Hammurabi Documentation

Release 0.1.0

Gábor Boros

Mar 13, 2020

CONTENTS

CHAPTER

ONE

HAMMURABI

Mass changes made easy.

Hammurabi is an extensible CLI tool responsible for enforcing user-defined rules on a git repository.

1.1 Features

Hammurabi integrates well with both git and Github to make sure that the execution happens on a separate branch and the committed changes are pushed to the target repository. After pushing to the target repository, a pull request will be opened.

Hammurabi supports several operations (Rules) by default. These Rules can do

- file and directory operations like copy, move, create or delete
- manipulation of attributes like ownership or access permissions change
- file and directory manipulations
- piped rule execution (output of a rule is the input of the next rule)
- children rule execution (output of a rule is the input of the upcoming rules)
- working with `plain text` and `ini` files

Upcoming file format support:

- `yaml`
- `toml`
- `json`
- `hocon`

1.2 Installation

Hammurabi can be installed by running `pip install hammurabi` and it requires Python 3.7.0+ to run. This is the preferred method to install Hammurabi, as it will always install the most recent stable release. If you don't have pip installed, this Python installation guide can guide you through the process.

1.3 Configuration

For configuration instructions, please visit the documentation site.

1.4 Command line options

```
hammurabi [OPTIONS] COMMAND [ARGS]...

Hammurabi is an extensible CLI tool responsible for enforcing user-defined
rules on a git repository.

Find more information at: https://hammurabi.readthedocs.io/latest/

Options:
-c, --config PATH               Set the configuration file.  [default:
                                pyproject.toml]
--target PATH                   Set target path. If target is a git repo,
                                commits will be created.
--repository TEXT               Set the remote repository. Required format:
                                owner/repository
--github-token TEXT             Set github access token
--log-level [DEBUG|INFO|WARNING|ERROR]
                                Set logging level.
--help                          Show this message and exit.

Commands:
describe  Show details of a specific resource or group of resources.
enforce   Execute all registered Law.
get       Show a specific resource or group of resources.
version   Print Hammurabi version.
```

1.5 Usage examples

1.5.1 Enforce registered laws

```
$ hammurabi enforce
[INFO]  2020-14-07 16:31 - Checkout branch "hammurabi"
[INFO]  2020-14-07 16:31 - Executing law "L001"
[INFO]  2020-14-07 16:31 - Running task for "configure file exists"
[INFO]  2020-14-07 16:31 - Rule "configure file exists" finished successfully
[INFO]  2020-14-07 16:31 - Running task for "Minimum clang version is set"
[INFO]  2020-14-07 16:31 - Rule "Minimum clang version is set" finished successfully
[INFO]  2020-14-07 16:31 - Running task for "Minimum icc version is set"
```

(continues on next page)

(continued from previous page)

```
[INFO]  2020-14-07 16:31 - Rule "Minimum icc version is set" finished successfully
[INFO]  2020-14-07 16:31 - Running task for "Minimum lessc version is set"
[INFO]  2020-14-07 16:31 - Rule "Minimum lessc version is set" finished successfully
[INFO]  2020-14-07 16:31 - Running task for "Maximum lessc version is set"
[INFO]  2020-14-07 16:31 - Rule "Maximum lessc version is set" finished successfully
[INFO]  2020-14-07 16:31 - Pushing changes
[INFO]  2020-14-07 16:35 - Checking for opened pull request
[INFO]  2020-14-07 16:35 - Opening pull request
```

1.5.2 Listing available laws

```
$ hammurabi get laws
- Gunicorn config set up properly
```

1.5.3 Get info about a law by its name

```
$ hammurabi get law "Gunicorn config set up properly"
Gunicorn config set up properly

Change the gunicorn configuration based on our learnings
described at: https://google.com/?q=gunicorn.

If the gunicorn configuration does not exist, create a
new one configuration file.
```

1.5.4 Get all registered (root) rules

```
$ hammurabi get rules
- Rule 1
- Rule 5
```

1.5.5 Get a rule by its name

```
$ hammurabi get rule "Rule 1"
Rule 1

Ensure that a file exists. If the file does not exists,
this :class:`hammurabi.rules.base.Rule` will create it.

Due to the file is already created by :func:`pre_task_hook`
there is no need to do anything just return the input parameter.
```

1.5.6 Describe a law by its name

```
$ hammurabi describe law "Gunicorn config set up properly"
Gunicorn config set up properly

Change the gunicorn configuration based on our learnings
described at: http://docs.gunicorn.org/en/latest/configure.html.

If the gunicorn configuration does not exist, create a
new one configuration file.

Rules:
--> Rule 1
--> Rule 2
--> Rule 3
--> Rule 4
--> Rule 5
```

1.5.7 Describe a rule by its name

```
$ hammurabi describe rule "Rule 1"
Rule 1

Ensure that a file exists. If the file does not exists,
this :class:`hammurabi.rules.base.Rule` will create it.

Due to the file is already created by :func:`pre_task_hook`
there is no need to do anything just return the input parameter.

Chain:
--> Rule 1
--> Rule 2
--> Rule 3
--> Rule 4
```

1.5.8 Getting the execution order of laws and rules

```
$ hammurabi get order
- Gunicorn config set up properly
--> Rule 1
--> Rule 2
--> Rule 3
--> Rule 4
--> Rule 5
```

1.6 Custom Rules

Although the project aims to support as many general operations as it can, the need for adding custom rules may arise.

To extend Hammurabi with custom rules, you will need to inherit a class from `Rule` and define its abstract methods.

The following example will show you how to create and use a custom rule. For more reference please check how the existing rules are implemented.

```
# custom.py
import shutil
import logging
from hammurabi.mixins import GitMixin
from hammurabi.rules.base import Rule


class CustomOwnerChanged(Rule, GitMixin):
    """
    Change the ownership of a file or directory to <original user>:admin.
    """

    def __init__(self, name: str, path: Optional[Path] = None, **kwargs):
        super().__init__(name, path, **kwargs)

    def post_task_hook(self):
        self.git_add(self.param)

    def task(self, param: Path) -> Path:
        logging.debug('Changing group of "%s" to admin', str(self.param))
        shutil.chown(param, group="admin")
        return param
```

1.7 Contributing

Hurray, You reached this section, which means you are ready to contribute.

