groundwork-validation Documentation

Release 0.1.3

team useblocks

May 08, 2017

Contents

groundwork framework

groundwork is a plugin based Python application framework, which can be used to create various types of applications: console scripts, desktop apps, dynamic websites and more.

Visit groundwork.useblocks.com or read the technical documentation for more information.

This Python package is designed for applications, which are based on the groundwork application framework.

All of its plugins and patterns are focused on application validation during runtime.

This package contains the following groundwork extensions:

Plugins

- *GwDbValidator* - Validates automatically all database model requests.

Patterns

- *GwValidatorsPattern* - Provides functions to hash and validate python objects.
- *GwDbValidatorsPattern* - Allows the registration of database model classes to validate retrieved data on each request.
- *GwFileValidatorsPattern* - Creates and validates hashes for given files.
- *GwCmdValidatorsPattern* - Validates output and return code of given command (E.g. to validate an installed version of a tool)

CHAPTER 1

Why validation is needed

Validation is mostly needed, if your application needs input data and must be sure that this data is valid and not somehow corrupted.

A common case is the usage of files, which must be copied from an external source. During the transport over the network, the data may get corrupted. To be sure that this is not the case, a hash of this file can be build and stored beside the file. After the file is downloaded, the hash is rebuild and compared to the stored one.

Another use case is the usage of databases. If your application is the only one which is allowed to store and change specific data inside a database, you should be able to validate these data before your plugin is using it again (This use case is supported by *GwDbValidatorsPattern* and *GwDbValidator*).

CHAPTER 2

Who requests validation?

In most cases validation may be overengineered, if you are developing a small script for yourself.

However there are scenarios and domains, which need a proven validation of data, so that your application is allowed and verified to be used inside this domains.

For instance if you are developing solutions for the automotive industry and your solutions may affect the software, which runs on electronic control units (ECUs) of a car, your application must be ISO 26262 compliant. And this normally needs a proven validation of in- and output data (beside a lot of other stuff).

CHAPTER 3

Installation

Via pip

To install `groundwork-validation` simply use `pip`:

```
pip install groundwork-validation
```

From sources

Using git and pip:

```
git clone https://github.com/useblocks/groundwork-validation
cd groundwork-validation
pip install -e .
```

CHAPTER 4

Content

Plugins

GwDbValidator

This plugin automatically activates the validation of all database models, which are and will be registered via groundwork-database.

On activation `GwDbValidator` fetches all existing database models and activates their validation by using *`register()`* of *`GwValidatorsPattern`*.

It also registers a receiver to get notified, if a new database model is registered. If this is the case, it also registers a new validator for this new model.

Activation and Usage

All you have to do is to activate the plugin, which is done by adding its name to your application configuration:

```
LOAD_PLUGINS = ["MyDbPlugin", "MyOtherPlugin", "GwDbValidator"]
```

That's it. From now on all important database actions get validated.

Patterns

GwValidatorsPattern

This pattern allows plugins to register validators, which can be used to hash and validate python objects.

A validator can be configured to use a specific hash algorithm and hash specific attributes of an given object only. This maybe necessary, if unhashable python object types are used inside given object.

Note: GwValidatorsPattern uses the pickle function of Python to build a hashable, binary-based representation of your data. There are some data types, which can not be pickeled. In this case the validator must be configured to ignore these specific attributes of your data.

Register a new validator

To register a new validator, a plugin must inherit from *GwValidatorsPattern* and use *register()* for registration:

```
from groundwork_validation.patterns import GwValidatorsPattern

 class My_Plugin(GwValidatorsPattern):
     def __init__(self, app, **kwargs):
         self.name = "My_Plugin"
         super(My_Plugin, self).__init__(app, **kwargs)

     def activate(self):
         self.validator = self.validators.register("my_validator", "test validator")
```

Creating a hash

Hashes can be build for nearly each python object by using *hash()*:

```
class My_Plugin(GwValidatorsPattern):
    ...

    def get_hash(self):
         data = "test this"
         self.my_hash = self.validator.hash(data)
```

Validate an object by given hash

To validate an object, all you need is the hash and the function *validate()*:

