCH_SHELF_FILE`. The default value for `WORKBENCH_SHELF_FILE` is `wb.shelf`. A `shelf` is always a path relative to `WORKBENCH_HOME`. Shelf names end with a trailing `/` as they represent the directory containing `WORKBENCH_SHELF_FILE` and not the file itself.

In the example above, the file `wb.shelf` is present at three locations. Hence, there are three shelves here.

```
/
foo/
bar/baz/
```

The table below maps the name of the *shelf* to the underlying resource file:

Shelf Name	Underlying resource filename
/	WORKBENCH_HOME/wb.shelf
foo/	WORKBENCH_HOME/foo/wb.shelf
bar/baz/	WORKBENCH_HOME/bar/baz/wb.shelf

The subdirectory `bar/` is not a *shelf* because it doesn't contain `wb.shelf`.

5.2.2 Bench

A `bench` is a file anywhere inside `WORKBENCH_HOME` with the extension as defined by `WORKBENCH_BENCH_EXTN`. The default value for `WORKBENCH_BENCH_EXTN` is `bench`. The extension separator `.` is assumed automatically and is not part of the value. Bench names are representative of files. They do not include the trailing `.<WORKBENCH_BENCH_EXTN>`

In the example above, there are four files with a `.bench` extension. Hence, four benches.

```
ash
bar/baz/maple
bar/birch
foo/pine
```

The table below maps the name of the *bench* to the underlying resource file:

Bench Name	Underlying resource filename
ash	WORKBENCH_HOME/ash.bench
bar/baz/maple	WORKBENCH_HOME/bar/baz/maple.bench
bar/birch	WORKBENCH_HOME/bar/birch.bench
foo/pine	WORKBENCH_HOME/foo/pine.bench

5.3 Analogy

Analogous to a real workbench, the top of a *bench* is where the work gets done. A discerning artisan might place minimal tools required for the task at hand on the *bench*, while rest of the tools might be placed in *shelves*, each of which ordered based on the frequency in which they get used; frequently used tools being closer than infrequent ones.

The abstract *shelf* (a place to stow tools) may also be imagined as a pegboard where tools are hung for easy access. An artisan can locate any tool quickly, use it and put it back.

In WorkBench, the *shelf* hierarchy is provided by the possible presence of the `WORKBENCH_SHELF_FILE` at different directory depths leading upto the `Bench`.

Code that is declared in a `Shelf` at the root; that is `WORKBENCH_HOME` will be sourced by every *workbench*. Code that is applicable only to a specific set of environments could be defined in `Shelves` in a subdirectory at the appropriate depth. Thus declarations & implementations common to multiple environments get organised into `Shelves`, while declarations which uniquely associate with one environment get placed in a `Bench`.

A pegboard approach could also be implemented by declaring functions in various `Shelves` but not calling them. The `Bench` would call those functions with various parameters for the task at hand.

5.4 Benefits

1. Overall there is less code to maintain.
2. It is easy to influence control on a whole group of environments by moving code to a `Shelf` at the appropriate subdirectory
3. Redeclaration results in overriding. Code in a *shelf* at a deeper depth overrides those at lower depths (closer to `WORKBENCH_HOME`). Code in a *bench* overrides all *shelves*. The workbench *tree* could be designed to be shallow, or deeply nested to cater to the amount of overriding required.
4. The hierarchical structure lends itself to organising and managing a tree of hundreds of *benches* easily.

CHAPTER 6

Usage Guide

WorkBench has a minimal set of commands. They are also short (usually one character).

Note: The following convention denotes `OR`. Example: `wb a|b|c` means `wb a` OR `wb b` OR `wb c`

6.1 View version and env – [`wb -V`, `wb -E`]

`wb -V` prints the version of WorkBench being used.

`wb -E` lists all environment variables starting with `WORKBENCH_`. These environment variables may be defined in your current shell, or may be defined in a `WORKBENCH_RC` file.

If you use an *rcfile* with WorkBench, the values you set in the *rcfile* will apply over everything else. The *rcfile* is sourced on every `wb` invocation regardless of the command.

6.2 Operating on Shelves and Benches – [`wb s`, `wb b`]

The following operations can be performed on *shelves* and *benches*:.

