MorpFramework Documentation

Release 0.4.0b2

Izhar Firdaus

Dec 02, 2022

CONTENTS

CHAPTER

ONE

MORPFW: HIGHLY PLUGGABLE PYTHON FRAMEWORK

Github: https://github.com/morpframework/morpfw

Morp is a python web development framework for developers who need a framework that provides some assistance in building applications that support distributed processing. Other web frameworks are primarily designed for standard wsgi processing, while distributed worker processing is primarily a feature that is added as a plugin, Morp is different in this sense because we try to make it a first class citizen of the framework.

Morp, built on top of Morepath, provides a highly extensible framework which supports:

- REST API framework
  - DataClass schema model definition
  - CRUD endpoints
  - Search endpoint, powered by rulez query
  - Aggregation endpoint
  - State machine / transition engine powered by pytransitions
  - Soft delete
- Modular storage engine
  - SQLAlchemy (primary)
  - Elasticsearch
- Authentication engine
  - Pluggable authentication system with JWT token and X-API-KEY header support
  - Hadoop style `user.id` GET parameter with trusted host check
  - `REMOTE_USER` header with trusted host check
- Authorization engine
  - Group & role management
- Distributed processing & task scheduling
  - Powered by celery
  - Future plan to support Streamparse
- Plugin based architecture
  - Powered by morepath, dectate and reg

CHAPTER

TWO

QUICKSTART

2.1 Bootstrapping a new project

MorpFW requires Python 3.7 or newer to run. Python 3.6 is also supported but you will need to install `dataclasses` backport into your environment.

The recommended way to install morpfw is to use buildout, skeleton that is generated using `mfw-template`. Please head to mfw-template documentation for tutorial.

2.2 Bootstrapping without `mfw-template`

If you prefer to use `virtualenv`, or other methods, you can follow these steps.

First, lets get `morpfw` installed

```
$ pip install morpfw
```

If you are using buildout, version locks files are available at `mfw_workspace` repository: https://github.com/morpframework/mfw_workspace/tree/master/versions

Lets create an `app.py`. In this example, we are creating a `SQLApp` application, which meant to use `SQLAlchemy` as its primary data source, and provides `SQLAlchemy` transaction & session management.

```
import morpfw
from morpfw.authz.pas import DefaultAuthzPolicy
from morpfw.crud import permission as crudperm
from morpfw.permission import All


class AppRoot(object):
    def __init__(self, request):
        self.request = request


class App(DefaultAuthzPolicy, morpfw.SQLApp):
    pass


@App.path(model=AppRoot, path="/")
def get_approot(request):
```

(continues on next page)

(continued from previous page)

```
    return AppRoot(request)


@App.permission_rule(model=AppRoot, permission=All)
def allow_all(identity, context, permission):
    """ Default permission rule, allow all """
    return True


@App.json(model=AppRoot)
def index(context, request):
    return {"message": "Hello World"}
```

morpfw boot up application using a `settings.yml` file, so lets create one:

```
application:
   title: My First App
   class: app:App
```

Make sure you change your working directory to where `app.py` is, and you can then start the application using

```
$ PYTHONPATH=. morpfw -s settings.yml start
```

2.3 Creating a simple resource type / CRUD model

morpfw adds a type engine with RESTful CRUD on top of morepath. To utilize it, your models will need to follow a particular convention:

- A `Collection` is created that inherits `morpfw.Collection`
- A `Model` is created that inherits `morpfw.Model`
- Both `Collection` and `Model` class have a `schema` attribute that reference to a `dataclass` based schema
- Schema must be written using `dataclass`, following convention from `inverter` project.
- A `Storage` class is implemented following the storage component API, and registered against the `Model` class.
- A named typeinfo component is registered with details of the resource type.

Following is an example boilerplate declaration of a resource type called `page`, which will hook up the necessary RESTful API CRUD views for a simple data model with `title` and `body` text.

```
import typing
from dataclasses import dataclass, field

import morpfw
import morpfw.sql
import sqlalchemy as sa


@dataclass
class PageSchema(morpfw.Schema):

```

(continues on next page)

(continued from previous page)

```
    title: typing.Optional[str] = field(default=None, metadata={"title": "Title"})
    body: typing.Optional[str] = field(default=None, metadata={"title": "Body"})


class PageCollection(morpfw.Collection):
    schema = PageSchema


class PageModel(morpfw.Model):
    schema = PageSchema


# SQLALchemy model
class Page(morpfw.sql.Base):

    __tablename__ = "test_page"

    title = sa.Column(sa.String(length=1024))
    body = sa.Column(sa.Text())


class PageStorage(morpfw.SQLStorage):
    model = PageModel
    orm_model = Page


@App.storage(model=PageModel)
def get_storage(model, request, blobstorage):
    return PageStorage(request, blobstorage=blobstorage)


@App.path(model=PageCollection, path="/pages")
def get_collection(request):
    storage = request.app.get_storage(PageModel, request)
    return PageCollection(request, storage)


@App.path(model=PageModel, path="/pages/{identifier}")
def get_model(request, identifier):
    col = get_collection(request)
    return col.get(identifier)


@App.permission_rule(model=PageCollection, permission=All)
def allow_collection_all(identity, context, permission):
    """ Default permission rule, allow all """
    return True


@App.permission_rule(model=PageModel, permission=All)
def allow_model_all(identity, context, permission):
    """ Default permission rule, allow all """
```