Please read our contibuting guideline. This guideline will walk you through how can you successfully contribute to Hammurabi.

1.7.1 Installation

For development you will need poetry. After poetry installed, simply run *poetry install*. This command will both create the virtualenv and install development dependencies for you.

1.7.2 Useful make Commands

Command	Description
help	Print available make commands
clean	Remove all artifacts
clean-build	Remove build artifacts
clean-mypy	Remove mypy artifacts
clean-pyc	Remove Python artifacts
clean-test	Remove test artifacts
doc	Genereate Sphinx documentation
format	Run several formatters
lint	Run several linters after format
test	Run all tests with coverage
test-unit	Run unit tests with coverage
test-integration	Run integration tests with coverage

1.8 Why Hammurabi?

Hammurabi was the sixth king in the Babylonian dynasty, which ruled in central Mesopotamia from c. 1894 to 1595 B.C.

The Code of Hammurabi was one of the earliest and most complete written legal codes and was proclaimed by the Babylonian king Hammurabi, who reigned from 1792 to 1750 B.C. Hammurabi expanded the city-state of Babylon along the Euphrates River to unite all of southern Mesopotamia. The Hammurabi code of laws, a collection of 282 rules, established standards for commercial interactions and set fines and punishments to meet the requirements of justice. Hammurabi's Code was carved onto a massive, finger-shaped black stone stele (pillar) that was looted by invaders and finally rediscovered in 1901.

CHAPTER

TWO

INSTALLATION

2.1 Stable release

To install Hammurabi, run this command in your terminal:

```
$ pip install hammurabi
```

This is the preferred method to install Hammurabi, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for Hammurabi can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/gabor-boros/hammurabi
```

Or download the tarball:

```
$ curl  -OL https://github.com/gabor-boros/hammurabi/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER

THREE

CONFIGURATION

3.1 Overview

3.2 Hammurabi configuration

You can set the following options in your `pyproject.toml` config file's *[hammurabi]* section.

Config option	Description	Default value
config	location of pyproject.toml	pyproject.toml
pillar	name of the pillar variable	pillar
log_level	logging level of the program	INFO
target	location of the target directory	. (current dir)
repository	github repository (owner/repo)	None
git_branch_name	working branch name	hammurabi
dry_run	enforce without any modification	False
rule_can_abort	if a rule fails it aborts the whole execution	False

3.2.1 Examples

Example content of the `pyproject.toml` file.

```
[hammurabi]
repository = "gabor-boros/hammurabi"
git_branch_name = "custom-branch"
log_level = "WARNING"
rule_can_abort = true
```

3.3 Pillar configuration

The pillar needs no configuration. All the thing the developer must do is creating a *hammurabi.pillar.Pillar* object and registering the laws to it.

3.3.1 Using custom rules

Custom rules are not different from built-in one. In case of a custom rule, just import and use it.

3.3.2 Examples

```
>>> from hammurabi import Law, Pillar
>>> from mycompany.rules import MyCustomRule
>>>
>>> meaning_of_life = Law(
>>>     name="...",
>>>     description="...",
>>>     rules=[MyCustomRule]
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(meaning_of_life)
```

CHAPTER

FOUR

HAMMURABI

4.1 hammurabi package

4.1.1 Subpackages

hammurabi.rules package

Submodules

hammurabi.rules.attributes module

Attributes module contains file and directory attribute manipulation rules which can be handy after creating new files or directories or even when adding execute permissions for a script in the project.

class hammurabi.rules.attributes.**ModeChanged**(*name: str*, *path: Optional[pathlib.Path] = None*, *new_value: Optional[int] = None*, ***kwargs*)

Bases: *hammurabi.rules.attributes.SingleAttributeRule*

Change the mode of a file or directory.

Supported modes:

Config option	Description
stat.S_ISUID	Set user ID on execution.
stat.S_ISGID	Set group ID on execution.
stat.S_ENFMT	Record locking enforced.
stat.S_ISVTX	Save text image after execution.
stat.S_IREAD	Read by owner.
stat.S_IWRITE	Write by owner.
stat.S_IEXEC	Execute by owner.
stat.S_IRWXU	Read, write, and execute by owner.
stat.S_IRUSR	Read by owner.
stat.S_IWUSR	Write by owner.
stat.S_IXUSR	Execute by owner.
stat.S_IRWXG	Read, write, and execute by group.
stat.S_IRGRP	Read by group.
stat.S_IWGRP	Write by group.
stat.S_IXGRP	Execute by group.
stat.S_IRWXO	Read, write, and execute by others.
stat.S_IROTH	Read by others.
stat.S_IWOTH	Write by others.
stat.S_IXOTH	Execute by others.

Example usage:

```
>>> import stat
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, ModeChanged
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         ModeChanged(
>>>             name="Update script must be executable",
>>>             path=Path("./scripts/update.sh"),
>>>             new_value=stat.S_IXGRP | stat.S_IXGRP | stat.S_IXOTH
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

task() → pathlib.Path

Change the mode of the given file or directory.

Returns Return the input path as an output

Return type Path

class hammurabi.rules.attributes.**OwnerChanged**(*name: str*, *path: Optional[pathlib.Path] = None*, *new_value: Optional[str] = None*, ***kwargs*)

Bases: *hammurabi.rules.attributes.SingleAttributeRule*

Change the ownership of a file or directory.

The new ownership of a file or directory can be set in three ways. To set only the user use

`new_value="username"`. To set only the group use `new_value=":group_name"` (please note the colon `:`). It is also possible to set both username and group at the same time by using `new_value="username:group_name"`.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, OwnerChanged
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         OwnerChanged(
>>>             name="Change ownership of nginx config",
>>>             path=Path("./nginx.conf"),
>>>             new_value="www:web_admin"
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

task() → pathlib.Path

Change the ownership of the given file or directory. None of the new username or group name can contain colons, otherwise only the first two colon separated values will be used as username and group name.

Returns Return the input path as an output

Return type Path

class hammurabi.rules.attributes.**SingleAttributeRule**(*name: str*, *path: Optional[pathlib.Path] = None*, *new_value: Optional[str] = None*, ***kwargs*)

Bases: *hammurabi.rules.common.SinglePathRule*

Extend *hammurabi.rules.base.Rule* to handle attributes of a single file or directory.

post_task_hook()

Run code after the *hammurabi.rules.base.Rule.task()* has been performed. To access the parameter passed to the rule, always use `self.param` for *hammurabi.rules.base.Rule.post_task_hook()*.

Note: This method can be used for execution of git commands like git add, or double checking a modification made.

Warning: This method is not called in dry run mode.

abstract task() → Any

Abstract method representing how a *hammurabi.rules.base.Rule.task()* must be parameterized. Any difference in the parameters will result in pylint/mypy errors.

For more details please check *hammurabi.rules.base.Rule.task()*.

hammurabi.rules.base module

This module contains the definition of Rule which describes what to do with the received parameter and does the necessary changes.

The Rule is an abstract class which describes all the required methods and parameters, but it can be extended and customized easily by inheriting from it. A good example for this kind of customization is *hammurabi.rules.text.LineExists* which adds more parameters to `hammurabi.rules.files.SingleFileRule` which inherits from *hammurabi.rules.base.Rule*.

class hammurabi.rules.base.**Rule**(*name: str*, *param: Any*, *preconditions: Iterable[Rule] = ()*, *pipe: Optional[Rule] = None*, *children: Iterable[Rule] = ()*)

Bases: `abc.ABC`

Abstract class which describes the bare minimum and helper functions for Rules. A rule defines what and how should be executed. Since a rule can have piped and children rules, the "parent" rule is responsible for those executions. This kind of abstraction allows to run both piped and children rules sequentially in a given order.

Example usage:

```
>>> from typing import Optional
>>> from pathlib import Path
>>> from hammurabi import Rule
>>> from hammurabi.mixins import GitMixin
>>>
>>> class SingleFileRule(Rule, GitMixin):
>>>     def __init__(self, name: str, path: Optional[Path] = None, **kwargs):
>>>         super().__init__(name, path, **kwargs)
>>>
>>>     def post_task_hook(self):
>>>         self.git_add(self.param)
>>>
>>>     @abstractmethod
>>>     def task(self, param: Path) -> Path:
>>>         pass
```

Parameters

- **name** (*str*) – Name of the rule which will be used for printing
- **preconditions** (*Iterable["Rule"]*) – "Boolean Rules" which returns a truthy or falsy value
- **pipe** (*Optional["Rule"]*) – Pipe will be called when the rule is executed successfully
- **children** (*Iterable["Rule"]*) – Children will be executed after the piped rule if there is any

> **Warning:** Preconditions can be used in several ways. The most common way is to run "Boolean Rules" which takes a parameter and returns a truthy or falsy value. In case of a falsy return, the precondition will fail and the rule will not be executed.
>
> If any modification is done by any of the rules which are used as a precondition, those changes will be committed.

property can_proceed

Evaluate if a rule can continue its execution. In case the execution is called with `dry_run` config option

set to true, this method will always return `False` to make sure not performing any changes. If preconditions are set, those will be evaluated by this method.

Returns Return with the result of evaluation

Return type bool

> **Warning:** *hammurabi.rules.base.Rule.can_proceed()* checks the result of `self.preconditions`, which means the preconditions are executed. Make sure that you are not doing any modifications within rules used as preconditions, otherwise take extra attention for those rules.

property description

Return the description of the *hammurabi.rules.base.Rule.task()* based on its docstring.

Returns Stripped description of *hammurabi.rules.base.Rule.task()*

Return type str

Note: As of this method returns the docstring of *hammurabi.rules.base.Rule.task()* method, it worth to take care of its description when initialized.

---

property documentation

Return the documentation of the rule based on its name, docstring and the description of its task.

Returns Concatenation of the rule's name, docstring, and task description

Return type str

Note: As of this method returns the name and docstring of the rule it worth to take care of its name and description when initialized.

---

execute(*param: Optional[Any] = None*)

Execute the rule's task, its piped and children rules as well.

The execution order of task, piped rule and children rules described in but not by *hammurabi.rules.base.Rule.get_rule_chain()*.

Parameters **param** (*Optional[Any]*) – Input parameter of the rule given by the user

Raise `AssertionError`

Returns None

Note: The input parameter can be optional because of the piped and children rules which are receiving the output of its parent. In this case the user is not able to set the param manually, since it is calculated.

> **Warning:** If `self.can_proceed` returns `False` the whole execution will be stopped immediately and `AssertionError` will be raised.

get_execution_order() → List[Rule]

Same as *hammurabi.rules.base.Rule.get_rule_chain()* but for the root rule.

get_rule_chain(*rule: Rule*) → List[Rule]
Get the execution chain of the given rule. The execution order is the following:

- task (current rule's *hammurabi.rules.base.Rule.task()*)
- Piped rule
- Children rules (in the order provided by the iterator used)

Parameters **rule** (*hammurabi.rules.base.Rule*) – The rule which execution chain should be returned

Returns Returns the list of rules in the order above

Return type List[*Rule*]

post_task_hook()
Run code after the *hammurabi.rules.base.Rule.task()* has been performed. To access the parameter passed to the rule, always use `self.param` for *hammurabi.rules.base.Rule.post_task_hook()*.

Note: This method can be used for execution of git commands like git add, or double checking a modification made.

Warning: This method is not called in dry run mode.

pre_task_hook()
Run code before performing the *hammurabi.rules.base.Rule.task()*. To access the parameter passed to the rule, always use `self.param` for *hammurabi.rules.base.Rule.pre_task_hook()*.

Warning: This method is not called in dry run mode.

abstract task() → Any
Abstract method representing how a *hammurabi.rules.base.Rule.task()* must be parameterized. Any difference in the parameters will result in pylint/mypy errors.

To be able to use the power of `pipe` and `children`, return something which can be generally used for other rules as in input.

Returns Returns an output which can be used as an input for other rules

Return type Any (usually same as *self.param*'s type)

Note: Although it is a good practice to return the same type for the output that the input has, but this is not the case for "Boolean Rules". "Boolean Rules" should return True (or truthy) or False (or falsy) values.

Example usage:

```
>>> import logging
>>> from pathlib import Path
>>> from hammurabi.rules.files import SingleFileRule
```

(continues on next page)

(continued from previous page)

