karton Documentation

Release 5.0.0

CERT Polska

Aug 02, 2022

KARTON REFERENCE:

Karton is a library made for analysis backend orchestration. Allows you to build flexible malware analysis pipelines and attach new Karton Services with ease.

This is achieved by combining powers of a few existing solutions, karton just glues them together and allows you to have some sane amount of abstraction over them.

Karton ecosystem consists of:

- Redis - store used for message exchange between Karton subsystems
- Temporary object storage compatible with Amazon S3 API, holds all the heavy objects (aka Resources) like samples, analyses or memory dumps. The recommended one is MinIO.

Task routing and data exchange is achieved with the help of **Karton-System** - core of the Karton, which routes the tasks and keeps everything in order (task lifecycle, garbage collection etc.)

```python
from karton.core import Karton, Task, Resource

class GenericUnpacker(Karton):
    """
    Performs sample unpacking
    """
    identity = "karton.generic-unpacker"
    filters = [
        {
            "type": "sample",
            "kind": "runnable",
            "platform": "win32"
        }
    ]

    def process(self, task: Task) -> None:
        # Get sample object
        packed_sample = task.get_resource('sample')
        # Log with self.log
        self.log.info(f"Hi {packed_sample.name}, let me analyze you!")
        ...
        # Send our results for further processing or reporting
        task = Task(
            {
               "type": "sample",
               "kind": "raw"
            }, payload = {
               "parent": packed_sample,
               "sample": Resource(filename, unpacked)
            })
        self.send_task(task)

if __name__ == "__main__":
    # Here comes the main loop
    GenericUnpacker.main()
```

CHAPTER

ONE

BREAKING CHANGES

This chapter will describe significant changes introduced in major version releases of Karton. Versions before 4.0.0 were not officially released, so they have value only for internal purposes. Don't worry about it if you are a new user.

1.1 What is changed in Karton 5.0.0

Karton-System and core services are still able to communicate with previous versions.

- Changed name of `karton.ini` section that contains S3 client configuration from `[minio]` to `[s3]`.

  In addition to this, you need to add a URI scheme to the `address` field and remove the `secure` field. If `secure` was 0, correct scheme is `http://`. If `secure` was 1, use `https://`.

```
- [minio]
+ [s3]
  access_key = karton-test-access
  secret_key = karton-test-key
- address = localhost:9000
+ address = http://localhost:9000
  bucket = karton
- secure = 0
```

  v5.0.0 maps `[minio]` configuration to correct `[s3]` configuration internally, but `[minio]` scheme is considered deprecated and can be removed in further major release.

- Karton library uses Boto3 library as a S3 client instead of Minio-Py underneath. You may want to check if your code relies on exceptions thrown by previous S3 client.
- `karton.core.Config` interface is changed. `config`, `minio_config` and `redis_config` attributes are no longer available.
- We noticed lots of issues caused by calling factory method `main()` on instance instead of class, which can be misleading (:py:meth:`karton.core.base.KartonBase.main` actually creates own instance of Karton service internally, so the initialization is doubled). To notice these errors more quickly, we prevented `main()` call on `KartonBase` instance

```
if __name__ == "__main__":
    MyConsumer.main()  # correct

if __name__ == "__main__":
    MyConsumer().main()  # throws TypeError
```

- `karton.core.Consumer.process` no longer accepts no arguments. First argument of this method is the incoming task.

```
# Correct
class MyConsumer(Karton):
    def process(self, task: Task) -> None:
        ...

# Wrong from v5.0.0
class MyConsumer(Karton):
    def process(self) -> None:
        ...
```

1.2 What is changed in Karton 4.0.0

Karton-System and core services are still compatible with both 3.x and 2.x versions.

- `SHA256` is evaluated always when `Resource` is created. If you already know it and don't want it to be recalculated, pass the hash to the constructor via `sha256=` argument.

```
sample = Resource(path="sample.exe", sha256="2e5d...")
```

- `DirectoryResource` has been removed in favor of `Resource.from_directory`. Resources created using this method are still deserialized to the `RemoteDirectoryResource` form by older Karton versions. `RemoteDirectoryResource` has been merged into `RemoteResource`, so all resources containing Zip files can be unzipped even if they were created as regular files.
- Asynchronous tasks has been removed. Busy waiting should be used instead.
- All crashed tasks are preserved in `Crashed` state until they are removed by Karton-System (default is 72 hours) or retried by user. Keep in mind that they hold all the referenced resources, so keep an eye on that queue.

1.3 What is changed in Karton 3.0.0

Karton-System and other core services in 3.x are compatible with 2.x. But if you want to use 3.x in Karton service code, all core services need to be upgraded first.

The good news:

- Karton subsystems expose the library version and class docstring in `karton.binds`
- Config is explicit and get by default from `karton.ini` file (yup, it's `karton.ini` not `config.ini`). But you can still provide another path if you want.
- There is no need to provide a suffix `".test"` as a part of identity for non-persistent consumer queues. Just set `persistent=False` in your Karton subsystem class
- You can provide `identity` as an argument.

So, instead of that code:

```
# Consumer part

class Subsystem(Karton):
```

(continues on next page)

(continued from previous page)

```
    identity = "karton.subsystem.test"
    filters = {...}

config = Config("config.ini")
subsystem = Subsystem(config).loop()

# Producer part

class NamedProducer(Producer):
    identity = "karton.named-producer"

config = Config("config.ini")
producer = NamedProducer(config).send_task(...)
```

You can write that code:

```
# Consumer part

class Subsystem(Karton):
    identity = "karton.subsystem"
    filters = {...}
    persistent = False

subsystem = Subsystem().loop()

# Producer part

producer = Producer(identity="karton.named-producer").send_task(...)
```

The bad news (for porting):

- Resource classes are completely reworked.
  - Resources are strictly divided to local (uploadable) and remote (downloadable) ones. The inheritance structure is different than in 2.x, so check the API first.
  - There is no `sha256` field, but `metadata` dictionary instead. For compatibility reasons: we expose `sha256` from Karton 2.x as `metadata["sha256"]` and back. New subsystems should not rely on that behavior.
  - `flags` are also not exposed.
  - Removed `is_directory` method.

    If you need to check whether your resource is directory, use `isinstance(resource, DirectoryResourceBase)` instead.
  - Remote resources are now lazy-objects bound with MinIO, so we can directly get the contents instead of using proxy methods.

    Code from 2.x:

    ```
    sample = self.current_task.get_resource("sample")
    # Calling Consumer method to get local version of resource
    local_sample = self.download_resource(sample)
    # Get the contents
    sample_content = local_sample.content
    ```

must be ported to:

```
sample = self.current_task.get_resource("sample")
# Contents will be lazy-loaded
# If you want to download them directly: use sample.download()
sample_content = sample.content
```

All related `Consumer` methods like `download_resource()` or `download_to_temporary_folder()` are completely removed. These methods were incomplete and inconsistent, especially for directories. Now, the whole power behind the Resource features is available directly via object methods.

- Removed `PayloadBag` wrappers with resource iterator methods. They provided additional level of complexity without adding new capabilities. There are classic dictionaries in place of them.

- Task classes also changed a bit
  - `payload_contains()` is renamed to `has_payload()` and doesn't check only non-persistent payload existence, but includes persistent payloads as well.
  - `persistent_payload_contains()` is renamed to `is_payload_persistent()`
  - `get_resource()` is not just `get_payload()` alias and provides type checking. It does not accept the *default* argument.
  - Instead of `get_resources()`, `get_directory_resources()` and `get_file_resources()` - use `iterate_resources()` and do type checking yourself.
- Removed 'kpm' (some kind of helper scripts will be provided in future versions, that one was outdated anyway)

CHAPTER

TWO

GETTING STARTED

2.1 Installation

You can get the Karton framework from pip:

```
python -m pip install karton-core
```

Or, if you're feeling adventurous, download the sources using git and install them manually.