(continues on next page)

(continued from previous page)

```
    return True


@App.typeinfo(name="test.page", schema=PageSchema)
def get_typeinfo(request):
    return {
        "title": "Test Page",
        "description": "",
        "schema": PageSchema,
        "collection": PageCollection,
        "collection_factory": get_collection,
        "model": PageModel,
        "model_factory": get_model,
    }
```

2.3.1 Configuring Database Connection

At the moment, `morpfw.SQLStorage` requires PostgreSQL to work correctly (due to coupling to some PostgreSQL specific dialect feature). To configure the database connection URI for SQLStorage, in `settings.yml`, add in `configuration` option:

```
configuration:
   morpfw.storage.sqlstorage.dburi: 'postgresql://postgres:postgres@localhost:5432/app_db
↪'
```

If you want to use beaker for session and caching, you can also add:

```
configuration:
   ...
   morpfw.beaker.session.type: ext:database
   morpfw.beaker.session.url: 'postgresql://postgres:postgres@localhost:5432/app_cache'
   morpfw.beaker.cache.type: ext:database
   morpfw.beaker.cache.url: 'postgresql://postgres:postgres@localhost:5432/app_cache'
   ...
```

2.3.2 Initializing Database Tables

`morpfw` provide integration with Alembic for generating SQLAlchemy based migrations.

To initialize alembic directory, you can run:

```
$ morpfw migration init migrations
```

To hook up your application SQLAlchemy models for alembic scan, you will need to edit `env.py` and add following imports, and configure `target_metadata` to include SQLStorage metadata:

```
from morpfw.crud.storage.sqlstorage import Base
import app
...
# configure target_metadata
target_metadata = Base.metadata
```

As `morpfw` uses some additional sqlalchemy libraries, `script.py.mako` need to also be edited to add additional imports:

```
import sqlalchemy_utils.types
import sqlalchemy_jsonfield.jsonfield
```

Then, configure `alembic.ini` (generated together during `migration init`) to point to your database:

```
[alembic]
...
sqlalchemy.url: 'postgresql://postgres:postgres@localhost:5432/app_db'
...
```

Now you can use `morpfw migration` to generate a migration script based on defined SQLAlchemy models.

```
$ PYTHONPATH=. morpfw migration revision --autogenerate -m "initialize"
```

You can then apply the migration using:

```
$ PYTHONPATH=. morpfw migration upgrade head
```

Finally you can start you application:

```
$ PYTHONPATH=. morpfw -s settings.yml start
```

2.4 CRUD REST API

If nothing goes wrong, you should get a CRUD REST API registered at `http://localhost:5000/pages/`.

```
>>> import requests
>>> resp = requests.get('http://localhost:5000/pages')
>>> resp.json()
{...}
```

Lets create a page

```
>>> resp = requests.post('http://localhost:5000/pages/', json={
...     'body': 'hello world'
... })
>>> objid = resp.json()['data']['uuid']
>>> resp = requests.get('http://localhost:5000/pages/%s' % objid)
>>> resp.json()
{...}
```

Lets update the body text

```
>>> resp = requests.patch(
...   'http://localhost:5000/pages/%s?user.id=foo' % objid, json={
...       'body': 'foo bar baz'
... })
>>> resp = requests.get('http://localhost:5000/pages/%s' % objid)
>>> resp.json()
{...}
```

Lets do a search

```
>>> resp = requests.get('http://localhost:5000/pages/+search')
>>> resp.json()
{...}
```

Lets delete the object

```
>>> resp = requests.delete('http://localhost:5000/pages/%s' % objid)
>>> resp.status_code
200
```

2.5 Python CRUD API

Python CRUD API is handled by `Collection` and `Model` objects. The `typeinfo` registry allows name based getter to `Collection` from the ``request` object.

```
page_collection = request.get_collection('test.page')
page = page_collection.get(page_uuid)
```

For more details, please refer to the *type system* documentation.