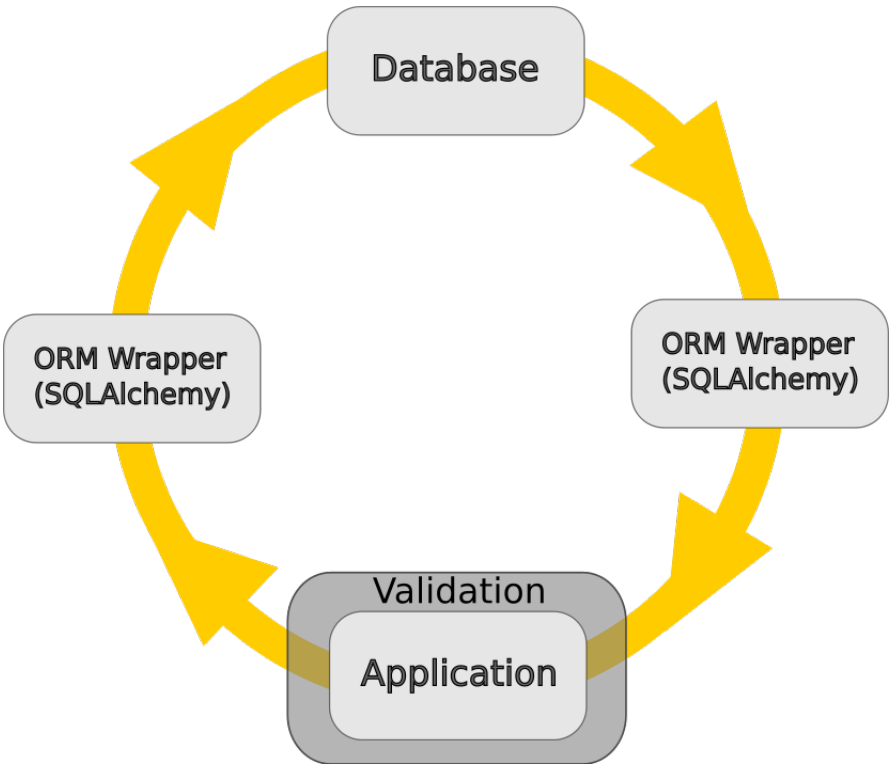
```
class My_Plugin(GwValidatorsPattern):
    ...

    def validate_hash(self):
        data = "test this"
        if self.validator.validate(data, self.my_hash) is True:
            print("Data is valid")
        else:
            print("Data is invalid. We stop here!")
            sys.exit(1)
```

Note: The plugin developer is responsible for safely storing hashes (e.g. inside a database).

GwDbValidatorsPattern

This patterns provides functions to automatically hash and validate data requests on SQLAlchemy models.

It is used to prove that data handling of used libraries and services works correct. The below image shows the flow of data which is stored to a database and requested back. As you can see at least 3 libraries/services are used, which behavior and source code is not under your full control.



Every time a registered database model is updated and uploaded to the database (add -> commit), `GwDbValidatorsPattern` creates and stores a hash of the updated data model.

And every time a request is made on a registered database model (e.g by model.query.filter_by(x="abc").all()), `GwDbValidatorsPattern` validates each received row against stored hashes.

Hashes are stored inside a database (via groundwork-database) and based on its configuration, an external database may be used so that hashes are still available and valid after application restarts.

Register a new database validator

To register a new database validator, a plugin must inherit from *GwDbValidatorsPattern* and use *register()* for registration:

```
from groundwork_validation.patterns import GwDbValidatorsPattern

class My_Plugin(GwDbValidatorsPattern):
    def __init__(self, app, **kwargs):
        self.name = "My_Plugin"
        super(My_Plugin, self).__init__(app, **kwargs)
        self.db = None
        self.Test = None
```

```
    def activate(self):

        # START: groundwork-database related configuration
        ##################################################################

        # Let's create a new database, which models shall use validated request.
        self.db = self.app.databases.register("test_db",
                                              "sqlite://",
                                              "database for test values")

        # A simple SQLAlchemy database model
        class Test(self.db.Base):
            __tablename__ = "test"
            id = Column(Integer, primary_key=True)
            name = Column(String(512), nullable=False, unique=True)

        # Register our database model
        self.Test = self.db.classes.register(Test)
        # Create all tables
        self.db.create_all()

        ##################################################################
        # END: groundwork-database related configuration

        # Register and activate validation for your model
        self.validators.db.register("db_test_validator",
                                    "my db test validator",
                                    self.Test)
```

Validate requests

Your validation has already started. The registration of a database model is enough to start the validation for each request. If a validation problem occurs, groundwork-validation will throw the exception *ValidationError*.