In addition to Karton core library, you'll also need to setup S3-compatible storage like MinIO and Redis server.

2.2 Configuration

Each Karton subsystem needs a `karton.ini` file that contains the connection parameters for Redis and S3.

You can also use this file to store custom fields and use them e.g. by *Extending configuration*.

By default, the config class will look for the config file in several places, but let's start by placing one in the root of our new Karton subsystem.

```
[s3]
secret_key = minioadmin
access_key = minioadmin
address = http://localhost:9000
bucket = karton

[redis]
host=localhost
port=6379
```

If everything was configured correctly, you should now be able to run the `karton-system` broker and get `"Manager karton.system started"` signaling that it was able to connect to Redis and S3 correctly.

2.3 Docker Compose development setup

Check out repository called Karton playground that provides similar setup coupled with MWDB Core and few open-source Karton services.

If you're just trying Karton out or you want a mimimal, quick & easy development environment setup, check out the `dev` folder in the Karton root directory.

It contains a small docker-compose setup that will setup the minimal development environment for you.

All you have to do is run

```
docker-compose up --build
```

And then connect additional Karton systems using the `karton.ini.dev` config file.

```
karton-classifier --config-file dev/karton.ini.dev
```

2.4 Writing your first Producer and Consumer

Since all great examples start with foobar, that's exactly what we're going to do. Let's start by writing a producer that spawns new tasks.

```
from karton.core import Producer, Task

if __name__ == "__main__":
    foo_producer = Producer(identity="foobar-producer")
    for i in range(5):
        task = Task(headers={"type": "foobar"}, payload={"data": i})
        foo_producer.send_task(task)
```

That was pretty short! Now for a bit longer consumer:

```
from karton.core import Consumer, Task

class FooBarConsumer(Consumer):
    identity = "foobar-consumer"
    filters = [
        {
            "type": "foobar"
        }
    ]
    def process(self, task: Task) -> None:
        num = task.get_payload("data")
        print(num)
        if num % 3 == 0:
            print("Foo")
        if num % 5 == 0:
            print("Bar")

if __name__ == "__main__":
    FooBarConsumer.main()
```

If we now run the consumer and spawn a few "foobar" tasks we should get a few foobars logs in return:

```
[INFO] Service foo-consumer started
[INFO] Service binds created.
[INFO] Binding on: {'type': 'foobar'}
[INFO] Received new task - 884880e0-e5fc-4a71-a93a-08f0caa92889
0
Foo
Bar
[INFO] Task done - 884880e0-e5fc-4a71-a93a-08f0caa92889
[INFO] Received new task - 60be2eb5-9e7e-4928-8823-a0d30bbe68ec
1
[INFO] Task done - 60be2eb5-9e7e-4928-8823-a0d30bbe68ec
[INFO] Received new task - 301d8a50-f21e-4e33-b30e-0f3b1cdbda03
2
[INFO] Task done - 301d8a50-f21e-4e33-b30e-0f3b1cdbda03
[INFO] Received new task - 3bb9aea2-4027-440a-8c21-57b6f476233a
3
Foo
[INFO] Task done - 3bb9aea2-4027-440a-8c21-57b6f476233a
[INFO] Received new task - 050cdace-05b0-4648-a070-bc4a7a8de702
4
[INFO] Task done - 050cdace-05b0-4648-a070-bc4a7a8de702
[INFO] Received new task - d3a39940-d64c-4033-a7da-80eae9786631
5
Bar
[INFO] Task done - d3a39940-d64c-4033-a7da-80eae9786631
```

Check *Karton service examples* for more details.

2.5 Command-line interface (CLI)

When you install `karton-core`, a new command called `karton` is added to your terminal. You can inspect its capabilities by running it:

```
(venv) user@computer ~/> karton
usage: karton [-h] [--version] [-c CONFIG_FILE] [-v] {list,logs,delete,configure} ...

Your red pill to the karton-verse

positional arguments:
{list,logs,delete,configure}
                        sub-command help
    list                List active karton binds
    logs                Start streaming logs
    delete              Delete an unused karton bind
    configure           Create a new configuration file

optional arguments:
-h, --help            show this help message and exit
--version             show program's version number and exit
-c CONFIG_FILE, --config-file CONFIG_FILE
                        Alternative configuration path
```

(continues on next page)

(continued from previous page)

```
-v, --verbose         More verbose log output
```

The commands are small, utility scripts that are supposed to make maintaining karton a bit easier.

list

List active karton consumers, this can be handy if you don't have a dashboard deployed

logs [–filter FILTER]

Subscribe to logs coming in from all services. This is very useful if you're trying to hunt down errors or some funky behavior. You can specify a filter that will limit incoming log messages, for example, to a specific identity - `--filter "karton.classifier"`.

delete <identity>

Delete a persistent queue that's no longer needed.

configure [–force]

Create a new `karton.ini` configuration file. The config wizard will ask you about various parameters, like the S3 credentials, Redis host, etc. and then save the information into a config file.

CHAPTER

THREE

KARTON SERVICE EXAMPLES

Here are few examples of common Karton system patterns.

3.1 Producer services

```
import sys
from karton.core import Config, Producer, Task, Resource

config = Config("karton.ini")
producer = Producer(config)

filename = sys.argv[1]
with open(filename, "rb") as f:
    contents = f.read()

resource = Resource(os.path.basename(filename), contents)

task = Task({"type": "sample", "kind": "raw"})

task.add_resource("sample", resource)
task.add_payload("tags", ["simple_producer"])
task.add_payload("additional_info", ["This sample has been added by simple producer␣
↪example"])

logging.info('pushing file to karton %s, task %s' % (name, task))
producer.send_task(task)
```

3.2 Consumer services

Consumer has to define `identity`, a name used for identification and binding in RMQ and `filters` - a list of dicts determining what types of tasks the service wants to process.

Elements in the list are OR'ed and items inside dicts are AND'ed.

```
import sys
from karton.core import Config, Consumer, Task, Resource

class Reporter(Consumer):
```

(continues on next page)

(continued from previous page)

```
    identity = "karton.reporter"
    filters = [
        {
            "type": "sample",
            "stage": "recognized"
        },
        {
            "type": "sample",
            "stage": "analyzed"
        },
        {
            "type": "config"
        }
    ]
```

Above example accepts headers like:

```
{
    "type": "sample",
    "stage": "recognized",
    "kind": "runnable",
    "platform": "win32",
    "extension": "jar"
}
```

or

```
{
    "type": "config",
    "kind": "cuckoo1"
}
```

but not

```
{
    "type": "sample",
    "stage": "something"
}
```

Next step is to define *process* method, this is handler for incoming tasks that match our filters.

```
def process(self, task: Task) -> None:
   if task.headers["type"] == "sample":
       return self.process_sample(task)
   else:
       return self.process_config(task)

def process_sample(self, task: Task) -> None:
    sample = task.get_resource("sample")
    # ...

def process_config(self, task: Task) -> None:
```

(continues on next page)

(continued from previous page)

```
    config = task.get_payload("config")
    # ...
```

`task.headers` gives you information on why task was routed and methods like *get_resource* or *get_payload* allow you to get resources or metadata from task.

Finally, we need to run our module, we get this done with *loop* method, which blocks on listening for new tasks, running *process* when needed.

```
if __name__ == "__main__":
    c = Reporter()
    c.loop()
```

3.3 Karton services (Producer + Consumer)

Karton class is simply Producer and Consumer bundled together.

As defined in *karton/core/karton.py*:

```
class Karton(Consumer, Producer):
    """
    This glues together Consumer and Producer - which is the most common use case
    """
```

Receiving data is done exactly like in Consumer. Using producer is no different as well, just use `self.send_task`.