CHAPTER

THREE

COMMAND LINE INTERFACE

CHAPTER

FOUR

FEATURES

4.1 Settings File

MorpFW settings is defined using YAML

```yaml
# listening port
server:
  listen_host: 127.0.0.1
  listen_port: 5000
  server_url: http://localhost:5000

environment:
  # environment variables to set when launching the app
  HTTP_PROXY: http://localhost:3128
  HTTPS_PROXY: http://localhost:3128

# core application configuration
application:
  # title of application
  title: My App
  # path to App class
  class: myproject.app:App
  # application object factory function, default: morpfw.main:create_app
  factory: morpfw.main:create_app


# This section defines key-value pair of config options for the app.
# Config keys are expected to be using namespacing to separate their
# purposes, and your program can make use of this section to store
# configurations.
#
# Following are some default configurations
configuration:
  # mark application as in development mode, default: true
  app.development_mode: true

  # list of additional packages to scan in string, default: []
  morpfw.scan:
    - library1
    - library2
```

(continues on next page)

(continued from previous page)

```
# sqlalchemy database URI, default: undefined
morpfw.storage.sqlstorage.dburl: 'postgresql://postgres:postgres@localhost:5432/app'

# Authentication policy, defaults to noauth
morpfw.authn.policy: morpfw.authn.noauth:AuthnPolicy
morpfw.authn.policy.settings: {}

# what would be the new user state
morpfw.user.new_user_state: active

# celery configuration
morpfw.celery:
  # celery settings variables
  broker_url: 'amqp://guest:guest@localhost:5672/'
  result_backend: 'db+postgresql://postgres@localhost:5432/morp_tests'

# network ACL
morpfw.security.allowed_nets:
  # only allow this network to access the service
  - 127.0.0.1/32

# more.jwtauth configuration
morpfw.security.jwt:
  master_secret: secret
  leeway: 10
  allow_refresh: true
  refresh_nonce-handler: morpfw.auth.pas.user.path.refresh_nonce_handler
```

4.2 Type System

MorpFW CRUD & object management revolves around the idea of resource type. A resource type represents a data model and its respective fields. Resource type definition consist of a Schema, a Collection and a Model class. Collection is very similar to the concept of database table, and model is very similar to a row. Model have a Schema which defines the columns available in the Model.

When designing your application, it helps to think and model your application around the concept of resource type model and collections because views are attached to them.

4.2.1 Schema

Resource type schema in MorpFW is defined through new python 3.7 dataclass library.

Schema in MorpFW is used for:

- data validation of JSON data on create/update REST API
- data validation on dictionary that is used to create a new instance of resource.
- generating JSON schema for publishing in REST API

When defining a schema, it is good that you inherit from `morpfw.Schema` as it defines the core metadata required for correct function of the framework.

```
import morpfw
import typing
from dataclasses import dataclass

@dataclass
class MySchema(morpfw.Schema):

    field1: typing.Optional[str] = None
    field2: typing.Optional[str] = 'hello world'
```

Due to the nature of dataclass inheritance, your field definition must include default values, and if it does not have any, you should define the field with `typing.Optional` data type with a default value of `None`

4.2.2 Model

Model is the object that is published on a MorpFW path. MorpFW base model class provides the necessary API for model manipulation such as update, delete, save and other model manipulation capabilities of MorpFW.

class morpfw.interfaces.**IModel**(*request: Request*, *collection:* ICollection, *data: dict*)

Model is a representation of a data object. It provide a common set of API which is then delegated down to the storage provider.

Model is subscriptable and you can use it like a dictionary to access stored data.

Parameters

- **request** – the request object
- **storage** – storage provider
- **data** – initial data on this model

after_blobput(*field: str*, *blob: IBlob*) → None

Triggered after BLOB is stored

after_created() → None

Triggered after resource have been created

after_updated() → None

Triggered after resource have been created

before_blobdelete(*field: str*) → None

Triggered before BLOB is deleted

If the return value is False-ish, delete will be prevented

before_blobput(*field: str*, *fileobj: BinaryIO*, *filename: str*, *mimetype: Optional[str] = None*, *size: Optional[int] = None*, *encoding: Optional[str] = None*) → None

Triggered before BLOB is stored

before_delete() → bool

Triggered before deleting resource

If the return value is False-ish, delete will be prevented

before_update(*newdata: dict*) → None

Triggered before updating resource with new values

abstract delete()

Delete model

abstract delete_blob(*field: str*)

Delete blob

abstract get_blob(*field: str*) → IBlob

Return blob

abstract json() → dict

Convert model to JSON-safe dictionary

abstract links() → list

Generate links for this model

abstract put_blob(*field: str*, *fileobj: BinaryIO*, *filename: str*, *mimetype: Optional[str] = None*, *size: Optional[int] = None*, *encoding: Optional[str] = None*) → IBlob

Receive and store blob object

abstract rulesprovider()

Return pluggable business rule adapter for this model

abstract save()

Persist model data into backend storage

abstract set_initial_state()

Initialize default statemachine state for this model

abstract statemachine()

Return PyTransition statemachine adapter for this model

abstract update(*newdata: dict*, *secure: bool*)

Update model with new data

abstract xattrprovider()

Return extended attributes provider for this model

blob_fields: List[str]