6.2.1 List

WorkBenches can be listed using `wb s|b`

6.2.2 Print path to the underlying file

`wb s|b <name>`, where <*name*> is either a <*shelfName*> or <*benchName*> prints the absolute path to the underlying resource file associated with that shelf or bench.

The path is generated and displayed for non-existent shelves and benches as well. A non-zero exit-code is returned if a shelf or bench doesn't exist.

6.2.3 Run a command against the underlying file

`wb s|b [options] <name> <command> [[arg]..]`

Runs `<command> [[arg]..] <path-to-underlying-file-for-name>`

Examples:

```
wb s <shelfName> cat       # view the file WORKBENCH_HOME/.../<shelf-file>
wb b <benchName> vim       # edit the file WORKBENCH_HOME/.../<bench-file> in ViM
```

Commands execute only when a <shelfName> or <benchName> exist on disk. It is possible to create a new *shelf* or *bench* inline, just before running a command on it by adding the `--new` switch.

```
wb s --new <newShelfName> vim
```

WorkBench prompts for confirmation if the <*command*> is `rm`. The `--yes` switch can be used to indicate *Yes* to skip the prompt. Alternatively, `WORKBENCH_AUTOCONFIRM` can be set to any non-empty value to disable this prompt and assume *Yes* always.

6.3 Auto-generated *workbench* and Entrypoints

A *workbench* is the auto-generated code composed by WorkBench (the tool), when the command `wb a|r|n <benchName>` is executed.

The switch `--dump` can be used to print the auto-generated code on *stdout* instead of executing it. The `--dump` switch does not validate the presence of a <*benchName*>. This switch can be used to review the generated code.

The auto-generated *workbench* has the following high-level sections:

```
┌──────────────┐
│     INIT     │   <---- Initial declarations are done here
├──────────────┤
│    SOURCE    │   <---- Shelves and bench are `sourced` here
├──────────────┤
│  ENTRYPOINT  │   <---- Entrypoint function is called with `args`
└──────────────┘
```

The *INIT* section of the *workbench* contains basic/no-op implementations for the default functions. *Shelves* and *Bench* are expected to define their own functions with an actual implementation to override those in *INIT*.

The *INIT* section defines the following variables:

1. `WORKBENCH_ENV_NAME`: Stores the *benchName* as the environment name
2. `ORIG_PS1`: Stores the current `PS1`, while `PS1` is reset to prefix the current *benchName*
3. `WORKBENCH_CHAIN`: Stores a `:` separated list of each sourced *shelf* and *bench* in the order in which they were sourced.

An entrypoint is a shell functions invoked after sourcing all the *shelves* and the *bench*. Each WorkBench execution command has a different *entrypoint* function associated with it. Any trailing arguments passed to the WorkBench's execution command are passed on to the *entrypoint*.

Entrypoint function names are configurable. The table below lists the environment variables which define the *entrypoints* and the default function names associated with each of them.

Type	Environment Variable Name	Default Function Name	Command
entrypoint	WORKBENCH_ACTIVATE_FUNC	workbench_OnActivate	a
entrypoint	WORKBENCH_RUN_FUNC	workbench_OnRun	r
entrypoint	WORKBENCH_NEW_FUNC	workbench_OnNew	n

The *entrypoint* is invoked by calling the entrypoint environment variable. Thus the value of the entrypoint environment variable can be redefined in the *shelf* or the *bench* to point to a non-default function as well.

6.3.1 Entrypoint Example

When `wb r <benchName> arg1 arg2` is executed, then the function that maps to `WORKBENCH_RUN_FUNC` becomes the actual entrypoint. The default entrypoint function name is `workbench_OnRun` and the *INIT* section has an implemetation for it.

A *shelf* or *bench* could redeclare `workbench_OnRun` multiple times; in files at different depths. The last declared implementation will be the one that executes with arguments `arg arg2`

It is also possible that `WORKBENCH_RUN_FUNC` could be assigned a new value like `my_custom_func` anywhere in the *shelves* or the *bench*. The last declared value of `WORKBENCH_RUN_FUNC` is now the new entrypoint function, and the last declared implementation of the function `my_custom_func` is the one that executes with arguments `arg1 arg2`

6.4 Executing *workbench* environments – [`wb a`, `wb r`, `wb n`]

The *workbench* stores the execution command in the varible `WORKBENCH_EXEC_MODE`. *Shelf* and *Bench* code could take decisions based on this value.