Test validation

To test the validation, you need to manipulate the data of a stored and monitored data model. This could be done via an external database editor like the Sqlite Browser or by executing SQL statements directly:

```
from groundwork_validation.patterns import GwDbValidatorsPattern

class My_Plugin(GwDbValidatorsPattern):
    ...

    def activate(self):
        ...
        my_test = self.Test(name="blub")
        self.db.add(my_test)
        self.db.commit()
        self.db.query(self.Test).all()

        my_test.name = "Boohaaaa"
        self.db.add(my_test)
        self.db.commit()
        self.db.query(self.Test).all()
```

```
        # Execute sql-statement, which does not trigger the sqlalchemy events.
        # So no hash gets updated.
        self.db.engine.execute("UPDATE test SET name='not_working' WHERE id=1")

        # Reloads the data from db and will throw an exception
        self.db.session.refresh(my_test)
```

Configuration

`GwDbValidatorsPattern` stores the hashes in its own database. Like other databases in groundwork, the used database connection string can be configured inside the application configuration file by setting **HASH_DB**:

```
HASH_DB = "sqlite://%s/hash_db" % APP_PATH
```

The format of the connection string is documented inside the SQLAlchemy documentation.

If no connection string is configured, **"sqlite://"** is used as default value.

Technical background

To provide a reliable validation, the *`GwDbValidatorsPattern`* hooks into the event system of SQLAlchemy to get notified about each important action and run own validation tasks.

To store its own hashes, `GwDbValidatorsPattern` is using its own database, which is registered and available in groundwork under the name **hash_db**.

For each database model, `GwDbValidatorsPattern` registers a validator with the help of *`GwValidatorsPattern`*. As attributes only the table columns are taking into account. So no additional attributes like SQLALchemy internal ones or model functions are used.

Storing data

`GwDbValidatorsPattern` has registered its own hash creation function for the SQLAlchemy events **after_update** and **after_insert**.

If one of these events is triggered, `GwDbValidatorsPattern` gets the model instance and creates with the help of *`GwValidatorsPattern`* a new hash.

This hash gets stored together with an ID into the hash database. The ID must be unique and our function must be able to regenerate it based on given and static information. So the ID contains: validator name, database table name and model instance id. Example: *my_validator.user_table.5*. This kind of an ID allows us to store hashes for all database models into one single database table.

Receiving data

`GwDbValidatorsPattern` has registered its own hash validation function for the SQLAlchemy event **refresh**.

If this gets called, `GwDbValidatorsPattern` retrieves the received database model instance. For this it regenerates the hash ID and requests the stored hash value. With the configured validator of the *`GwValidatorsPattern`* it validates the stored hash against the retrieved database model instance.

If the validation fails, the exception *`ValidationError`* gets raised. If this happens, the plugin developer is responsible to handle this exception the correct way.

GwFileValidatorsPattern

Creating a hash

For each file a hash value can be created. *GwFileValidatorsPattern* cares about the correct handling of files, even if the file size is too big to get handled in one step.

To create a hash, all you have to do is to use the function *hash()*:

```
from groundwork_validation.patterns import GwFileValidatorsPattern

class My_Plugin(GwFileValidatorsPattern):
    def __init__(self, app, **kwargs):
        self.name = "My_Plugin"
        super(My_Plugin, self).__init__(app, **kwargs)

    def activate(self):
        my_file = "/path/to/file.txt"

        # Generate and retrieve a string based hash value
        my_hash = self.validators.file.hash(my_file)

        # Store hash value directly into a file
        my_hash = self.validators.file.hash(my_file, hash_file = "/path/to/file.txt.
↪hash")

    def deactivate(self):
        pass
```

Please see *hash()* for a complete list of available parameters.