List of blob field names allowed on this model

blobstorage_field: str

Field name on the model data which will be storing blob references

data: IDataProvider

Data provider

delete_view_enabled: bool

When set to True, will enable DELETE view to delete model

hidden_fields: list

List of fields that should be hidden from output

identifier: str

url identifier for this model

linkable: bool

Set whether object is linkable or not. If an object is linkable, its json result will have links attribute

abstract property schema: Type[ISchema]

The dataclass schema which this model will be using

update_view_enabled: bool

When set to True, will enable PATCH view to update model

uuid: str

uuid of this model

4.2.3 Collection

Collection is the container for Model objects. Collection manages the single type of Model and and provide collection level Model object management API such as create, search and aggregate.

class morpfw.interfaces.**ICollection**

Collection provide an API for querying group of model from its storage

abstract aggregate(*query: Optional[dict] = None*, *group: Optional[dict] = None*, *order_by: Optional[tuple] = None*) → List[*IModel*]

Get aggregated results

: param query: Rulez based query : param group: Grouping structure : param order_by: Tuple of `(field, order)` where `order` is

`'asc'` or `'desc'`

: todo: Grouping structure need to be documented

before_create(*data: dict*) → None

Triggered before the creation of resource

abstract create(*data: dict*) → *IModel*

Create a model from data

abstract get(*identifier*) → *IModel*

Get model by url identifier key

abstract get_by_uuid(*uuid: str*) → *IModel*

Get model by uuid

abstract json() → dict

JSON-safe dictionary representing this collection

abstract links() → list

Links related to this collection

abstract search(*query: Optional[dict] = None*, *offset: int = 0*, *limit: Optional[int] = None*, *order_by: Optional[tuple] = None*, *secure: bool = False*) → List[*IModel*]

Search for models

Filtering is done through `rulez` based JSON/dict query, which defines boolean statements in JSON/dict structure.

: param query: Rulez based query : param offset: Result offset : param limit: Maximum number of result : param order_by: Tuple of `(field, order)` where `order` is

`'asc'` or `'desc'`

: param secure: When set to True, this will filter out any object which
current logged in user is not allowed to see

: todo: `order_by` need to allow multiple field ordering

4.2.4 Storage

Model and collection gets their data from a storage provider. It abstracts the interface to storage backends, allowing custom storage backends to be implemented.

class morpfw.interfaces.**IStorage**(*request: Request*, *blobstorage: Optional[IBlobStorage] = None*)

Aggregateable storage

abstract aggregate(*query: Optional[dict] = None*, *group: Optional[dict] = None*, *order_by: Union[None, list, tuple] = None*) → list

return aggregation result based on specified rulez query and group

abstract create(*data: dict*) → *IModel*

Create a model from submitted data

abstract delete(*identifier*, *model*)

delete model data

abstract get(*identifier*) → Optional[*IModel*]

return model from identifier

abstract get_by_id(*id*) → Optional[*IModel*]

return model from internal ID

abstract get_by_uuid(*uuid*) → Optional[*IModel*]

return model from uuid

abstract search(*query: Optional[dict] = None*, *offset: Optional[int] = None*, *limit: Optional[int] = None*, *order_by: Union[None, list, tuple] = None*) → Sequence[*IModel*]

return search result based on specified rulez query

abstract update(*identifier*, *data*)

update model with values from data

4.2.5 BlobStorage

Storage provider may have a BLOB storage backend implemented which will handle the management of BLOBs

class morpfw.interfaces.**IBlobStorage**

abstract delete(*uuid: str*)

Delete blob data

abstract get(*uuid: str*) → Optional[IBlob]

Return blob data

abstract put(*field: str*, *fileobj: BinaryIO*, *filename: str*, *mimetype: Optional[str] = None*, *size: Optional[int] = None*, *encoding: Optional[str] = None*, *uuid: Optional[str] = None*) → IBlob

Receive and store blob data

4.3 Type System Python API

To manipulate resource types, we provide a simple mechanism to interact with the collection and model.

Lets take for example, the following resource type definition:

```
import typing
from dataclasses import dataclass, field

import morpfw
import morpfw.sql
import sqlalchemy as sa
from morpfw.authz.pas import DefaultAuthzPolicy
from morpfw.crud import permission as crudperm
from morpfw.permission import All


class AppRoot(object):
    def __init__(self, request):
        self.request = request


class App(DefaultAuthzPolicy, morpfw.SQLApp):
    pass


@App.path(model=AppRoot, path="/")
def get_approot(request):
    return AppRoot(request)


@App.permission_rule(model=AppRoot, permission=All)
def allow_all(identity, context, permission):
    """ Default permission rule, allow all """
    return True


@App.json(model=AppRoot)
def index(context, request):
    return {"message": "Hello World"}


@dataclass
class PageSchema(morpfw.Schema):

    body: typing.Optional[str] = field(default=None, metadata={"title": "Body"})
    value: typing.Optional[int] = field(default=0, metadata={"title": "Value"})
```