A no-op function `workbench_pre_execute_hook` executes just before a *workbench* is built. This function could be implemented by the `WORKBENCH_RC` with logic that decides whether to go ahead with execution. Refer to the *Security* chapter for more details.

6.4.1 Activate – [`wb a`]

The *activate* command is equivalent of `bash --rcfile <workbench>`. It spawns a subshell with the auto-generated *workbench*, with `WORKBENCH_ACTIVATE_FUNC` as the entrypoint.

Nested *activations* are prevented by checking if `WORKBENCH_ENV_NAME` has already been set.

Deactivating a *workbench* is done by simply running *exit*.

Occasionally, there may be cases where some code needs to be executed when an *exit* is issued. This can be achieved by redeclaring the *exit* function, calling user-defined code, followed by calling *builtin exit*.

Example:

```
exit () {
    <your-deactivation-code-goes-here>
    builtin exit $? 2> /dev/null
}
```

6.4.2 Run – [wb r]

The *run* command is the equivalent of `bash -c <workbench>`. It executes the *workbench* non-interactively, with `WORKBENCH_RUN_FUNC` as the entrypoint. The *run* command is used to invoke one-off commands which may be defined in the *workbench*.

For example, a *workbench* could declare subcommands like `start`, `stop`, `build`, `deploy` etc, as independent functions. The entrypoint function defined by `WORKBENCH_RUN_FUNC` could parse arguments and dispatch them to respective subcommands.

Thus, for the same *workbench*, the *activate* and *run* commands could be used to trigger different functionality.

6.4.3 New – [wb n]

The *new* command is a variant of the *run* command. It's execution is similar to that of the *run* command (non-interactive), but with `WORKBENCH_NEW_FUNC` as the entrypoint.

When the command `wb n <newBenchName>` is invoked, WorkBench creates all intermediate *shelf* files (if they don't already exist) followed by the *bench*. The *bench* must either not exist, or must be a zero-byte file.

The last declared function as defined by `WORKBENCH_NEW_FUNC` is then called, which is expected to write contents into the new *bench*.

Consider a programming language like Python, Go etc. All projects of a language would require a common set of steps to build up a workspace for the language. For Python, tools like *virtualenv*, with *virtualenvwrapper* are already available. Similar tools exist for other languages too.

It is easy to implement code in a *shelf* to define the behavior for all projects for a particular language/group. The code could wrap around an existing tool (like *virtualenv*) or provide all functionality by itself.

The aspect that varies between each project of a language might be: (a) Name, (b) Project URL, may be (c) language version etc. But, such values are few. The *shelf's* implementation of `WORKBENCH_NEW_FUNC` could request this information for a new project and dump the metadata into the *bench*. The *bench* could therefore be minimal; may be an *env* file with key-values.

CHAPTER 7

Environment Variables

The table below contains a list of environment variables which WorkBench consumes in its configuration.

Environment Variable Name	Default Value	Description
WORKBENCH_RC	$HOME/.workbenchrc	Auto-load location for the rcfile
WORKBENCH_HOME	$HOME/.workbench	Directory containg shelves and benches
WORKBENCH_ALLOW_INSECURE_PATH		Skips using 'realpath' if set.
WORKBENCH_GREPPER	egrep	Grep tool used to list env. vars
WORKBENCH_AUTOCONFIRM	–	Skip confirmation prompt for *rm* if set
WORKBENCH_SHELF_FILE	wb.shelf	Filename for the shelf file
WORKBENCH_BENCH_EXTN	bench	File extension for the bench file
WORKBENCH_ACTIVATE_CMD	/bin/bash –rcfile	Command to invoke subshell in intereactive mode
WORKBENCH_COMMAND_CMD	/bin/bash -c	Command to invoke a script in non-interactive mode
WORKBENCH_ACTIVATE_FUNC	workbench_OnActivate	Entrypoint function name for the *activate* command
WORKBENCH_RUN_FUNC	workbench_OnRun	Entrypoint function name for the *run* command
WORKBENCH_NEW_FUNC	workbench_OnNew	Entrypoint function name for the *new* command

The table below contains a list of environment variables which are injected as part of the auto-generated *workbench*.