Full-blown example below.

```
from karton.core import Karton, Task

class SomeNameKarton(Karton):
    # Define identity and filters as you would in the Consumer class
    identity = "karton.somename"
    filters = [
        {
            "type": "config",
        },
        {
            "type": "analysis",
            "kind": "cuckoo1"
        },
    ]

    # Method called by Karton library
    def process(self, task: Task) -> None:
        # Getting resources we need without downloading them locally
        analysis_resource = task.get_resource('analysis')
        config_resource = task.get_resource('config')

        # Log with self.log
        self.log.info("Got resources, lets analyze them!")
        ...
```

(continues on next page)

(continued from previous page)

```
        # Send our results for further processing or reporting
        # Producer part
        t = Task({"type": "sample"})
        t.add_resource("sample", Resource(filename, content))
        self.send_task(task)
```

3.4 Log consumer

By default, all logs created in Karton systems are published to a specialized log consumer using the Redis `PUBSUB` pattern.

This is a very simple example of a system that implements the `LogConsumer` interface and prints logs to stderr.

```
import sys
from karton.core.karton import LogConsumer


class StdoutLogger(LogConsumer):
    identity = "karton.stdout-logger"

    def process_log(self, event: dict) -> None:
        # there are "log" and "operation" events
        if event.get("type") == "log":
            print(f"{event['name']}: {event['message']}", file=sys.stderr, flush=True)


if __name__ == "__main__":
    StdoutLogger().loop()
```

CHAPTER

FOUR

HEADERS, PAYLOADS AND RESOURCES

Task consists of two elements: **headers** and **payload**.

4.1 Task headers

Headers specify the purpose of a task and determine how task will be routed by karton-system. They're defined by flat collection of keys and values.

Example:

```
task = Task(
    headers = {
        "type": "sample",
        "kind": "runnable",
        "platform": "win32",
        "extension": "dll"
    }
)
```

Consumers listen for specific set of headers, which is defined by *filters*.

```
class GenericUnpacker(Karton):
    """
    Performs sample unpacking
    """
    identity = "karton.generic-unpacker"
    filters = [
        {
            "type": "sample",
            "kind": "runnable"
        },
        {
            "type": "sample",
            "kind": "script",
            "platform": "win32"
        }
    ]

    def process(self, task: Task) -> None:
        # Get incoming task headers
```

(continues on next page)

(continued from previous page)

```
        headers = task.headers
        self.log.info("Got %s sample from %s", headers["kind"], headers["origin"])
```

If Karton-System finds that a task matches any of subsets defined by consumer queue filters then the task will be routed to that queue.

Following the convention proposed in examples above, it means that `GenericUnpacker` will get all tasks contain samples directly runnable in sandboxes (regardless of target platform) or Windows 32-bit only scripts.

Headers can be used to process our input differently, depending on the kind of sample:

```
class GenericUnpacker(Karton):
    ...

    def process(self, task: Task) -> None:
        # Get incoming task headers
        headers = task.headers
        if headers["kind"] == "runnable":
            self.process_runnable()
        elif headers["kind"] == "script":
            self.process_script()
```

Few headers have special meaning and are added automatically by Karton to incoming/outgoing tasks.

- `{"origin": "<identity>"}` specifies the identity of task sender. It can be used for listening for tasks incoming only from predefined identity.
- `{"receiver": "<identity>"}` is added by Karton when task is routed to the consumer queue. On the receiver side, value is always equal to `self.identity`

4.2 Filter patterns

New in version 5.0.0.

Filter matching follows two simple rules. If we want task to be routed to the consumer:

- task headers must match **any** of consumer filters
- task headers match consumer filter if they match **all values** defined in filter

Starting from 5.0.0, consumer filters support basic wildcards and exclusions.

Pattern	Meaning
`{"foo": "bar"}`	matches ‘bar’ value of ‘foo’ header
`{"foo": "!bar"}`	matches any value other than ‘bar’ in ‘foo’ header
`{"foo": "ba?"}`	matches ‘ba’ value followed by any character
`{"foo": "ba*"}`	matches ‘ba’ value followed by any substring (including empty)
`{"foo": "ba[rz]"}`	matches ‘ba’ value followed by ‘r’ or ‘z’ character
`{"foo": "ba[!rz]"}`	matches ‘ba’ value followed by any character other than ‘r’ or ‘z’
`{"foo": "!ba[!rz]"}`	matches any value of ‘foo’ header that doesn’t match to the “bar[!rz]” pattern

Filter logic can be used to fulfill specific use-cases:

`filters` value	Meaning
`[]`	matches no tasks (no headers allowed). Can be used to turn off queue and consume tasks left.
`[{}]`	matches any task (no header conditions). Can be used to intercept all tasks incoming to Karton.
`[{"foo": "bar"}, {"foo": "baz"}]`	'foo' header is required and must have 'bar' or 'baz' value.
`[{"foo": "!*"}]`	'foo' header must be not defined.

Warning: It's recommended to use only strings in filter and header values

Although some of non-string types are allowed, they will be converted to string for comparison which may lead to unexpected results.

4.3 Task payload

Payload is also a dictionary, but it's not required to be a flat structure like headers are. Its contents do not affect the routing so task semantics must be defined by headers.

```
task = Task(
    headers = ...,
    payload = {
        "entrypoints": [
            "_ExampleFunction@12"
        ],
        "matched_rules": {
            ...
        },
        "sample": Resource("original_name.dll", path="uploads/original_name.dll")
    }
)
```

Payload can be accessed by Consumer using `Task.get_payload()` method.

```
class KartonService(Karton):
    ...
    def process(self, task: Task) -> None:
        entrypoints = task.get_payload("entrypoints", default=[])
```

But payload dictionary itself still must be **lightweight and JSON-encodable**, because it's stored in Redis along with the whole task definition.

If task operates on binary blob or complex structure, which is probably the most common use-case, payload can still be used to store the reference to that object. The only requirement is that object must be placed in separate, shared storage, available for both Producer and Consumer. That's exactly how `Resource` objects work.

4.4 Resource objects

Resources are part of a payload that represent a reference to the file or other binary large object. All objects of that kind are stored in S3-compatible storage, which is used as shared object storage between Karton subsystems.

```
task = Task(
    headers = ...,
    payload = {
        "sample": Resource("original_name.dll", path="uploads/original_name.dll")
    }
)
```

Resource objects created by producer (`LocalResource`) are uploaded to S3 and transformed to `RemoteResource` objects. RemoteResource is lazy object that allows to download the object contents via `RemoteResource.content` property.

```
class GenericUnpacker(Karton):
    ...

    def unpack(self, packed_content: bytes) -> bytes:
        ...

    def process(self, task: Task) -> None:
        # Get sample resource
        sample = task.get_resource("sample")
        # Do the job
        unpacked = self.unpack(sample.content)
        # Publish the results
        task = Task(
            headers={
                "type": "sample",
                "kind": "unpacked"
            },
            payload={
                "sample": Resource("unpacked", content=unpacked)
            }
        )
        self.send_task(task)
```

If expected resource is too big for in-memory processing or we want to launch external tools that need the file system path, resource contents can be downloaded using `RemoteResource.download_to_file()` or `RemoteResource.download_temporary_file()`.

```
class KartonService(Karton):
    ...
    def process(self, task: Task) -> None:
        archive = task.get_resource("archive")
        with archive.download_temporary_file() as f:
            # f is file-like named object
            archive_path = f.name
```

If you want to pass original sample along with new task, you can just put a reference back into its payload.

```
task = Task(
    headers={
        "type": "sample",
        "kind": "unpacked"
    },
    payload={
        "sample": Resource("unpacked", content=unpacked),
        "parent": sample  # Reference to original (packed) sample
    }
)
self.send_task(task)
```

Each resource has its own metadata store where we can provide additional information about file e.g. SHA-256 checksum

```
sample = Resource("sample.exe",
                  content=sample_content,
                  metadata={
                    "sha256": hashlib.sha256(sample_content).hexdigest()
                  })
```

Starting from v5.0.0, resources can be nested in other objects like lists or dictionaries.

```
task = Task(
    headers={
        "type": "analysis",
        "kind": "artifacts"
    },
    payload={
        "artifacts": [
            Resource("file1", content=file1),
            Resource("file2", content=file2),
            Resource("file3", content=file3)
        ]
        "parent": sample  # Reference to original (packed) sample
    }
)
self.send_task(task)
```

More information about resources can be found in API documentation.