(continues on next page)

(continued from previous page)

```
class PageCollection(morpfw.Collection):
    schema = PageSchema


class PageModel(morpfw.Model):
    schema = PageSchema

    blob_fields = ['attachment']

# SQLALchemy model
class Page(morpfw.sql.Base):

    __tablename__ = "test_page"

    body = sa.Column(sa.Text())
    value = sa.Collection(sa.Integer())


class PageStorage(morpfw.SQLStorage):
    model = PageModel
    orm_model = Page


@App.storage(model=PageModel)
def get_storage(model, request, blobstorage):
    return PageStorage(request, blobstorage=blobstorage)


@App.path(model=PageCollection, path="/pages")
def get_collection(request):
    storage = request.app.get_storage(PageModel, request)
    return PageCollection(request, storage)


@App.path(model=PageModel, path="/pages/{identifier}")
def get_model(request, identifier):
    col = get_collection(request)
    return col.get(identifier)


class PageStateMachine(morpfw.StateMachine):

    states = ["new", "pending", "approved"]
    transitions = [
        {"trigger": "approve", "source": ["new", "pending"], "dest": "approved"},
        {"trigger": "submit", "source": "new", "dest": "pending"},
    ]


@App.statemachine(model=PageModel)
```

(continues on next page)

(continued from previous page)

```
def get_pagemodel_statemachine(context):
    return PageStateMachine(context)


@App.permission_rule(model=PageCollection, permission=All)
def allow_collection_all(identity, context, permission):
    """ Default permission rule, allow all """
    return True


@App.permission_rule(model=PageModel, permission=All)
def allow_model_all(identity, context, permission):
    """ Default permission rule, allow all """
    return True


@App.typeinfo(name="test.page", schema=PageSchema)
def get_typeinfo(request):
    return {
        "title": "Test Page",
        "description": "",
        "schema": PageSchema,
        "collection": PageCollection,
        "collection_factory": get_collection,
        "model": PageModel,
        "model_factory": get_model,
    }
```

4.3.1 Collection API

Getting collection object

All resource types are registered in a central registry using `typeinfo` directive. This allows you to query for collection by name, and use it in your program.

```
col = request.get_collection('test.page')
```

Creating records

Records can be created through `create` method on collections.

```
page1 = col.create({'body': 'Hello world', 'value': 123})
```

`create` method by default expect a dictionary with JSON serialized values, which mean, date and time would need to be passed to the method as Avro style date or time integers. (unix timestamp in miliseconds for `datetime`, number of days from epoch for `date`)

If you already deserialized the values beforehand, and your date and time are `date` or `datetime` objects. You will need to pass `deserialize=False` to the method.

```
page1 = col.create({'body': 'Hello world', 'value': 123}, deserialize=False)
```

By default, `create` method is set to insecure mode, which mean, it will allow `state` to be set during record creation (which ideally, you should not do this, because this should be handled by statemachine), and also will allow setting values for fields which are marked with `initializable=False`. If you want to force security check, add `secure=True` to the parameter.

Getting individual record

If you know the UUID of a specific record, you can get the record using:

```
page1 = col.get(record_uuid) # record_uuid is a 32 char uuid string
```

Searching for records

To search for records, you can use the `search` method. Filtering of search results is using `rulez` JSON boolean statements, which you can refer to rulez documentation for details.

```
import rulez
pages = col.search(rulez.field('value') == 123) # returns a list of Page model
```

Aggregation query

It is also possible to aggregate through the collection API. Aggregation is done through a `group` query which uses the following structure:

```
{
   "<output_field>" : {
      "function": "<aggregation_function>",
      "field": "<field_name>"
   },
   "<output_field2>" : {
      "function": "<aggregation_function2>",
      "field": "<field_name2>"
   },
}
```

For example:

```
group = {
   'hour': {
      'function': 'hourly',
      'field': 'created'
   },
   'count': {
      'function': 'count',
      'field': 'uuid'
   }
}
results = col.aggregate(group=group)
```

Only basic aggregation is supported through this API, primarily for the purpose for presenting data for analytics. For more complex aggregation, it is suggested that you develop that without using this aggregate API.