Environment Variable Name	Description
WORKBENCH_ENV_NAME	Name of the currently active *bench*
WORKBENCH_EXEC_MODE	The mode in which the workbench was launched. One of 'a', 'c', 'n'
WORKBENCH_CHAIN	A `:` separated list of every sourced shelf and bench
ORIG_PS1	Stores the existing `PS1` before redefining it

CHAPTER 8

Exit Codes

WorkBench exits with different exit-codes when it encounters errors. The table below lists the error names, exit-codes and a description.

Error Name	ExitCode	Description
ERR_FATAL	1	General/fatal errors.
ERR_MISSING	3	Resource does not exist.
ERR_INVALID	4	Failed input validation.
ERR_DECLINED	5	Opted *No* on confirmation prompt
ERR_EXISTS	6	Resource already exists.

CHAPTER 9

Security

WorkBench is a bash script capable of executing shell code from your system. This page discloses some of the inner-workings of WorkBench for user awareness. It also covers guidelines and best-practices that you should follow to securely use WorkBench.

It is vital that you understand the contents of this page before you use WorkBench. A discerning user might find the contents here a tad verbose. However, it is in the best interest of a potential user.

This document assumes that you are already operate a secure system where the statements below (but not limited to) are true:

1. You trust the OS binaries that are installed.
2. Only you have access to the contents of your *user* directory. (*$HOME*)
3. You own and understand the contents of your shell's rcfiles. (like *.bashrc*).

The rest of the this page discusses how WorkBench fits in, and the baggage that it brings with it.

9.1 Single User Context

WorkBench is designed for a single-user. You should use WorkBench on systems where you (as a *nix user), and ONLY YOU own and are in are in complete control of:

1. WorkBench (the tool), and the location where it is deployed.
2. The location(s) and contents of all *WORKBENCH_RC* files
3. The location(s) and contents of all *WORKBENCH_HOME* folders (the entire tree)

Every time you run *wb*, you are not only executing the code in *wb*, but also the contents of the *rcfile*. The default location `$HOME/.workbenchrc` is tried if *WORKBENCH_RC* is not defined. The *rcfile* here is a shell script. The code within it will execute even without the *execute* permission set on the file (similar to your *.bashrc*, *.bash_profile*)

Depending on values defined against *WORKBENCH_SHELF_FILE* and *WORKBENCH_BENCH_EXTN*, all files matching the filename and extension respectively within your *WORKBENCH_HOME* are assumed to be shell scripts.

Code from these files will be sourced when you invoke *wb a*, *wb r* or *wb n* commands. As above, they too don't require the *execute* permission to be set on them.

WorkBench is not in control of any files or commands that may be sourced or executed within *shelves* or *benches*. It is possible that code (content within the *shelf* or *bench*) might source files outside of *WORKBENCH_HOME*, or outside of your *user* directory too.

The name of the *entrypoint* function that WorkBench executes can be redefined using *WORKBENCH_ACTIVATE_FUNC*, *WORKBENCH_RUN_FUNC* and *WORKBENCH_NEW_FUNC* respectively. Depending on the command invoked, the control will land on one of these functions.

The values for these variables can be replaced by any executable binary or an existing definition in your current shell (parent shell).

Example:

```
WORKBENCH_RUN_FUNC=echo wb r <benchName> Hello World

Will print "Hello World" in the last line of the command's output.
```

9.1.1 Guidelines

1. Ensure/change the ownership of `wb`, *rcfiles* and all contents of *WORKBENCH_HOME* to you (as a user). Ensure that *write* and *execute* permissions are not available for *group* and *all*.

   Ideal permissions:

   ```
   chmod 0700 <wb>
   chmod 0600 <rcfiles>
   chomd 0600 <WORKBENCH_HOME and all its files>
   ```

2. Do not introduce new content into *WORKBENCH_HOME* that you haven't personally written or reviewed.

   For example:

   (a) Do not extract archives inside *WORKBENCH_HOME*.

   (b) Do not `git clone` repositories into your *WORKBENCH_HOME*.