4.5 Directory resource objects

Resource objects work well for single files, but sometimes we need to deal with bunch of artifacts e.g. process memory dumps from dynamic analysis. Very common way to do that is to pack them into Zip archive using Python zipfile module facilities.

Karton library includes a helper method for that kind of archives, called `LocalResource.from_directory()`.

```
task = Task(
    headers={
        "type": "analysis"
```

(continues on next page)

(continued from previous page)

```
    },
    payload={
        "dumps": LocalResource.from_directory(analysis_id,
                                              directory_path=f"analyses/{analysis_id}/
↪dumps"),
    }
)
self.send_task(task)
```

Files contained in `directory_path` are stored under relative paths to the provided directory path. Default compression level is `zipfile.ZIP_DEFLATED` instead of `zipfile.ZIP_STORED`.

Directory resources are deserialized to the usual `RemoteResource` objects but in contrary to the usual resources they can for example be extracted to directories using `RemoteResource.extract_temporary()`

```
class KartonService(Karton):
    ...
    def process(self, task: Task) -> None:
        dumps = task.get_resource("dumps")
        with dumps.extract_temporary() as dumps_path:
            ...
```

If we don't want to extract all files, we can work directly with `zipfile.ZipFile` object, which will be internally downloaded from S3 to the temporary file using `RemoteResource.download_temporary_file()` method.

```
class KartonService(Karton):
    ...
    def process(self, task: Task) -> None:
        dumps = task.get_resource("dumps")

        with dumps.zip_file() as zipf:
            with zipf.open("sample_info.txt") as info:
                ...
```

More information about resources can be found in API documentation.

4.6 Persistent payload

Part of payload that is propagated to the whole task subtree. The common use-case is to keep information related not with single artifact but the whole analysis, so they're available everywhere even if not explicitly passed by the Karton Service.

```
task = Task(
    headers=...,
    payload=...,
    payload_persistent={
        "uploader": "psrok1"
    }
)
```

Incoming persistent payload (task received by Karton Service) is merged by Karton library with the outgoing tasks (result tasks sent by Karton Service). Karton service can't overwrite or delete the incoming payload keys.

```
class KartonService(Karton):
    ...
    def process(self, task: Task) -> None:
        uploader = task.get_payload("uploader")

        assert task.is_payload_persistent("uploader")

        task = Task(
            headers=...,
            payload=...
        )
        # Outgoing task also contains "uploader" key
        self.send_task(task)
```

Regular payloads and persistent payload keys have common namespace so persistent payload can't be overwritten by regular payload as well e.g.

```
task = Task(
    headers=...,
    payload={
        "common_key": "<this will be ignored>"
    },
    payload_persistent={
        "common_key": "<and this value will be used>"
    }
)
```

Warning: Because merging strategy is quite aggressive, it's not recommended to overuse that feature. They should be treated as "analysis-wide payload". It's recommended to set them only in initial task.

Don't store any references to resources or other heavy objects here, unless you need to. Persistent payload is, as the name says, persistent, so it is propagated to the whole task subtree and **can't be removed** during analysis. Resource referenced by persistent payload won't be garbage-collected until the whole analysis (task subtree) ends, even if it's not needed by further analysis steps.

CHAPTER

FIVE

CONFIGURATION AND CUSTOMIZATION

This chapter describes how to configure and customize Karton services, including ready-made ones available on PyPi/Github.

5.1 Basic configuration

Karton services can be configured using various ways. Let's take a look at basic configuration.

```
[s3]
secret_key = minioadmin
access_key = minioadmin
address = http://localhost:9000
bucket = karton

[redis]
host=localhost
port=6379
```

Configuration values are read from various sources using the following precedence:

- `/etc/karton/karton.ini` file (global)
- `~/.config/karton/karton.ini` file (user local)
- `./karton.ini` file (subsystem local)
- `--config-path` <path> optional, additional path provided in arguments
- `KARTON_SECTION_OPTION` values from environment variables e.g. (`secret_key` option in `[s3]` section can be overridden using `KARTON_S3_SECRET_KEY` variable)
- Command-line arguments (if `Karton.main()` method is used as entrypoint)

You can build your configuration hierarchically e.g. by providing common settings in `/etc/karton/karton.ini`, service-specific settings in local `./karton.ini` and secrets in env vars.

Common Karton configuration fields are listed below:

Section	Option	Description
[s3]	address	S3 API address
[s3]	access_key	S3 API access key (username)
[s3]	secret_key	S3 API secret key (password)
[s3]	bucket	Default bucket name for storing produced resources
[redis]	host	Redis server hostname
[redis]	port	Redis server port
[redis]	db	Redis server database id (default: 0)
[redis]	username	Redis server AUTH username (default: None)
[redis]	password	Redis server AUTH password (default: None)
[redis]	socket_timeout	Socket timeout for Redis operations in seconds (default: 30, use 0 to turn off if timeout doesn't work properly)
[karton]	identity	Karton service identity override (overrides the name provided in class / constructor arguments)
[karton]	persistent	Karton service queue persistency override
[karton]	task_timeout	Karton service task execution timeout in seconds. Useful if your service sometimes hangs. Karton will schedule SIGALRM if this value is set.
[logging]	level	Logging level for Karton service logger (default: INFO)
[signaling]	status	Turns on producing of 'karton.signaling.status' tasks, signalling the task start and finish events by Karton service (default: 0, off)

5.2 Karton System configuration

Most core services can be tuned depending on your needs. Custom service configuration is handled the same way as general Karton configuration.

Good example is Karton System:

Section	Option	Description
[system]	gc_interval	Spawn interval for garbage collection tasks in seconds. Default is 3 minutes.
[system]	task_dispatched_timeout	Timeout for tasks that are stuck in DISPATCHED state (e.g. Producer crashed during upload of resources). Default is 24 hours.
[system]	task_started_timeout	Timeout for tasks that are stuck in STARTED state (e.g. non-graceful crash of Consumer during task processing). Default is 24 hours.
[system]	task_crashed_timeout	Timeout for removal of crashed tasks. Default is 3 days.
[system]	enable_gc	Enable garbage collection. GC can be turned off if you want to scale up routing using several Karton System instances.
[system]	enable_router	Enable task routing. Routing can be turned off if you want to use dedicated Karton System instance for GC.

All settings can be set using command-line.

```
$ karton-system --help
usage: karton-system [-h] [--version] [--config-file CONFIG_FILE] [--identity IDENTITY]␣
↪[--log-level LOG_LEVEL] [--setup-bucket] [--disable-gc] [--disable-router] [--gc-
↪interval GC_INTERVAL]
                     [--task-dispatched-timeout TASK_DISPATCHED_TIMEOUT] [--task-started-
↪timeout TASK_STARTED_TIMEOUT] [--task-crashed-timeout TASK_CRASHED_TIMEOUT]

Karton message broker.

options:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  --config-file CONFIG_FILE
                        Alternative configuration path
  --identity IDENTITY   Alternative identity for Karton service
  --log-level LOG_LEVEL
                        Logging level of Karton logger
  --setup-bucket        Create missing bucket in S3 storage
  --disable-gc          Do not run GC in this instance
  --disable-router      Do not run task routing in this instance
  --gc-interval GC_INTERVAL
                        Garbage collection interval
  --task-dispatched-timeout TASK_DISPATCHED_TIMEOUT
                        Timeout for non-enqueued tasks stuck in Dispatched state (non-
↪graceful shutdown of producer)
  --task-started-timeout TASK_STARTED_TIMEOUT
                        Timeout for non-enqueued tasks stuck in Started state (non-
↪graceful shutdown of consumer)
  --task-crashed-timeout TASK_CRASHED_TIMEOUT
                        Timeout for tasks in Crashed state
```

5.3 Extending configuration

During development of your own Karton services you may want to provide your own configuration fields.