SQLStorage aggregate functions

Aggregate functions are storage specific, and currently, only following aggregate functions are supported for `sqlstorage`:

- Dimensions
  - `year`
  - `month`
  - `day`
  - `date`
  - `hourly`
- Metrics
  - `count`
  - `sum`
  - `avg`
  - `min`
  - `max`

ElasticsearchStorage aggregate functions

Aggregate functions are storage specific, and currently, only following aggregate functions are supported for `elasticsearchstorage`:

- Dimensions
  - `year`
  - `month`
  - `day`
  - `date`
  - `interval_1m`
  - `interval_15m`
  - `interval_30m`
  - `interval_1h`
- Metrics
  - `count`
  - `sum`
  - `avg`

4.3.2 Model API

Reading data

Model is subscriptable and data can be accessed similar to a dictionary.

```
body = page1['body']
```

Updating data

Updating data on a record can be done using `update` method, which have similar API as Collection's `create` method.

```
page1.update({'body': 'new body text'})
```

Deleting record

To delete a record, you can call the `delete` method.

```
page1.delete()
```

BLOB management

As BLOBs are not stored in the main data storage, but rather in a separate `blobstorage`, manipulating BLOBs are done usine a separate API.

Saving BLOB

To save a BLOB into a model, the API would be:

```
import os
import mimetypes
file_path = '/path/to/file'

# in a view, you likely can get these information from
# the request itself
stat = os.stat(file_path)
filename = os.path.basename(file_path)
mt = mimetypes.guess_type(filename)

with open('file','b') as f:
   page1.put_blob('attachment', f,
                  filename=filename,
                  mimetype=mt[0], size=stat.st_size)
```

If you are in a view, and file is uploaded as `multipart/form-data`, you can get `mimetype` and `file` object using following example:

```
# assuming file is uploaded as ``upload`` field

@App.json(model=Page, name='upload-attachment', request_method='POST')
def view(context, request):
    upload = request.POST.get('upload')

    filename = os.path.basename(upload.filename)
    mimetype = upload.type
    fileobj = upload.file

    context.put_blob('attachment', fileobj, filename=filename, mimetype=mimetype)
    return {"status": "ok"}
```

Reading BLOB

Saved BLOBs can be read using:

```
blob = page1.get_blob('attachment')

with blob.open() as f:
    data = f.read()
```

You can also return a BLOB as a streaming response in a view

```
@App.view(model=Page, name='get-attachment')
def get_blob(context, request):
    blob = context.get_blob('attachment')
    return request.get_response(blob)
```

Deleting BLOBs

To delete BLOBs, you can use:

```
page1.delete_blob('attachment')
```

Accessing state machine

If your model have a state machine registered with it, you can get the state machine object using `statemachine` method.

```
# get state machine
sm = page1.statemachine()
# trigger ``approve`` transition
sm.approve()
```

To learn more about state machine object, you can refer to PyTransitions documentation as the state machine is built using it.

4.4 Type System REST API

For each published resource type, several endpoints are automatically made available by the framework to use. This is done through Morepath view inheritance on model/collection objects.

Lets take for example the following resource type definition:

```
import typing
from dataclasses import dataclass, field

import morpfw
import morpfw.sql
import sqlalchemy as sa
from morpfw.authz.pas import DefaultAuthzPolicy
from morpfw.crud import permission as crudperm
from morpfw.permission import All


class AppRoot(object):
    def __init__(self, request):
        self.request = request


class App(DefaultAuthzPolicy, morpfw.SQLApp):
    pass


@App.path(model=AppRoot, path="/")
def get_approot(request):
    return AppRoot(request)


@App.permission_rule(model=AppRoot, permission=All)
def allow_all(identity, context, permission):
    """ Default permission rule, allow all """
    return True


@App.json(model=AppRoot)
def index(context, request):
    return {"message": "Hello World"}


@dataclass
class PageSchema(morpfw.Schema):

    body: typing.Optional[str] = field(default=None, metadata={"title": "Body"})
    value: typing.Optional[int] = field(default=0, metadata={"title": "Value"})


class PageCollection(morpfw.Collection):
    schema = PageSchema
```

(continues on next page)

(continued from previous page)

```
class PageModel(morpfw.Model):
    schema = PageSchema

    blob_fields = ['attachment']

# SQLALchemy model
class Page(morpfw.sql.Base):

    __tablename__ = "test_page"

    body = sa.Column(sa.Text())
    value = sa.Collection(sa.Integer())


class PageStorage(morpfw.SQLStorage):
    model = PageModel
    orm_model = Page


@App.storage(model=PageModel)
def get_storage(model, request, blobstorage):
    return PageStorage(request, blobstorage=blobstorage)


@App.path(model=PageCollection, path="/pages")
def get_collection(request):
    storage = request.app.get_storage(PageModel, request)
    return PageCollection(request, storage)


@App.path(model=PageModel, path="/pages/{identifier}")
def get_model(request, identifier):
    col = get_collection(request)
    return col.get(identifier)


class PageStateMachine(morpfw.StateMachine):

    states = ["new", "pending", "approved"]
    transitions = [
        {"trigger": "approve", "source": ["new", "pending"], "dest": "approved"},
        {"trigger": "submit", "source": "new", "dest": "pending"},
    ]


@App.statemachine(model=PageModel)
def get_pagemodel_statemachine(context):
    return PageStateMachine(context)


@App.permission_rule(model=PageCollection, permission=All)
```