   You must treat the contents of *WORKBENCH_HOME* in the same light as your *rcfiles*, *dotfiles* etc.

9.2 Detecting changes in your WORKBENCH_HOME

WorkBench provides a function `workbench_pre_execute_hook` which allows you to implement your own *pre* checks before executing a *workbench*. This is an ideal place to implement checks to track change to *WORKBENCH_HOME*.

You can implement `workbench_pre_execute_hook` as a function inside your *WORKBENCH_RC*. If the function returns with a non-zero return code, WorkBench will exit with that code.

Perhaps the easiest way to achieve this would be to turn your *WORKBENCH_HOME* into a Git repo and let Git track your changes. (Example: `git status -s`)

9.3 Canonical paths and directory traversal

WorkBench uses the `realpath` (GNU) utility to convert all relative paths to absolute paths before operating on them. WorkBench ensures that every *shelf* and *bench* that gets sourced as part of building a *workbench*, also reside within `WORKBENCH_HOME`.

Important:

1. `WORKBENCH_RC` and `WORKBENCH_HOME` are excluded from checks. It is highly recommended that they reside inside your `HOME` directory, but this is not enforced.
2. WorkBench does not detect *source* statements inside the code residing in a *shelf* or *bench*. Placing such *source* statements is discouraged. If you do, then you should ensure that you *source* it from locations within `WORKBENCH_HOME`.

It is possible that `realpath` might not be present on every OS, and you might have to install it before using WorkBench.

WorkBench also provides a way to disable this feature. You can do so by setting `WORKBENCH_ALLOW_INSECURE_PATH` to any value to disable directory traversal checks.

9.3.1 What is a directory traversal attack? How is it harmful?

Directory traversal attack is a way by which software is made to expose or operate on files outside a directory boundary. It takes the form of an *attack* when it is used with malicious intent. WorkBench implements checks largely to prevent inadvertent sourcing of content.

A directory traversal attack involves a *path* derived from user input which includes `../`. This indicates the parent of the intended directory. With directory traversal checks disabled, one could supply a command like: `wb r ../benchName` to access a *shelf* and a *bench* that is located at the parent directory of `WORKBENCH_HOME`. The input could include multiple `../` to craft a *path* that points to any other file on your drive.

Note: WorkBench strips preceeding `/` from *shelf* and *bench* names, and makes them relative to *WORKBENCH_HOME*. This handles the case of input *shelf* or *bench* names supplied as absolute paths.

9.4 Temp files

WorkBench creates temp files with the auto-generated *workbench* contents when the commands `wb a`, `wb r`, `wb n` are executed without the `--dump` switch. The temp files are created using `mktemp` utility. This creates a file within `/tmp` with the content that you see in the `--dump` switch. The temp files have a default permission *0600* which makes them accessible to only you, the user. WorkBench deletes the temp file after the command completes execution.

CHAPTER 10

Contribution

You are welcome to contribute to the WorkBench project.

WorkBench has been built with the philosophy of *less-is-more*. It is nearly feature complete. No big features are planned. Improvements however are always welcome.

10.1 Where can I contribute?

You can contribute to:

1. Discussing and suggesting improvements to provide hooks or tweaks, such that WorkBench could be adopted for use in more scenarios.
2. Testing: WorkBench compatibiltiy tests are work-in-progress. This involves testing against various bash versions and against other shells (zsh, ash, etc)
3. A plan to start a Wiki is on the cards, where you can contribute your ideas and recipies on the best ways to use WorkBench.

10.2 How do I start?

You must start a discussion by opening a Github Issue first. You'll be guided on the next steps through the discussion. PR which don't go through this route will probably be rejected.

CHAPTER 11

Testing

WorkBench was built with TDD. Unittests are written in Python3's unittest framework. They are best run with Python 3.7.

Tests can be run by cloning the repo and executing `make test`

Code coverage is on the cards using `bashcov`. This can be taken up after an enhancement in *bashcov* Issue-47 is addressed.

CHAPTER 12

License

All content, logos, source-code under the WorkBench project is release under the Apache 2.0 license, unless explicitly stated otherwise.

A copy of the license can be found in the LICENSE file in the WorkBench repo.

CHAPTER 13

Indices and tables

- genindex
- modindex
- search