All configuration values set in `karton.ini` files and `KARTON_` envs are available in `self.config` object and don't require additional definition.

The only thing that needs to be extended is argument parser if you want to use command-line arguments. Fortunately, Karton classes expose dedicated methods for this purpose.

```python
import argparse

from karton import Config, Karton, Task

class SmolKarton(Karton):
    identity = "karton.smol"
    filters = [{
        "type": "smol-tasks"
    }]

    def process(self, task: Task) -> None:
```

(continues on next page)

(continued from previous page)

```
        if self.config.has_option("smol", "how_smol")
            how_smol = self.config.getint("smol", "how_smol")
            if task.headers["size"] > how_smol:
                # Task is not smol enough UwU
                return
        ...

    @classmethod
    def args_parser(cls) -> argparse.ArgumentParser:
        # Remember to call super method to include base arguments
        parser = super().args_parser()
        parser.add_argument(
            "--how-smol",
            type=int,
            default=cls.GC_INTERVAL,
            help="Sets size limit for tasks",
        )
        return parser

    @classmethod
    def config_from_args(cls, config: Config, args: argparse.Namespace) -> None:
        # Remember to call super method to include base arguments
        super().config_from_args(config, args)
        config.load_from_dict(
            {
                "smol": {
                    "how_smol": args.how_smol,
                }
            }
        )

if __name__ == "__main__":
    SmolKarton.main()
```

`args_parser` method exposes the `argparse.ArgumentParser` that is used for handling CLI arguments. Values from argparse are then passed to `config_from_args` that maps arguments into sections and options of configuration. That mechanism allows you to define your own arguments and include these values in the final configuration.

5.4 Customizing ready-made Karton services

Ready-made Karton services like `karton-mwdb-reporter` are coming with a predefined set of filters and emitted headers. If you want to extend them or override them without forking the whole project, you can simply extend the Karton class and override things you need.

```
from karton.mwdb_reporter import MWDBReporter

class CustomMWDBReporter(MWDBReporter):
    filters = [
        *CustomMWDBReporter,
        {"type": "sample", "stage", "my-stage"}
```

(continues on next page)

(continued from previous page)

```
    ]

if __name__ == "__main__":
    CustomMWDBReporter.main()
```

Warning: It's recommended to pin to the specific version of service you derive from in case of conflicting changes.

CHAPTER

SIX

ADVANCED CONCEPTS

6.1 Routed and unrouted tasks (task forking)

During its lifetime, the task will transfer between various states and its reference will be passed through several queues, a simple way to understand it is to see how the tasks state changes in various moments:

Each new task is registered in the system by a call to `karton.Producer.send_task()` and starts its life in the **unrouted task queue** with a `TaskState.Declared` state.

All actual task data is stored in the `Karton.task` namespace and all other (routed and unrouted) queues will be always only holding a reference to a record from this place.

The main broker - `karton.System` constantly looks over the unrouted (`karton.tasks`) queue and keeps the tasks running as well as clears up leftover unneeded data.

Because task headers can be accepted by more than one consumer the task has to be forked before it goes to the appropriate **consumer (routed) queues**. Based on **unrouted task**, `Karton.System` generates as many **routed tasks** as there are matching queues. These tasks are separate, independent instances, so they have different **uid** than original unrouted task.

Note: While **uid** of routed and unrouted tasks are different, **parent_uid** stays the same. **parent_uid** always identifies the routed task.

Reference to the unrouted task is called **orig_uid**.

Each registered consumer monitors its (routed) queue and performs analysis on all tasks that appear there. As soon as the consumer starts working on a given task, it sends a signal to the broker to mark the tasks state as `TaskState.Started`.

If everything goes smoothly, the consumer finishes the tasks and sends a similar signal, this time marking the task as `TaskState.Finished`. If there is a problem and an exception is thrown within the `self.process` function, `TaskState.Crashed` is used instead.

As a part of its housekeeping, `Karton.System` removes all `TaskState.Finished` tasks immediately and `TaskState.Crashed` tasks after a certain grace period to allow for inspection and optional retry.

6.2 Task tree (analysis) and task life cycle

Every analysis starts from **initial task** spawned by `karton.Producer`. **Initial task** is consumed by consumers, which then produce next tasks for further processing. These various tasks originating from initial task can be grouped together into a **task tree**, representing the analysis.

Each task is identified by a tuple of four identifiers:

- **uid** - unique task identifier
- **parent_uid** - identifier of task that spawned current task as a result of processing
- **root_uid** - task tree identifier (analysis identifier, derived from uid of initial **unrouted** task)
- **orig_uid** - identifier of the original task that was forked to create this task (unrouted task or retried crashed task)

In order to better understand how those identifiers are inherited and passed between tasks take a look at the following example:

6.3 Handling logging

By default, all systems inheriting from `karton.core.KartonBase()` will have a custom `logging.Logger()` instance exposed as `log()`. It publishes all logged messages to a special PUBSUB key on the central Redis database.

In order to store the logs into a persistent storage like Splunk or Rsyslog you have to implement a service that will consume the log entries and send them to the final database, for an example of such service see *Log consumer*.

The logging level can be configured using the standard karton config and setting `level` in the `logging` section to appropriate level like `"DEBUG"`, `"INFO"` or `"ERROR"`.

6.4 Consumer queue persistence

Consumer queue is created on the first registration of consumer and it gets new tasks even if all consumer instances are offline. It guarantees that analysis will complete even after short downtime of part of subsystems. Unfortunately, it also blocks completion of the analysis when we connect a Karton Service which is currently developed or temporary.

We can turn off queue persistence using the `persistent = False` attribute in the Karton subsystem class.

```
class TemporaryConsumer(Karton):
    identity = "karton.temporary-consumer"
    filters = ...
    persistent = False

    def process(self, task: Task) -> None:
        ...
```

This is also the (hacky) way to remove persistent queue from the system. Just launch empty consumer with identity you want to remove, wait until all tasks will be consumed and shut down the consumer.

```
from karton.core import Karton

class DeleteThisConsumer(Karton):
    identity = "karton.identity-to-be-removed"
    filters = {}
    persistent = False

    def process(self, task: Task) -> None:
        pass

DeleteThisConsumer().loop()
```

6.5 Prioritized tasks

Karton allows to set priority for task tree: `TaskPriority.HIGH`, `TaskPriority.NORMAL` (default) or `TaskPriority.LOW`. Priority is determined by producer spawning an initial task.

```
producer = Producer()
task = Task(
    headers=...,
    priority=TaskPriority.HIGH
)
producer.send_task(task)
```

All tasks within the same task tree have the same priority, which is derived from the priority of initial task. If consumer will try to set different priority for spawned tasks, new priority settings will be simply ignored.

6.6 Extending configuration

During processing we may need to fetch data from external service or use libraries that need to be pre-configured. The simplest approach is to use separate configuration file, but this is a bit messy.

Karton configuration is represented by special object `karton.Config`, which can be explicitly provided as an argument to the Karton constructor. *Config* is based on `configparser.ConfigParser`, so we can extend it with additional sections for custom configuration.