Validate a file

Using a hash string

For validation the function *validate()* is available:

```
from groundwork_validation.patterns import GwFileValidatorsPattern

class My_Plugin(GwFileValidatorsPattern):
    def __init__(self, app, **kwargs):
        self.name = "My_Plugin"
        super(My_Plugin, self).__init__(app, **kwargs)

    def activate(self):
        my_file = "/path/to/file.txt"

        my_hash = self.validators.file.hash(my_file)  # Generate a hash

        if self.validators.file.validate(my_file, my_hash):
            print("Hash is valid")
        else:
            print("Hash is NOT valid")

    def deactivate(self):
        pass
```

Using a hash file

It is also possible to validate a file against a hash file, which has stored the hash at the first line:

```
from groundwork_validation.patterns import GwFileValidatorsPattern

class My_Plugin(GwFileValidatorsPattern):
    def __init__(self, app, **kwargs):
        self.name = "My_Plugin"
        super(My_Plugin, self).__init__(app, **kwargs)

    def activate(self):
        my_file = "/path/to/file.txt"
        my_hash_file = "/path/to/file.hash"

        if self.validators.file.validate(my_file, hash_file=my_hash_file):
            print("Hash is valid")
        else:
            print("Hash is NOT valid")

    def deactivate(self):
        pass
```

Please see *validate()* for a complete list of available parameters.

GwCmdValidatorsPattern

The *GwCmdValidatorsPattern* can be used to valid the execution of a command.

This can helpful to verify the version of an installed tool by checking, if the output contains the correct version.

For some cases also the correct behavior can be validated by checking the correct return value or by setting a limit for the maximum allowed execution time.

Validating the output of a command

All different types of command validations are available by using the function *validate()*:

```
import sys
from groundwork_validation.patterns import GwCmdValidatorsPattern

class My_Plugin(GwCmdValidatorsPattern):
      def __init__(self, app, **kwargs):
          self.name = "My_Plugin"
          super(My_Plugin, self).__init__(app, **kwargs)

      def activate(self):
          if self.validators.cmd.validate("dir", search="my_folder"):
              print("Command 'dir' works as expected.")
          else:
              print("Command 'dir' seems not to work correctly. We stop here")
              sys.exit(1)

      def deactivate(self):
          pass
```

Instead of searching for a specific string, you can also use a regular expression:

```
# Checks for an e-mail address
if self.validators.cmd.validate("dir",
                                regex="(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.
↪]+$)"):
    print("Found at least one e-mail address")
```

Validating the return code

By validating the return code, you can easily check if the command is available and exits like expected. If the return code is not allowed, the exception *NotAllowedReturnCode* is raised:

```
import sys
from groundwork_validation.patterns import GwCmdValidatorsPattern
from groundwork_validation.patterns.gw_cmd_validators_pattern.gw_cmd_validators_
↪pattern import NotAllowedReturnCode

 class My_Plugin(GwCmdValidatorsPattern):
        def __init__(self, app, **kwargs):
            self.name = "My_Plugin"
            super(My_Plugin, self).__init__(app, **kwargs)

        def activate(self):
            try:
                if self.validators.cmd.validate("dir", search="my_folder", allowed_
↪return_codes=[0, 1]):
                    print("Command 'dir' works a expected.")
                else:
                    print("Command 'dir' seems not to work correctly. We stop here")
                    sys.exit(1)
            except NotAllowedReturnCode:
                print("Command exists with not allowed status code. Validation failed!
↪")
                sys.exit(1)
```

Setting a timeout

By default the command is killed after a timeout of 2 seconds and *CommandTimeoutExpired* is raised. You are free to set your own timeout for each validation:

```
import sys
from groundwork_validation.patterns import GwCmdValidatorsPattern
from groundwork_validation.patterns.gw_cmd_validators_pattern.gw_cmd_validators_
↪pattern \
    import NotAllowedReturnCode, CommandTimeoutExpired

class My_Plugin(GwCmdValidatorsPattern):
        def __init__(self, app, **kwargs):
            self.name = "My_Plugin"
            super(My_Plugin, self).__init__(app, **kwargs)

        def activate(self):
            try:
                if self.validators.cmd.validate("dir", search="my_folder", timeout=5):
```

```
                    print("Command 'dir' works a expected.")
                else:
                    print("Command 'dir' seems not to work correctly. We stop here")
                    sys.exit(1)
            except CommandTimeoutExpired:
                print("Command has not finished and raised a timeout. This is not␣
↪expected. We stop here!")
                sys.exit(1)
```

test:

```
pip install
```

API

Plugins

GwDbValidator

class **GwDbValidator**(*app*, ***kwargs*)

Automatically adds and activate validation to eahc database model.

activate()

During activation, a receiver is created and listing for new database models. Existing database models are collected and validation gets activated.