```
>>>
>>> class FileExists(SingleFileRule):
>>>     def task(self) -> Path:
>>>         logging.debug('Creating file "%s" if not exists', str(self.param))
>>>         self.param.touch()
>>>         return self.param
```

static validate(*val: Any*, *cast_to: Optional[Any] = None*, *required=False*) → Any

Validate and/or cast the given value to another type. In case the existence of the value is required or casting failed an exception will be raised corresponding to the failure.

Parameters

- **val** (*Any*) – Value to validate
- **cast_to** (*Any*) – Type in which the value should be returned
- **required** (*bool*) – Check that the value is not falsy

Raise `ValueError` if the given value is required but falsy

Returns Returns the value in its original or casted type

Return type Any

Example usage:

```
>>> from typing import Optional
>>> from pathlib import Path
>>> from hammurabi import Rule
>>>
>>> class MyAwesomeRule(Rule):
>>>     def __init__(self, name: str, param: Optional[Path] = None):
>>>         self.param = self.validate(param, required=True)
>>>
>>>     # Other method definitions ...
>>>
```

hammurabi.rules.common module

class hammurabi.rules.common.**MultiplePathRule**(*name: str*, *paths: Optional[Iterable[pathlib.Path]] = ()*, ***kwargs*)

Bases: *hammurabi.rules.base.Rule*, *hammurabi.mixins.GitMixin*

Abstract class which extends *hammurabi.rules.base.Rule* to handle operations on multiple files.

post_task_hook()

Run code after the *hammurabi.rules.base.Rule.task()* has been performed. To access the parameter passed to the rule, always use `self.param` for *hammurabi.rules.base.Rule.post_task_hook()*.

Note: This method can be used for execution of git commands like git add, or double checking a modification made.

> **Warning:** This method is not called in dry run mode.

abstract task() → Any
Abstract method representing how a *hammurabi.rules.base.Rule.task()* must be parameterized. Any difference in the parameters will result in pylint/mypy errors.

For more details please check *hammurabi.rules.base.Rule.task()*.

class hammurabi.rules.common.**SinglePathRule**(*name: str*, *path: Optional[pathlib.Path] = None*, ***kwargs*)
Bases: *hammurabi.rules.base.Rule*, *hammurabi.mixins.GitMixin*

Abstract class which extends *hammurabi.rules.base.Rule* to handle operations on a single directory.

post_task_hook()
Run code after the *hammurabi.rules.base.Rule.task()* has been performed. To access the parameter passed to the rule, always use `self.param` for *hammurabi.rules.base.Rule.post_task_hook()*.

> **Note:** This method can be used for execution of git commands like git add, or double checking a modification made.

> **Warning:** This method is not called in dry run mode.

abstract task() → Any
Abstract method representing how a *hammurabi.rules.base.Rule.task()* must be parameterized. Any difference in the parameters will result in pylint/mypy errors.

For more details please check *hammurabi.rules.base.Rule.task()*.

hammurabi.rules.directories module

Directories module contains directory specific manipulation rules. Please note that those rules which can be used for files and directories are located in other modules like *hammurabi.rules.operations* or *hammurabi.rules.attributes*.

class hammurabi.rules.directories.**DirectoryEmptied**(*name: str*, *path: Optional[pathlib.Path] = None*, ***kwargs*)
Bases: *hammurabi.rules.common.SinglePathRule*

Ensure that the given directory's content is removed. Please note the difference between emptying a directory and recreating it. The latter results in lost ACLs, permissions and modes.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, DirectoryEmptied
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
```

(continues on next page)

(continued from previous page)

```
>>>         DirectoryEmptied(
>>>             name="Empty results directory",
>>>             path=Path("./test-results")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

task() → pathlib.Path

Iterate through the entries of the given directory and remove them. If an entry is a file simply remove it, otherwise remove the whole subdirectory and its content.

Returns Return the input path as an output

Return type Path

class hammurabi.rules.directories.**DirectoryExists**(*name: str*, *path: Optional[pathlib.Path] = None*, ***kwargs*)

Bases: *hammurabi.rules.common.SinglePathRule*

Ensure that a directory exists. If the directory does not exists, make sure the directory is created.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, DirectoryExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         DirectoryExists(
>>>             name="Create secrets directory",
>>>             path=Path("./secrets")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

task() → pathlib.Path

Create the given directory if not exists.

Returns Return the input path as an output

Return type Path

class hammurabi.rules.directories.**DirectoryNotExists**(*name: str*, *path: Optional[pathlib.Path] = None*, ***kwargs*)

Bases: *hammurabi.rules.common.SinglePathRule*

Ensure that the given directory does not exists.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, DirectoryNotExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         DirectoryNotExists(
>>>             name="Remove unnecessary directory",
>>>             path=Path("./temp")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

post_task_hook()
Remove the given directory from git index.

task() → pathlib.Path
Remove the given directory.

Returns Return the input path as an output

Return type Path

hammurabi.rules.files module

Files module contains file specific manipulation rules. Please note that those rules which can be used for files and directories are located in other modules like *hammurabi.rules.operations* or *hammurabi.rules.attributes*.

class hammurabi.rules.files.**FileEmptied**(*name: str*, *path: Optional[pathlib.Path] = None*, ***kwargs*)
Bases: *hammurabi.rules.common.SinglePathRule*

Remove the content of the given file, but keep the file. Please note the difference between emptying a file and recreating it. The latter results in lost ACLs, permissions and modes.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, FileEmptied
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         FileEmptied(
>>>             name="Empty the check log file",
>>>             path=Path("/var/log/service/check.log")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

task() → pathlib.Path
Remove the content of the given file. If the file does not exists this rule will create the file without content.

Returns Return the emptied/created file's path

Return type Path

class hammurabi.rules.files.**FileExists**(*name: str*, *path: Optional[pathlib.Path] = None*, ***kwargs*)
Bases: *hammurabi.rules.common.SinglePathRule*

Ensure that a file exists. If the file does not exists, make sure the file is created.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, FileExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         FileExists(
>>>             name="Create service descriptor",
>>>             path=Path("./service.yaml")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

task() → pathlib.Path
If the target file not exists, create the file to make sure we can manipulate it.

Returns The created/existing file's path

Return type Path

class hammurabi.rules.files.**FileNotExists**(*name: str*, *path: Optional[pathlib.Path] = None*, ***kwargs*)
Bases: *hammurabi.rules.common.SinglePathRule*

Ensure that the given file does not exists. If the file exists remove it, otherwise do nothing and return the original path.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, FileNotExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         FileNotExists(
>>>             name="Remove unused file",
>>>             path=Path("./debug.yaml")
>>>         ),
>>>     )
>>> )
```