For example, if we need to communicate with MWDB, we can make MWDB binding available via `self.config.mwdb`

```
import mwdblib

class MWDBConfig(Config):
    def __init__(self, path=None) -> None:
        super().__init__(path)
        self.mwdb_config = dict(self.config.items("mwdb"))

    def mwdb(self) -> mwdblib.MWDB:
        api_key=self.mwdb_config.get("api_key")
        api_url=self.mwdb_config.get("api_url", mwdblib.api.API_URL)

        mwdb = mwdblib.MWDB(api_key=api_key, api_url=api_url)
```

(continues on next page)

(continued from previous page)

```
        if not api_key:
            mwdb.login(
                self.mwdb_config["username"],
                self.mwdb_config["password"])
        return mwdb

class GenericUnpacker(Karton):
    ...

    def process(self, task: Task) -> None:
        file_hash = task.get_payload("file_hash")
        sample = self.config.mwdb().query_file(file_hash)

if __name__ == "__main__":
    GenericUnpacker(MWDBConfig()).loop()
```

and provide additional section in *karton.ini* file:

```
[s3]
secret_key = <redacted>
access_key = <redacted>
address = http://127.0.0.1:9000
bucket = karton

[redis]
host = 127.0.0.1
port = 6379

[mwdb]
api_url = http://127.0.0.1:5000/api
api_key = <redacted>
```

6.6.1 Karton-wide and instance-wide configuration

By default the configuration is searched in the following locations (by searching order):

- `/etc/karton/karton.ini`
- `~/.config/karton/karton.ini`
- `./karton.ini`
- environment variables

Each next level overrides and merges with the values loaded from the previous path. It means that we can provide karton-wide configuration and specialized instance-wide extended configuration specific for subsystem.

Contents of `/etc/karton/karton.ini`:

```
[s3]
secret_key = <redacted>
access_key = <redacted>
address = http://127.0.0.1:9000
bucket = karton
```

(continues on next page)

(continued from previous page)

```
[redis]
host = 127.0.0.1
port = 6379
```

and specialized configuration in the working directory `./karton.ini`

```
[mwdb]
api_url = http://127.0.0.1:5000/api
api_key = <redacted>
```

6.7 Passing tasks to the external queue

Karton can be used to delegate tasks to separate queues e.g. external sandbox. External sandboxes usually have their own concurrency and queueing mechanisms, so Karton subsystem needs to:

- dispatch task to the external service
- wait until service ends processing
- fetch results and spawn result tasks keeping the *root_uid* and *parent_uid*

We tried to solve this using asynchronous tasks but it turned out to be very hard to be implemented correctly and didn't really fit in to with the Karton model.

6.7.1 Busy waiting

The simplest way to do that is to perform all of these actions synchronously, inside the `process()` method.

```
def process(self, task: Task) -> None:
    sample = task.get_resource("sample")

    # Dispatch task, getting the analysis_id
    with sample.download_temporary_file() as f:
        analysis_id = sandbox.push_file(f)

    # Wait until analysis finish
    while sandbox.is_finished(analysis_id):
        # Check every 5 seconds
        time.sleep(5)

    # If analysis has been finished: get the results and process them
    analysis = sandbox.get_results(analysis_id)
    self.process_results(analysis)
```

CHAPTER

SEVEN

WRITING UNIT TESTS

7.1 Basic unit test

So you want to test your karton systems, that's great! The karton core actually comes with a few helper methods to make it a bit easier.

The building block of all karton tests is `karton.core.test.KartonTestCase()`. It's a nifty class that wraps around your karton system and allows you to run tasks on it without needing to create a producer. What's more important however, is that it runs without any Redis or S3 interaction and thus creates no side effects.

```python
from .math_karton import MathKarton
from karton.core import Task
from karton.core.test import KartonTestCase


class MathKartonTestCase(KartonTestCase):
    """Test a karton that accepts an array of integers in "numbers" payload and
    returns their sum in "result".
    """
    karton_class = MathKarton

    def test_addition(self) -> None:
        # prepare a fake test task that matches the production format
        task = Task({
            "type": "math-task",
        }, payload={
            "numbers": [1, 2, 3, 4],
        })

        # dry-run the fake task on the wrapped karton system
        results = self.run_task(task)

        # prepare a expected output task and check if it matches the one produced
        expected_task = Task({
            "origin": "karton.math",
            "type": "math-result"
        }, payload={
            "result": 10,
        })

        self.assertTasksEqual(results, [expected_task])
```

7.2 Testing resources

That was pretty simple, but what about testing karton systems that accept and spawn payloads containing resources?

`karton.core.test.KartonTestCase()` already takes care of them for you. Just use normal `karton.core.Resource()` like you would normally do.

```
from .reverser_karton import ReverserKarton
from karton.core import Task, Resource
from karton.core.test import KartonTestCase


class ReverserKartonTestCase(KartonTestCase):
    """
    Test a karton that expects a KartonResource in "file" key and spawns a new
    task containing that file reversed.
    """

    karton_class = ReverserKarton

    def test_reverse(self) -> None:
        # prepare input data
        input_data = b"foobarbaz"
        # create fake, mini-independent resources
        input_sample = Resource("sample.txt", input_data)
        output_sample = Resource("sample.txt", input_data[::-1])

        # prepare a fake test task that matches the production format
        task = Task({
            "type": "reverse-task",
        }, payload={
            "file": input_sample
        })

        # dry-run the fake task on the wrapped karton system
        results = self.run_task(task)

        # prepare a expected output task and check if it matches the one produced
        expected_task = Task({
            "origin": "karton.reverser",
            "type": "reverse-result"
        }, payload={
            "file": output_sample,
        })

        self.assertTasksEqual(results, [expected_task])
```

CHAPTER

EIGHT

KARTON API REFERENCE

8.1 karton.core.Producer, karton.core.Consumer

class karton.core.**Producer**(*config: Optional[karton.core.config.Config] = None*, *identity: Optional[str] = None*, *backend: Optional[karton.core.backend.KartonBackend] = None*)

Producer part of Karton. Used for dispatching initial tasks into karton.

Parameters

- **config** (karton.Config) – Karton configuration object (optional)
- **identity** (*str*) – Producer name (optional)

Usage example:

```
from karton.core import Producer

producer = Producer(identity="karton.mwdb")
task = Task(
    headers={
        "type": "sample",
        "kind": "raw"
    },
    payload={
        "sample": Resource("sample.exe", b"put content here")
    }
)
producer.send_task(task)
```

Parameters

- **config** – Karton config to use for service configuration
- **identity** – Karton producer identity
- **backend** – Karton backend to use

classmethod args_description() → str

Return short description for argument parser.

classmethod args_parser() → argparse.ArgumentParser

Return ArgumentParser for main() class method.

This method should be overridden and call super methods if you want to add more arguments.

classmethod config_from_args(*config:* karton.core.config.Config, *args: argparse.Namespace*) → None

Updates configuration with settings from arguments

This method should be overridden and call super methods if you want to add more arguments.

classmethod karton_from_args(*args: Optional[argparse.Namespace] = None*)

Returns Karton instance configured using configuration files and provided arguments

Used by `KartonServiceBase.main()` method

property log: logging.Logger

Return Logger instance for Karton service

If you want to use it in code that is outside of the Consumer class, use `logging.getLogger()`:

```
import logging
logging.getLogger("<identity>")
```

Returns `Logging.Logger()` instance

property log_handler: karton.core.logger.KartonLogHandler

Return KartonLogHandler bound to this Karton service.

Can be used to setup logging on your own by adding this handler to the chosen loggers.

send_task(*task:* karton.core.task.Task) → bool

Sends a task to the unrouted task queue. Takes care of logging. Given task will be child of task we are currently handling (if such exists).

Parameters **task** – Task object to be sent

Returns Bool indicating if the task was delivered

setup_logger(*level: Optional[Union[str, int]] = None*) → None

Setup logger for Karton service (StreamHandler and *karton.logs* handler)

Called by `Consumer.loop()`. If you want to use logger for Producer, you need to call it yourself, but remember to set the identity.

Parameters **level** – Logging level. Default is logging.INFO (unless different value is set in Karton config)

class karton.core.**Consumer**(*config: Optional[*karton.core.config.Config*] = None*, *identity: Optional[str] = None*, *backend: Optional[karton.core.backend.KartonBackend] = None*)

Base consumer class, this is the part of Karton responsible for processing incoming tasks

Parameters

- **config** – Karton config to use for service configuration
- **identity** – Karton service identity
- **backend** – Karton backend to use

add_post_hook(*callback: Callable[[*karton.core.task.Task*, Optional[Exception]], None]*, *name: Optional[str] = None*) → None

Add a function to be called after processing each task.