(continues on next page)

(continued from previous page)

```
def allow_collection_all(identity, context, permission):
    """ Default permission rule, allow all """
    return True


@App.permission_rule(model=PageModel, permission=All)
def allow_model_all(identity, context, permission):
    """ Default permission rule, allow all """
    return True


@App.typeinfo(name="test.page", schema=PageSchema)
def get_typeinfo(request):
    return {
        "title": "Test Page",
        "description": "",
        "schema": PageSchema,
        "collection": PageCollection,
        "collection_factory": get_collection,
        "model": PageModel,
        "model_factory": get_model,
    }
```

4.4.1 Collection

GET /pages

Display page collection metadata

Example Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "schema": {
        "$schema": "http://json-schema.org/draft-04/schema#",
        "type": "object",
        "properties": {
            "id": {
                "type": "integer"
            },
            "uuid": {
                "type": "string"
            },
            "creator": {
                "type": "string"
            },
            "created": {
                "type": "string",
                "format": "date-time"
            },
```

(continues on next page)

(continued from previous page)

```
            "modified": {
                "type": "string",
                "format": "date-time"
            },
            "state": {
                "type": "string"
            },
            "deleted": {
                "type": "string",
                "format": "date-time"
            },
            "blobs": {
                "type": "object"
            },
            "xattrs": {
                "type": "object"
            },
            "body": {
                "type": "string"
            },
            "value": {
                "type": "integer"
            }
        },
        "additionalProperties": true
    },
    "links": [
        {
            "rel": "create",
            "href": "http://localhost:5000/pages",
            "method": "POST"
        },
        {
            "rel": "search",
            "href": "http://localhost:5000/pages/+search"
        }
    ]
}
```

POST /pages

Create new page

Example request:

```
POST /pages/ HTTP/1.1
Content-Type: application/json

{
    "body": "Hello world"
}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "data": {
        "id": 1,
        "uuid": "ea31ebb4eb814572b2cbfc2d30fac7f2",
        "creator": "285969eefd7547d38fb3a5d06996f93e",
        "created": "2019-01-29T08:37:48.653715",
        "modified": "2019-01-29T08:37:48.653715",
        "state": null,
        "deleted": null,
        "body": "Hello world",
        "value": 0
    },
    "links": [
        {
            "rel": "self",
            "href": "http://localhost:5000/pages/ea31ebb4eb814572b2cbfc2d30fac7f2"
        },
        {
            "rel": "update",
            "href": "http://localhost:5000/pages/ea31ebb4eb814572b2cbfc2d30fac7f2",
            "method": "PATCH"
        },
        {
            "rel": "delete",
            "href": "http://localhost:5000/pages/ea31ebb4eb814572b2cbfc2d30fac7f2",
            "method": "DELETE"
        }
    ]
}
```

GET /pages/+aggregate

The aggregate API allows you to query for aggregate of fields from your resource dataset.

Query Parameters

- **group** – grouping structure
- **q** – `rulez` dsl based filter query
- **order_by** – string in `field:order` format where `order` is `asc` or `asc` and `field` is the field name.

Example request:

```
from urllib.parse import urlencode
import json
import requests


qs = urlencode({
    'group': ("count:count(uuid), year:year(created), month:month(created),"
              "day:day(created), sum:sum(value), avg:avg(value)")
```

(continues on next page)

(continued from previous page)

```
})
requests.get('/pages/+aggregate?%s' % qs)
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {
        "year": 2019,
        "month": 1,
        "day": 1,
        "sum": 45,
        "count": 11,
        "avg": 4.5
    },
    {
        "year": 2019,
        "month": 1,
        "day": 2,
        "sum": 60,
        "count": 14,
        "avg": 4.6
    }
]
```

GET /pages/+search

The search API allows you to do advanced querying on your resources using Rulez query structure.

Query Parameters

- **select** – jsonpath field selector
- **q** – `rulez` dsl based filter query
- **order_by** – string in `field:order` format where `order` is `asc` or `asc` and `field` is the field name.
- **offset** – result offset
- **limit** – result limit

> **Warning:** `select` query parameter would alter the response data structure from `{"data":{}, "links":[]}` to `["val1","val2","val3" ... ]`

Example request:

```
from urllib.parse import urlencode
import json
import requests

qs = urlencode({
    'q': 'body in ["Hello"]'
```

(continues on next page)

(continued from previous page)