Returns None

deactivate()

Currently nothing happens here *sigh*

Returns None

Patterns

GwValidatorsPattern

class **GwValidatorsPattern**(*app*, ***kwargs*)

Allows the creation of hashes for python objects (and its validation).

activate()

Must be overwritten by the plugin class itself.

deactivate()

Must be overwritten by the plugin class itself.

class **ValidatorsPlugin**(*plugin*)

Cares about the Validator handling on plugin level.

get(*name*)

Returns a single or a list of validator instance, which were registered by the current plugin.

Parameters **name** – Name of the validator. If None, all validators of the current plugin are returned.

Returns Single or list of Validator instances

register(*name*, *description*, *algorithm=None*, *attributes=None*)
Registers a new validator on plugin level.

Parameters

- **name** – Unique name of the validator
- **description** – Helpful description of the validator
- **algorithm** – A hashlib compliant function. If None, hashlib.sha256 is taken.
- **attributes** – List of attributes, for which the hash must be created. If None, all contained attributes are used.

Returns Validator instance

unregister(*name*)

class ValidatorsApplication(*app*)
Cares about the Validator handling on application level.

get(*name*, *plugin*)
Returns a single or a list of validator instance

Parameters

- **name** – Name of the validator. If None, all validators are returned.
- **plugin** – Plugin instance, which has registered the requested validator. If None, all validators are returned.

Returns Single or list of Validator instances

register(*name*, *description*, *plugin*, *algorithm=None*, *attributes=None*)
Registers a new validator on application level.

Parameters

- **name** – Unique name of the validator
- **description** – Helpful description of the validator
- **algorithm** – A hashlib compliant function. If None, hashlib.sha256 is taken.
- **attributes** – List of attributes, for which the hash must be created. If None, all contained attributes are used.
- **plugin** – Plugin instance, for which the validator gets registered.

Returns Validator instance

unregister(*name*)

class Validator(*name*, *description*, *algorithm=None*, *attributes=None*, *plugin=None*)
Represent the final validator, which provides functions to hash a given python object and to validate a python object against a given hash.

get_hash_object()
Returns a hash object, which can be used as input for validate functions.

Returns An unused hash object

hash(*data*, *hash_object=None*, *return_hash_object=False*, *strict=False*)
Generates a hash of a given Python object.

Parameters

- **data** – Python object

- **return_hash_object** – If true, the complete hashlib object is returned instead of a hexdigest representation as string.
- **hash_object** – An existing hash object, which will be updated. Instead of creating a new one.
- **strict** – If True, all configured attributes **must** exist in the given data, otherwise an exception is thrown.

Returns hash as string

validate(*data*, *hash_string*)
Validates a python object against a given hash

Parameters

- **data** – Python object
- **hash_string** – hash as string, which must be compliant to the configured hash algorithm of the used validator.

Returns True, if object got validated by hash. Else False

GwDbValidatorsPattern

class GwDbValidatorsPattern(*app*, ***kwargs*)
Allows the validation of database model requests.

Builds automatically hashes of table rows/model instances and validates these hashes, if a request is made on these rows.

activate()
Must be overwritten by the plugin class itself.

deactivate()
Must be overwritten by the plugin class itself.

class DbValidatorsPlugin(*plugin*)
Cares about database validators on plugin level

get(*name*)

register(*name*, *description*, *db_class*)
Registers a new database model and starts its validation.

Parameters

- **name** – Unique name
- **description** – Meaningful description
- **db_class** – sqlalchemy based database model

Returns Instance of DbValidator

unregister(*name*)

class DbValidatorsApplication(*app*)
Cares about database validators on application level

get(*name*, *plugin*)

register(*name*, *description*, *db_class*, *plugin*)
Registers a new database model and starts its validation.