Parameters

- **callback** – Function of the form `callback(task, exception)` where `task` is a `karton.Task` and `exception` is an exception thrown by the `karton.Consumer.process()` function or `None`.
- **name** – Name of the post-hook

add_pre_hook(*callback: Callable[*karton.core.task.Task*, None]*, *name: Optional[str] = None*) → None

Add a function to be called before processing each task.

Parameters

- **callback** – Function of the form `callback(task)` where `task` is a `karton.Task`
- **name** – Name of the pre-hook

classmethod args_description() → str

Return short description for argument parser.

classmethod args_parser() → argparse.ArgumentParser

Return ArgumentParser for main() class method.

This method should be overridden and call super methods if you want to add more arguments.

classmethod config_from_args(*config:* karton.core.config.Config, *args: argparse.Namespace*) → None

Updates configuration with settings from arguments

This method should be overridden and call super methods if you want to add more arguments.

classmethod karton_from_args(*args: Optional[argparse.Namespace] = None*)

Returns Karton instance configured using configuration files and provided arguments

Used by `KartonServiceBase.main()` method

property log: logging.Logger

Return Logger instance for Karton service

If you want to use it in code that is outside of the Consumer class, use `logging.getLogger()`:

```
import logging
logging.getLogger("<identity>")
```

Returns `Logging.Logger()` instance

property log_handler: karton.core.logger.KartonLogHandler

Return KartonLogHandler bound to this Karton service.

Can be used to setup logging on your own by adding this handler to the chosen loggers.

main() → None

Main method invoked from CLI.

abstract process(*task:* karton.core.task.Task) → None

Task processing method.

Parameters **task** – The incoming task object

self.current_task contains task that triggered invocation of `karton.Consumer.process()` but you should only focus on the passed task object and shouldn't interact with the field directly.

setup_logger(*level: Optional[Union[str, int]] = None*) → None

Setup logger for Karton service (StreamHandler and *karton.logs* handler)

Called by `Consumer.loop()`. If you want to use logger for Producer, you need to call it yourself, but remember to set the identity.

Parameters **level** – Logging level. Default is logging.INFO (unless different value is set in Karton config)

class karton.core.**Karton**(*config: Optional[karton.core.config.Config] = None*, *identity: Optional[str] = None*, *backend: Optional[karton.core.backend.KartonBackend] = None*)

This glues together Consumer and Producer - which is the most common use case

8.2 karton.core.LogConsumer

class karton.core.**LogConsumer**(*config: Optional[karton.core.config.Config] = None*, *identity: Optional[str] = None*, *backend: Optional[karton.core.backend.KartonBackend] = None*)

Base class for log consumer subsystems.

You can consume logs from specific logger by setting a `logger_filter()` class attribute.

You can also select logs of specific level via `level()` class attribute.

Parameters

- **config** – Karton config to use for service configuration
- **identity** – Karton service identity
- **backend** – Karton backend to use

abstract process_log(*event: Dict[str, Any]*) → None

The core log handler that should be overwritten in implemented log handlers

Parameters **event** – Dictionary containing the log event data

8.3 karton.core.Resource

karton.core.resource.**Resource**

alias of *karton.core.resource.LocalResource*

class karton.core.resource.**LocalResource**(*name: str*, *content: Optional[Union[str, bytes]] = None*, *path: Optional[str] = None*, *bucket: Optional[str] = None*, *metadata: Optional[Dict[str, Any]] = None*, *uid: Optional[str] = None*, *sha256: Optional[str] = None*, *fd: Optional[IO[bytes]] = None*, *_flags: Optional[List[str]] = None*, *_close_fd: bool = False*)

Represents local resource with arbitrary binary data e.g. file contents.

Local resources will be uploaded to object hub (S3) during task dispatching.

```
# Creating resource from bytes
sample = Resource("original_name.exe", content=b"X5O!P%@AP[4\
PZX54(P^)7CC)7}$EICAR-STANDARD-ANT...")
```

(continues on next page)

(continued from previous page)

```
# Creating resource from path
sample = Resource("original_name.exe", path="sample/original_name.exe")
```

Parameters

- **name** – Name of the resource (e.g. name of file)
- **content** – Resource content
- **path** – Path of file with resource content
- **bucket** – Alternative S3 bucket for resource
- **metadata** – Resource metadata
- **uid** – Alternative S3 resource id
- **sha256** – Resource sha256 hash
- **fd** – Seekable file descriptor
- **_flags** – Resource flags
- **_close_fd** – Close file descriptor after upload (default: False)

property content: bytes

Resource content. Reads the file if the file was not read before.

Returns Content bytes

classmethod from_directory(*name: str*, *directory_path: str*, *compression: int = 8*, *in_memory: bool = False*, *bucket: Optional[str] = None*, *metadata: Optional[Dict[str, Any]] = None*, *uid: Optional[str] = None*) → *karton.core.resource.LocalResource*

Resource extension, allowing to pass whole directory as a zipped resource.

Reads all files contained in directory_path recursively and packs them into zip file.

```
# Creating zipped resource from path
dumps = LocalResource.from_directory("dumps", directory_path="dumps/")
```

Parameters

- **name** – Name of the resource (e.g. name of file)
- **directory_path** – Path of the resource directory
- **compression** – Compression level (default is zipfile.ZIP_DEFLATED)
- **in_memory** – Don't create temporary file and make in-memory zip file (default: False)
- **bucket** – Alternative S3 bucket for resource
- **metadata** – Resource metadata
- **uid** – Alternative S3 resource id

Returns *LocalResource* instance with zipped contents

property sha256: Optional[str]

Resource sha256

Returns Hexencoded resource SHA256 hash

property size: int

Resource size

Returns Resource size

property uid: str

Resource identifier (UUID)

Returns Resource identifier

class karton.core.resource.**RemoteResource**(*name: str*, *bucket: Optional[str] = None*, *metadata: Optional[Dict[str, Any]] = None*, *uid: Optional[str] = None*, *size: Optional[int] = None*, *backend: Optional[KartonBackend] = None*, *sha256: Optional[str] = None*, *_flags: Optional[List[str]] = None*)

Keeps reference to remote resource object shared between subsystems via object storage (S3)

Should never be instantiated directly by subsystem, but can be directly passed to outgoing payload.

Parameters

- **name** – Name of the resource (e.g. name of file)
- **bucket** – Alternative S3 bucket for resource
- **metadata** – Resource metadata
- **uid** – Alternative S3 resource id
- **size** – Resource size
- **backend** – `KartonBackend()` to bind to this resource
- **sha256** – Resource sha256 hash
- **_flags** – Resource flags

property content: bytes

Resource content. Performs download when resource was not loaded before.

Returns Content bytes

download() → bytes

Downloads remote resource content from object hub into memory.

```
sample = self.current_task.get_resource("sample")

# Ensure that resource will be downloaded before it will be
# passed to processing method
sample.download()

self.process_sample(sample)
```

Returns Downloaded content bytes

download_temporary_file(*suffix=None*) → Iterator[IO[bytes]]

Downloads remote resource into named temporary file.

```
sample = self.current_task.get_resource("sample")

with sample.download_temporary_file() as f:
    contents = f.read()
    path = f.name

# Temporary file is deleted after exitting the "with" scope
```

Returns ContextManager with the temporary file

download_to_file(*path: str*) → None

Downloads remote resource into file.

```
sample = self.current_task.get_resource("sample")

sample.download_to_file("sample/sample.exe")

with open("sample/sample.exe", "rb") as f:
    contents = f.read()
```

Parameters **path** – Path to download the resource into

extract_temporary() → Iterator[str]

If resource contains a Zip file, extracts files contained in Zip to the temporary directory.