(continues on next page)

(continued from previous page)

```
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

post_task_hook()
Remove the given file from git index.

task() → pathlib.Path
Remove the given file if exists, otherwise do nothing and return the original path.

Returns Return the removed file's path

Return type Path

class hammurabi.rules.files.**FilesExist**(*name: str*, *paths: Optional[Iterable[pathlib.Path]] = ()*, ***kwargs*)
Bases: *hammurabi.rules.common.MultiplePathRule*

Ensure that all files exists. If the files does not exists, make sure the files are created.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, FilesExist
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         FilesExist(
>>>             name="Create test files",
>>>             paths=[
>>>                 Path("./file_1"),
>>>                 Path("./file_2"),
>>>                 Path("./file_3"),
>>>             ]
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

task() → Iterable[pathlib.Path]
If the target files not exist, create the files to make sure we can manipulate them.

Returns The created/existing files' path

Return type Iterable[Path]

class hammurabi.rules.files.**FilesNotExist**(*name: str*, *paths: Optional[Iterable[pathlib.Path]] = ()*, ***kwargs*)
Bases: *hammurabi.rules.common.MultiplePathRule*

Ensure that the given files does not exist. If the files exist remove them, otherwise do nothing and return the original paths.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, FilesNotExist
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         FilesNotExist(
>>>             name="Remove several files",
>>>             paths=[
>>>                 Path("./file_1"),
>>>                 Path("./file_2"),
>>>                 Path("./file_3"),
>>>             ]
>>>         ),
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

post_task_hook()
Remove the given files from git index.

task() → Iterable[pathlib.Path]
Remove all existing files.

Returns Return the removed files' paths

Return type Iterable[Path]

hammurabi.rules.ini module

Ini module is an extension for text rules tailor made for .ini/.cfg files. The main difference lies in the way it works. First, the .ini/.cfg file is parsed, then the modifications are made on the already parsed file.

class hammurabi.rules.ini.**OptionRenamed**(*name: str*, *path: Optional[pathlib.Path] = None*, *option: Optional[str] = None*, *new_name: Optional[str] = None*, ***kwargs*)
Bases: *hammurabi.rules.ini.SingleConfigFileRule*

Ensure that an option of a section is renamed.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, OptionRenamed
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         OptionRenamed(
>>>             name="Rename an option",
>>>             path=Path("./config.ini"),
>>>             section="my_section",
>>>             option="typo",
```

(continues on next page)

(continued from previous page)

```
>>>             new_name="correct",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

task() → pathlib.Path

Rename an option of a section. In case a section can not be found, a `LookupError` exception will be raised to stop the execution. The execution must be stopped at this point, because if dependant rules will fail otherwise.

Raises `LookupError` raised if no section can be renamed

Returns Return the input path as an output

Return type Path

class hammurabi.rules.ini.**OptionsExist**(*name: str*, *path: Optional[pathlib.Path] = None*, *options: Iterable[Tuple[str, Any]] = None*, *force_value: bool = False*, ***kwargs*)

Bases: *hammurabi.rules.ini.SingleConfigFileRule*

Ensure that the given config option exists. If needed, the rule will create a config option with the given value. In case the `force_value` parameter is set to True, the original values will be replaced by the give ones.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, OptionsExist
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         OptionsExist(
>>>             name="Ensure options are changed",
>>>             path=Path("./config.ini"),
>>>             section="fetching",
>>>             options=(
>>>                 ("interval", "2s"),
>>>                 ("abort_on_error", True),
>>>             ),
>>>             force_value=True,
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

Warning: When using the `force_value` parameter, please note that all the existing option values will be replaced by those set in `options` parameter.

task() → pathlib.Path

Remove one or more option from a section. In case a section can not be found, a `LookupError` exception

will be raised to stop the execution. The execution must be stopped at this point, because if dependant rules will fail otherwise.

Raises `LookupError` raised if no section can be renamed

Returns Return the input path as an output

Return type Path

class hammurabi.rules.ini.**OptionsNotExist**(*name: str*, *path: Optional[pathlib.Path] = None*, *options: Iterable[str] = ()*, ***kwargs*)

Bases: *hammurabi.rules.ini.SingleConfigFileRule*

Remove one or more option from a section.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, OptionsNotExist
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         OptionsNotExist(
>>>             name="Ensure options are removed",
>>>             path=Path("./config.ini"),
>>>             section="invalid",
>>>             options=(
>>>                 "remove",
>>>                 "me",
>>>                 "please",
>>>             )
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

task() → pathlib.Path

Remove one or more option from a section. In case a section can not be found, a `LookupError` exception will be raised to stop the execution. The execution must be stopped at this point, because if dependant rules will fail otherwise.

Raises `LookupError` raised if no section can be renamed

Returns Return the input path as an output

Return type Path

class hammurabi.rules.ini.**SectionExists**(*name: str*, *path: Optional[pathlib.Path] = None*, *target: Optional[str] = None*, *options: Iterable[Tuple[str, Any]] = ()*, *add_after: bool = True*, ***kwargs*)

Bases: *hammurabi.rules.ini.SingleConfigFileRule*

Ensure that the given config section exists. If needed, the rule will create a config section with the given name, and optionally the specified options. In case options are set, the config options will be assigned to that config sections.

Similarly to *hammurabi.rules.text.LineExists*, this rule is able to add a section before or after a target section. The limitation compared to `LineExists` is that the `SectionExists` rule is only able to add the new entry exactly before or after its target.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, SectionExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         SectionExists(
>>>             name="Ensure section exists",
>>>             path=Path("./config.ini"),
>>>             section="polling",
>>>             target="add_after_me",
>>>             options=(
>>>                 ("interval", "2s"),
>>>                 ("abort_on_error", True),
>>>             ),
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

Warning: When `options` parameter is set, make sure you are using an iterable tuple. The option keys must be strings, but there is no limitation for the value. It can be set to anything what the parser can handle. For more information on the parser, please visit the documentation of configupdater.

task() → pathlib.Path

Ensure that the given config section exists. If needed, create a config section with the given name, and optionally the specified options.

In case options are set, the config options will be assigned to that config sections. A `LookupError` exception will be raised if the target section can not be found.

Raises `LookupError` raised if no target can be found

Returns Return the input path as an output

Return type Path

class hammurabi.rules.ini.**SectionNotExists**(*name: str*, *path: Optional[pathlib.Path] = None*, *section: Optional[str] = None*, ***kwargs*)

Bases: *hammurabi.rules.ini.SingleConfigFileRule*

Make sure that the given file not contains the specified line. When a section removed, all the options belonging to it will be removed too.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, SectionNotExists
```