Parameters

- **name** – Unique name
- **description** – Meaningful description
- **db_class** – sqlalchemy based database model
- **plugin** – Plugin, which registers the DbValidator

Returns Instance of DbValidator

unregister(*name*)

class **DbValidator**(*name*, *description*, *db_class*, *db*, *hash_model*, *plugin=None*)
Class for storing a database validator. For each registered database validator an instance of this class gets created and configured.

class **ValidationError**
Exception, which is thrown if a validation fails.

GwFileValidatorsPattern

class **GwFileValidatorsPattern**(*app*, ***kwargs*)
Allows the creation and validation of hashes for given files.

Usage:

```
from groundwork_validation.patterns import GwFileValidatorsPattern

class My_Plugin(GwFileValidatorsPattern):
    def __init__(self, app, **kwargs):
        self.name = "My_Plugin"
        super(My_Plugin, self).__init__(app, **kwargs)

    def activate(self):
        my_hash = self.validators.file.hash("/path/to/file.txt")
        self.validators.file.validate("/path/to/file.txt", my_hash)
```

activate()
Must be overwritten by the plugin class itself.

deactivate()
Must be overwritten by the plugin class itself.

class **FileValidatorsPlugin**(*plugin*)

hash(*file*, *validator=None*, *hash_file=None*, *blocksize=65536*, *return_hash_object=False*)
Creates a hash of a given file.

Parameters

- **file** – file path of the hashable file
- **validator** – validator, which shall be used. If none is given, a default validator will be used. validator should be registered be the GwValidatorsPattern. Default is None
- **hash_file** – Path to a file, which is used to store the calculated hash value. Default is None
- **blocksize** – Size of each file block, which is used to update the hash. Default is 65536

- **return_hash_object** – Returns the hash object instead of the hash itself. Default is False

Returns string, which represents the hash (hexdigest)

validate(*file*, *hash_value=None*, *hash_file=None*, *validator=None*, *blocksize=65536*)
Validates a file against a given hash. The given hash can be a string or a hash file, which must contain the hash on the first row.

Parameters

- **file** – file path as string
- **hash_value** – hash, which is used for comparision
- **hash_file** – file, which contains a hash value
- **validator** – groundwork validator, which shall be used. If None is given, a default one is used.
- **blocksize** – Size of each file block, which is used to update the hash.

Returns True, if validation is correct. Otherwise False

GwCmdValidatorsPattern

class GwCmdValidatorsPattern(*app*, ***kwargs*)
Allows the validation of output, return code and execution time of a given command.

Usage:

```
class My_Plugin(GwCmdValidatorsPattern):
    def __init__(self, app, **kwargs):
        self.name = "My_Plugin"
        super(My_Plugin, self).__init__(app, **kwargs)

    def activate(self):
        if self.validators.cmd.validate("dir", search="my_folder"):
            print("Command 'dir' works a expected.")
        else:
            print("Command 'dir' seems not to work correctly. We stop here")
            sys.exit(1)

    def deactivate(self):
        pass
```

activate()
Must be overwritten by the plugin class itself.

deactivate()
Must be overwritten by the plugin class itself.

class CmdValidatorsPlugin(*plugin*)

validate(*command*, *search=None*, *regex=None*, *timeout=2*, *allowed_return_codes=None*, *decode='utf-8'*)
Validates the output of a given command.

The validation can be based on a simple string search or on a complex regular expression. Also the return_code can be validated. As well as the execution duration by setting a timeout.

Parameters

- **command** – string, which is used as command for a new subprocess. E.g. 'git -v'.
- **search** – string, which shall be contained in the output of the command. Default is None
- **regex** – regular expression, which is tested against the command output. Default is None
- **timeout** – Time ins seconds, after which the execution is stopped and the validation fails. Default is 2 seconds
- **allowed_return_codes** – List of allowed return values. Default is []
- **decode** – Format of the console encoding, which shall be used. Default is 'utf-8'

Returns True, if validation succeeded. Else False.

class NotAllowedReturnCode

class CommandTimeoutExpired

Python Module Index

g

Index

U

V