Returns path of directory with extracted files. Directory is recursively deleted after leaving the context.

```
dumps = self.current_task.get_resource("dumps")

with dumps.extract_temporary() as dumps_path:
    print("Fetched dumps:", os.listdir(dumps_path))
```

By default: method downloads zip into temporary file, which is deleted after extraction. If you want to load zip into memory, call *RemoteResource.download()* first.

Returns ContextManager with the temporary directory

extract_to_directory(*path: str*) → None

If resource contains a Zip file, extracts files contained in Zip into provided path.

By default: method downloads zip into temporary file, which is deleted after extraction. If you want to load zip into memory, call *RemoteResource.download()* first.

Parameters **path** – Directory path where the resource should be unpacked

loaded() → bool

Checks whether resource is loaded into memory

Returns Flag indicating if the resource is loaded or not

property sha256: Optional[str]

Resource sha256

Returns Hexencoded resource SHA256 hash

property size: int

Resource size

Returns Resource size

property uid: str

Resource identifier (UUID)

Returns Resource identifier

unload() → None

Unloads resource object from memory

zip_file() → Iterator[zipfile.ZipFile]

If resource contains a Zip file, downloads it to the temporary file and wraps it with ZipFile object.

```
dumps = self.current_task.get_resource("dumps")

with dumps.zip_file() as zipf:
    print("Fetched dumps: ", zipf.namelist())
```

By default: method downloads zip into temporary file, which is deleted after leaving the context. If you want to load zip into memory, call *RemoteResource.download()* first.

If you want to pre-download Zip under specified path and open it using zipfile module, you need to do this manually:

```
dumps = self.current_task.get_resource("dumps")

# Download zip file
zip_path = "./dumps.zip"
dumps.download_to_file(zip_path)

zipf = zipfile.Zipfile(zip_path)
```

Returns ContextManager with zipfile

8.4 karton.core.Task

class karton.core.task.**Task**(*headers: Dict[str, Any]*, *payload: Optional[Dict[str, Any]] = None*, *payload_persistent: Optional[Dict[str, Any]] = None*, *priority: Optional[karton.core.task.TaskPriority] = None*, *parent_uid: Optional[str] = None*, *root_uid: Optional[str] = None*, *orig_uid: Optional[str] = None*, *uid: Optional[str] = None*, *error: Optional[List[str]] = None*)

Task representation with headers and resources.

Parameters

- **headers** – Routing information for other systems, this is what allows for evaluation of given system usefulness for given task. Systems filter by these.
- **payload** – Any instance of `dict` - contains resources and additional informations
- **payload_persistent** – Persistent payload set for whole task subtree, propagated from initial task

- **priority** – Priority of whole task subtree, propagated from initial task like *payload_persistent*
- **parent_uid** – Id of a routed task that has created this task by a karton with *send_task()*
- **root_uid** – Id of an unrouted task that is the root of this task's analysis tree
- **orig_uid** – Id of an unrouted (or crashed routed) task that was forked to create this task
- **uid** – This tasks unique identifier
- **error** – Traceback of a exception that happened while performing this task

add_payload(*name: str*, *content: Any*, *persistent: bool = False*) → None

Add payload to task

Parameters

- **name** – Name of the payload
- **content** – Payload to be added
- **persistent** – Flag if the payload should be persistent

add_resource(*name: str*, *resource: karton.core.resource.ResourceBase*, *persistent: bool = False*) → None

Add resource to task.

Alias for *add_payload()*

Deprecated since version 3.0.0: Use *add_payload()* instead.

Parameters

- **name** – Name of the resource
- **resource** – Resource to be added
- **persistent** – Flag if the resource should be persistent

derive_task(*headers: Dict[str, Any]*) → *karton.core.task.Task*

Creates copy of task with different headers, useful for proxying resource with added metadata.

```
class MZClassifier(Karton):
    identity = "karton.mz-classifier"
    filters = {
        "type": "sample",
        "kind": "raw"
    }

    def process(self, task: Task) -> None:
        sample = task.get_resource("sample")
        if sample.content.startswith(b"MZ"):
            self.log.info("MZ detected!")
            task = task.derive_task({
                "type": "sample",
                "kind": "exe"
            })
            self.send_task(task)
        self.log.info("Not a MZ :<")
```

Changed in version 3.0.0: Moved from static method to regular method:

`Task.derive_task(headers, task)` must be ported to `task.derive_task(headers)`

Parameters **headers** – New headers for the task

Returns Copy of task with new headers

get_payload(*name: str*, *default: Optional[Any] = None*) → Any

Get payload from task

Parameters

- **name** – name of the payload
- **default** – Value to be returned if payload is not present

Returns Payload content

get_resource(*name: str*) → karton.core.resource.ResourceBase

Get resource from task.

Ensures that payload contains an Resource object. If not - raises `TypeError`

Parameters **name** – Name of the resource to get

Returns `karton.ResourceBase` - resource with given name

has_payload(*name: str*) → bool

Checks whether payload exists

Parameters **name** – Name of the payload to be checked

Returns If tasks payload contains a value with given name

is_payload_persistent(*name: str*) → bool

Checks whether payload exists and is persistent

Parameters **name** – Name of the payload to be checked

Returns If tasks payload with given name is persistent

iterate_resources() → Iterator[karton.core.resource.ResourceBase]

Get list of resource objects bound to Task

Returns An iterator over all task resources

remove_payload(*name: str*) → None

Removes payload for the task

If payload doesn't exist or is persistent - raises KeyError

Parameters **name** – Payload name to be removed

walk_payload_bags() → Iterator[Tuple[Dict[str, Any], str, Any]]

Iterate over all payload bags and direct payloads contained in them

Generates tuples (payload_bag, key, value)

Returns An iterator over all task payload bags

walk_payload_items() → Iterator[Tuple[str, Any]]

Iterate recursively over all payload items

Generates tuples (path, value).

Returns An iterator over all task payload values

8.5 karton.core.Config

class karton.core.config.**Config**(*path: Optional[str] = None*, *check_sections: Optional[bool] = True*)

Simple config loader.

Loads configuration from paths specified below (in provided order):

- `/etc/karton/karton.ini` (global)
- `~/.config/karton/karton.ini` (user local)
- `./karton.ini` (subsystem local)
- <path> optional, additional path provided in arguments

It is also possible to pass configuration via environment variables. Any variable named KARTON_FOO_BAR is equivalent to setting 'bar' variable in section 'foo' (note the lowercase names).

Environment variables have higher precedence than those loaded from files.

Parameters

- **path** – Path to additional configuration file
- **check_sections** – Check if sections `redis` and `s3` are defined in the configuration

append_to_list(*section_name: str*, *option_name: str*, *value: Any*) → None

Appends value to a list in configuration

get(*section_name: str*, *option_name: str*, *fallback: Optional[Any] = None*) → Any

Gets value from configuration or returns `fallback` (None by default) if value was not set.

getboolean(*section_name: str*, *option_name: str*, *fallback: bool*) → bool

getboolean(*section_name: str*, *option_name: str*) → Optional[bool]

Gets value from configuration or returns `fallback` (None by default) if value was not set. Value is coerced to bool type.

See also:

https://docs.python.org/3/library/configparser.html#configparser.ConfigParser.getboolean

getint(*section_name: str*, *option_name: str*, *fallback: int*) → int

getint(*section_name: str*, *option_name: str*) → Optional[int]

Gets value from configuration or returns `fallback` (None by default) if value was not set. Value is coerced to int type.

has_option(*section_name: str*, *option_name: str*) → bool

Checks if configuration value is set

has_section(*section_name: str*) → bool

Checks if configuration section exists

load_from_dict(*data: Dict[str, Dict[str, Any]]*) → None

Updates configuration values from dictionary compatible with `ConfigParser.read_dict`. Accepts value in native type, so you don't need to convert them to string.

None values are treated like missing value and are not added.

```
{
   "section-name": {
       "option-name": "value"
   }
}
```

set(*section_name: str*, *option_name: str*, *value: Any*) → None

Sets value in configuration

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

k

INDEX

L

M

P

R

S

T

U

W

Z