(continues on next page)

(continued from previous page)

```
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         SectionNotExists(
>>>             name="Ensure section removed",
>>>             path=Path("./config.ini"),
>>>             section="invalid",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

task() → pathlib.Path

Remove the given section including its options from the config file.

Returns Return the input path as an output

Return type Path

class hammurabi.rules.ini.**SectionRenamed**(*name: str*, *path: Optional[pathlib.Path] = None*, *new_name: Optional[str] = None*, ***kwargs*)

Bases: *hammurabi.rules.ini.SingleConfigFileRule*

Ensure that a section is renamed. None of its options will be changed.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, SectionRenamed
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         SectionRenamed(
>>>             name="Ensure section renamed",
>>>             path=Path("./config.ini"),
>>>             section="polling",
>>>             new_name="fetching",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

task() → pathlib.Path

Rename the given section to a new name. None of its options will be changed. In case a section can not be found, a `LookupError` exception will be raised to stop the execution. The execution must be stopped at this point, because if other rules depending on the rename will fail otherwise.

Raises `LookupError` raised if no section can be renamed

Returns Return the input path as an output

Return type Path

class hammurabi.rules.ini.**SingleConfigFileRule**(*name: str*, *path: Optional[pathlib.Path] = None*, *section: Optional[str] = None*, ***kwargs*)

Bases: *hammurabi.rules.common.SinglePathRule*

Extend *hammurabi.rules.base.Rule* to handle parsed content manipulations on a single file.

pre_task_hook()

Parse the configuration file for later use.

abstract task() → Any

Abstract method representing how a *hammurabi.rules.base.Rule.task()* must be parameterized. Any difference in the parameters will result in pylint/mypy errors.

For more details please check *hammurabi.rules.base.Rule.task()*.

hammurabi.rules.operations module

Operations module contains common file/directory operation which can be handy when need to move, rename or copy files.

class hammurabi.rules.operations.**Copied**(*name: str*, *path: Optional[pathlib.Path] = None*, *destination: Optional[pathlib.Path] = None*, ***kwargs*)

Bases: *hammurabi.rules.common.SinglePathRule*

Ensure that the given file or directory is copied to the new path.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, Copied
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         Copied(
>>>             name="Create backup file",
>>>             path=Path("./service.yaml"),
>>>             destination=Path("./service.bkp.yaml")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

post_task_hook()

Add the destination and not the original path.

task() → pathlib.Path

Copy the given file or directory to a new place.

Returns Returns the path of the copied file/directory

Return type Path

class hammurabi.rules.operations.**Moved**(*name: str*, *path: Optional[pathlib.Path] = None*, *destination: Optional[pathlib.Path] = None*, ***kwargs*)

Bases: *hammurabi.rules.common.SinglePathRule*

Move a file or directory from "A" to "B".

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, Moved
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         Moved(
>>>             name="Move pyproject.toml to its place",
>>>             path=Path("/tmp/generated/pyproject.toml.template"),
>>>             destination=Path("./pyproject.toml"),  # Notice the rename!
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

post_task_hook()

Add both the new and old git objects.

task() → pathlib.Path

Move the given path to the destination. In case the file got a new name when destination is provided, the file/directory will be moved to its new place with its new name.

Returns Returns the new destination of the file/directory

Return type Path

class hammurabi.rules.operations.**Renamed**(*name: str*, *path: Optional[pathlib.Path] = None*, *new_name: Optional[str] = None*, ***kwargs*)

Bases: *hammurabi.rules.operations.Moved*

This rule is a shortcut for *hammurabi.rules.operations.Moved*. Instead of destination path a new name is required.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, Renamed
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         Renamed(
>>>             name="Rename pyproject.toml.bkp",
>>>             path=Path("/tmp/generated/pyproject.toml.bkp"),
>>>             new_name="pyproject.toml",
>>>         ),
>>>     )
```

(continues on next page)

(continued from previous page)

```
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

hammurabi.rules.text module

Text module contains simple but powerful general file content manipulations. Combined with other simple rules like *hammurabi.rules.files.FileExists* or *hammurabi.rules.attributes.ModeChanged* almost anything can be achieved. Although any file's content can be changed using these rules, for common file formats like `ini`, `yaml` or `json` dedicated rules are created.

class hammurabi.rules.text.**LineExists**(*name: str*, *path: Optional[pathlib.Path] = None*, *text: Optional[str] = None*, *criteria: Optional[str] = None*, *target: Optional[str] = None*, *position: int = 1*, *respect_indentation: bool = True*, ***kwargs*)

Bases: *hammurabi.rules.common.SinglePathRule*

Make sure that the given file contains the required line. This rule is capable for inserting the expected text before or after the unique target text respecting the indentation of its context.

The default behaviour is to insert the required text exactly after the target line, and respect its indentation. Please note that `text`, `criteria` and `target` parameters are required.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, LineExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         LineExists(
>>>             name="Extend gunicorn config",
>>>             path=Path("./gunicorn.conf.py"),
>>>             text="keepalive = 65",
>>>             criteria=r"^keepalive.*",
>>>             target=r"^bind.*",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

Note: The indentation of the target text will be extracted by a simple regular expression. If a more complex regexp is required, please inherit from this class.

---

task() → pathlib.Path

Make sure that the given file contains the required line. This rule is capable for inserting the expected rule before or after the unique target text respecting the indentation of its context.

Raises `LookupError`

Returns Returns the path of the modified file

Return type Path

class hammurabi.rules.text.**LineNotExists**(*name: str*, *path: Optional[pathlib.Path] = None*, *text: Optional[str] = None*, ***kwargs*)

Bases: *hammurabi.rules.common.SinglePathRule*

Make sure that the given file not contains the specified line.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, LineNotExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         LineNotExists(
>>>             name="Remove keepalive",
>>>             path=Path("./gunicorn.conf.py"),
>>>             text="keepalive = 65",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

task() → pathlib.Path

Make sure that the given file not contains the specified line based on the given criteria.

Returns Returns the path of the modified file

Return type Path

class hammurabi.rules.text.**LineReplaced**(*name: str*, *path: Optional[pathlib.Path] = None*, *text: Optional[str] = None*, *target: Optional[str] = None*, *respect_indentation: bool = True*, ***kwargs*)

Bases: *hammurabi.rules.common.SinglePathRule*

Make sure that the given text is replaced in the given file.

The default behaviour is to replace the required text with the exact same indentation that the target line has. This behaviour can be turned off by setting the `respect_indentation` parameter to False. Please note that `text` and `target` parameters are required.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, LineReplaced
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         LineReplaced(
>>>             name="Replace typo using regex",
>>>             path=Path("./gunicorn.conf.py"),
```

(continues on next page)

(continued from previous page)

```
>>>             text="keepalive = 65",
>>>             target=r"^kepalive.*",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

Note: The indentation of the target text will be extracted by a simple regular expression. If a more complex regexp is required, please inherit from this class.

Warning: This rule will replace all the matching lines in the given file. Make sure the given target regular expression is tested before the rule used against production code.

task() → pathlib.Path

Make sure that the given text is replaced in the given file.

Raises LookupError

Returns Returns the path of the modified file

Return type Path

Module contents

4.1.2 Submodules

4.1.3 hammurabi.config module

class hammurabi.config.**Config**

Bases: object

Simple configuration object which used across Hammurabi. The *Config* loads the given pyproject.toml according to PEP-518.

load(*file: Union[str, pathlib.Path]*)

Load and parse the given pyproject.toml file.

property repo

Get the target directory.

4.1.4 hammurabi.exceptions module

exception hammurabi.exceptions.**AbortLawError**
Bases: Exception

Custom exception to make sure that own exception types are caught by the Law's execution.

4.1.5 hammurabi.helpers module

hammurabi.helpers.**full_strip**(*value: str*) → str
Strip every line.

4.1.6 hammurabi.law module

This module contains the definition of Law which is responsible for the execution of its registered Rules. Every Law can have multiple rules to execute.

In case a rule raises an exception the execution may abort and none of the remaining rules will be executed neither pipes or children. An abort can cause an inconsistent state or a dirty git branch. If `rule_can_abort` config is set to True, the whole execution of the :class:`hammurabi.pillar.Pillar` will be aborted and the original exception will be re-raised.

class hammurabi.law.**Law**(*name: str, description: str, rules: Iterable[hammurabi.rules.base.Rule]*)
Bases: *hammurabi.mixins.GitMixin*

A Law is a collection of Rules which is responsible for the rule execution and git committing.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, FileExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         FileExists(
>>>             name="Create pyproject.toml",
>>>             path=Path("./pyproject.toml")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

commit()
Commit the changes made by registered rules and add a meaningful commit message.

Example commit message:

```
Migrate to next generation project template
* Create pyproject.toml
* Add meta info from setup.py to pyproject.toml
* Add existing dependencies
* Remove requirements.txt
* Remove setup.py
```

property documentation
Get the name and description of the Law object.

Returns Return the name and description of the law as its documentation

Return type str

enforce()
Execute all registered rule. If `rule_can_abort` config option is set to `True`, all the rules will be aborted and an exception will be raised.

When the whole execution chain is finished, the changes will be committed except the failed ones.

Raises `AbortLawError`

get_execution_order() → List[hammurabi.rules.base.Rule]
Get the execution order of the registered rules. The order will contain the pipes and children as well.

This helper function is useful in debugging and information gathering.

Returns Return the execution order of the rules

Return type List[*Rule*]

4.1.7 hammurabi.main module

4.1.8 hammurabi.mixins module

Mixins module contains helpers for both laws and rules. Usually this file will contain Git commands related helpers. Also, this module contains the extensions for several online git based VCS.

class hammurabi.mixins.**GitHubMixin**
Bases: *hammurabi.mixins.GitMixin*

Extending *hammurabi.mixins.GitMixin* to be able to open pull requests on GitHub after changes are pushed to remote.

create_pull_request()
Create a PR on GitHub after the changes are pushed to remote. The pull request details (repository, branch) are set by the project configuration. The mapping of the details and configs:

Detail	Configuration
repo	repository (owner/repository format)
base	git_base_name
branch	git_branch_name

static generate_pull_request_body(*pillar*) → str
Generate the body of the pull request based on the registered laws and rules. The pull request body is markdown formatted.

Parameters **pillar** (*hammurabi.pillar.Pillar*) – Pillar configuration

Returns Returns the generated pull request description

Return type str

class hammurabi.mixins.**GitMixin**
Bases: `object`

Simple mixin which contains all the common git commands which are needed to push a change to an online VCS like GitHub or GitLab. This mixin could be used by `hammurabi.law.Law`s, :class:`hammurabi.rules.base` or any rules which can make modifications during its execution.

static checkout_branch()
Perform a simple git checkout, to not pollute the default branch and use that branch for the pull request later. The branch name can be changed in the config by setting the `git_branch_name` config option.

The following command is executed:

```
git checkout -b <branch name>
```

static git_add(*param: pathlib.Path*)
Add file contents to the index.

Parameters **param** (*Path*) – Path to add to the index

The following command is executed:

```
git add <path>
```

git_commit(*message: str*)
Commit the changes on the checked out branch.

Parameters **message** (*str*) – Git commit message

The following command is executed:

```
git commit -m "<commit message>"
```

static git_remove(*param: pathlib.Path*)
Remove files from the working tree and from the index.

Parameters **param** (*Path*) – Path to remove from the working tree and the index

The following command is executed:

```
git rm <path>
```

property has_changes
Check if the rule made any changes. The check will return True if the git branch is dirty after the rule execution or the Rule set the `made_changes` attribute directly. In usual cases, the `made_changes` attribute will not be set directly by any rule.

Returns True if the git branch is dirty or `made_changes` attribute is True

Return type bool

static push_changes()
Push the changes with the given branch set by `git_branch_name` config option to the remote origin.

The following command is executed:

```
git push origin <branch name>
```

4.1.9 hammurabi.pillar module

Pillar module is responsible for handling the whole execution chain including managing a lock file, executing the registered laws, pushing the changes to the VCS and creating a pull request. All the laws registered to the pillar will be executed not in the order of the registration.

class hammurabi.pillar.**Pillar**

Bases: *hammurabi.mixins.GitHubMixin*

Pillar is responsible for the execution of the chain of laws and rules. During the execution process a lock file will be created at the beginning of the process and at the end, the lock file will be released.

All the registered laws and rules can be retrieved using the `laws` and `rules` properties, or if necessary single laws and rules can be accessed using the resource's name as a parameter for `get_law` or `get_rule` methods.

create_lock_file()

Create a lock file. If the lock file presents, the execution for the same target will be prevented.

enforce()

Run all the registered laws and rules one by one. This method is responsible for creating and releasing the lock file, executing the registered laws, push changes to the git origin and open the pull request.

This method glues together the lower level components and makes sure that the execution of laws and rules can not be called more than once at the same time for a target.

get_law(*name: str*) → hammurabi.law.Law

Get a law by its name. In case of no Laws are registered or the law can not be found by its name, a `StopIteration` exception will be raised.

Parameters **name** (*str*) – Name of the law which will be used for the lookup

Raises `StopIteration` exception if Law not found

Returns Return the searched law

Return type *hammurabi.law.Law*

get_rule(*name: str*) → hammurabi.rules.base.Rule

Get a registered rule (and its pipe/children) by the rule's name.

This helper function is useful in debugging and information gathering.

Parameters **name** (*str*) – Name of the rule which will be used for the lookup

Raises `StopIteration` exception if Rule not found

Returns Return the rule in case of a match for the name

Return type *Rule*

property laws

Return the registered laws not in order of the registration.

register(*law: hammurabi.law.Law*)

Register the given Law to the Pillar. The order of the registration does not matter. The laws should never depend on each other.

Parameters **law** (`hammurabi.law.Law`) – Initialized Law which should be registered

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, FileExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         FileExists(
>>>             name="Create pyproject.toml",
>>>             path=Path("./pyproject.toml")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

> **Warning:** The laws should never depend on each other, because the execution may not happen in the same order the laws were registered. Instead, organize the depending rules in one law to resolve any dependency conflicts.

release_lock_file()
Releasing the previously created lock file if exists.

property rules
Return all the registered laws' rules.

4.1.10 Module contents

CHAPTER

FIVE

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/gabor-boros/hammurabi/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

5.1.4 Write Documentation

Hammurabi could always use more documentation, whether as part of the official Hammurabi docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/gabor-boros/hammurabi/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *hammurabi* for local development.

1. Fork the *hammurabi* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/hammurabi.git
```

3. Install your local copy. Assuming you have poetry installed, this is how you set up your fork for local development:

```
$ cd hammurabi/
$ poetry install
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass linters and the tests:

```
$ poetry shell
$ make lint
$ make test
```

   You will need make not just for executing the command, but to build (and test) the documentations page as well.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.7 and 3.8.

5.4 Releasing

A reminder for the maintainers on how to release. Make sure all your changes are committed (including an entry in CHANGELOG.rst).

After all, create a tag and a release on GitHub. The rest will be handled by Travis.

Please follow this checklist for the release:

1. Make sure that formatters are not complaining (`make format` returns 0)
2. Make sure that linters are not complaining (`make lint` returns 0)
3. Update CHANGELOG.rst
4. Update version in `pyproject.toml` and `docs/conf.py`
5. Create a new Release on GitHub with a detailed release description based on the previous releases.

CHAPTER

SIX

VULNERABILITIES

Note: Important! In case you found vulnerability or security issue in one of the libraries we use or somewhere else in the code, please contact us via e-mail at gabor.brs@gmail.com. Please do not use this channel for support.

6.1 Reporting vulnerabilities

6.1.1 What is vulnerability?

Vulnerability is a cyber-security term that refers to a flaw in a system that can leave it open to attack. The vulnerability may also refer to any type of weakness in a computer system itself, in a set of procedures, or in anything that leaves information security exposed to a threat. - by techopedia

6.1.2 In case you found a vulnerability

In case you found vulnerability or security issue in one of the libraries we use or somewhere else in the code, please do not publish it, instead, contact us via e-mail at gabor.brs@gmail.com. We will take the necessary steps to fix the issue. We are handling the vulnerabilities privately.

To make report processing easier, please consider the following:

- Use clear and expressive subject
- Have a short, clear, and direct description including the details
- Include OWASP link, CVE references or links to other public advisories and standards
- Add steps on how to reproduce the issue
- Describe your environment
- Attach screenshots if applicable

Note: This *article* is a pretty good resource on how to report vulnerabilities.

In case you have any further questions regarding vulnerability reporting, feel free to open an issue on GitHub.

CHAPTER

SEVEN

CREDITS

7.1 Development Lead

- Gábor Boros (@gabor-boros)

7.2 Contributors

Special thanks to Péter Turi (@turip) for the initial idea.

Check the whole list of contributors here.

CHAPTER

EIGHT

CHANGELOG

All notable changes to this project will be documented in this file. The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

8.1 [0.1.0] - 2020-03-12

8.1.1 Added

- **Basic file manipulations**
  - Create file
  - Create files
  - Remove file
  - Remove files
  - Empty file
- **Basic directory manipulations**
  - Create directory
  - Remove directory
  - Empty directory
- **Basic file and directory operations**
  - Change owner
  - Change mode
  - Move file or directory
  - Copy file or directory
  - Rename file or directory
- **Plain text/general file manipulations**
  - Add line
  - Remove line
  - Replace line
- **INI file specific manipulations**
  - Add section

  - Remove section
  - Rename section
  - Add option
  - Remove option
  - Rename option

- **Miscellaneous**
  - Initial documentation
  - CI/CD integration

8.2 [Unreleased]

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

h

INDEX

L

M

O

P

R

S

T

V