```
})
requests.get('http://localhost:5000/pages/+search?%s' % qs)
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "results": [
        {"data": {}, "links": []},
        {"data": {}, "links": []},
        {"data": {}, "links": []},
        {"data": {}, "links": []}
    ],
    "q": null
}
```

4.4.2 Model

GET /page/{uuid}

Display resource data

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "data": {
        "id": 1,
        "uuid": "ea31ebb4eb814572b2cbfc2d30fac7f2",
        "creator": "285969eefd7547d38fb3a5d06996f93e",
        "created": "2019-01-29T08:37:48.653715",
        "modified": "2019-01-29T08:37:48.653715",
        "state": null,
        "deleted": null,
        "body": "Hello world",
        "value": 0
    },
    "links": [
        {
            "rel": "self",
            "href": "http://localhost:5000/pages/ea31ebb4eb814572b2cbfc2d30fac7f2"
        },
        {
            "rel": "update",
            "href": "http://localhost:5000/pages/ea31ebb4eb814572b2cbfc2d30fac7f2",
            "method": "PATCH"
        },
        {
```

(continues on next page)

(continued from previous page)

```
            "rel": "delete",
            "href": "http://localhost:5000/pages/ea31ebb4eb814572b2cbfc2d30fac7f2",
            "method": "DELETE"
        }
    ]
}
```

PATCH /page/{uuid}

Update resource data

Example request:

```
PATCH /pages/ea31ebb4eb814572b2cbfc2d30fac7f2 HTTP/1.1
Content-Type: application/json

{
    "body": "Foo bar"
}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{"status": "success"}
```

DELETE /page/{uuid}

Delete resource

Example request:

```
DELETE /pages/ea31ebb4eb814572b2cbfc2d30fac7f2 HTTP/1.1
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{"status": "success"}
```

POST /page/{uuid}/+blobs?field={blobfieldname}

Upload blob using HTTP file upload

Example request:

```
import json
import requests

requests.post('http://localhost:5000/pages/ea31ebb4eb814572b2cbfc2d30fac7f2/+blobs?
↪field=attachment',
              files={'upload': open('/path/to/file.jpg')})
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{"status": "success"}
```

`GET /page/{uuid}/+blobs?field={blobfieldname}`

Download blob

`DELETE /page/{uuid}/+blobs?field={blobfieldname}`

Delete blob

`GET /page/{uuid}/+xattr-schema`

Get JSON schema for validating extended attributes. This view is only available if your model have an extended attribute provider registered

`GET /page/{uuid}/+xattr`

Return extended attributes. This view is only available if your model have an extended attribute provider registered

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "field1": "value1",
    "field2": "value2"
}
```

`PATCH /page/{uuid}/+xattr`

Update extended attributes. This view is only available if your model have an extended attribute provider registered

Example request:

```
PATCH /pages/ea31ebb4eb814572b2cbfc2d30fac7f2/+xattr HTTP/1.1
Content-Type: application/json

{
    "field1": "value3",
    "field3": "value4"
}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{"status": "success"}
```

`POST /pages/{uuid}/+statemachine`

Apply transition. This view is only available if your model have a state machine registered.

Example request:

```
POST /pages/ea31ebb4eb814572b2cbfc2d30fac7f2 HTTP/1.1
Content-Type: application/json

{"transition": "approve"}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{"status":"success"}
```

4.5 Resource CRUD Hooks

MorpFW includes several event hooks for CRUD activities so that you can slot in your custom code without having to override the built-in views. Simply implement the methods on your respective `Collection` and `Model` classes.

`ICollection.`**`before_create`**(*data: dict*) → None

Triggered before the creation of resource

`IModel.`**`after_created`**() → None

Triggered after resource have been created

`IModel.`**`before_update`**(*newdata: dict*) → None

Triggered before updating resource with new values

`IModel.`**`after_updated`**() → None

Triggered after resource have been created

`IModel.`**`before_delete`**() → bool

Triggered before deleting resource

If the return value is False-ish, delete will be prevented

`IModel.`**`before_blobput`**(*field: str*, *fileobj: BinaryIO*, *filename: str*, *mimetype: Optional[str] = None*, *size: Optional[int] = None*, *encoding: Optional[str] = None*) → None

Triggered before BLOB is stored

`IModel.`**`after_blobput`**(*field: str*, *blob: IBlob*) → None

Triggered after BLOB is stored

`IModel.`**`before_blobdelete`**(*field: str*) → None

Triggered before BLOB is deleted

If the return value is False-ish, delete will be prevented

4.6 Views

Morp inherits its view implementation from Morepath and it is best to head to Morepath's view documentation to get better understanding on how views work in MorpFW.

Every `morpfw.Model` and `morpfw.Collection` can be attached with your custom views or you may also override the built-in views on your own application.

Todo: some basic examples on how to register your own custom views

4.7 State Machine

Resources can be registered with a state machine to manage its states. MorpFW uses PyTransitions as its default state machine engine and provides the REST API to transition the states of resources.

class morpfw.interfaces.**IStateMachine**

abstract get_triggers() → List

Returns list of available triggers

abstract property readonly_states: List

List of readonly states

abstract property state: Optional[str]

Current resource state

abstract property states: List

List of `pytransitions` states

abstract property transitions: List

List of `pytransitions` transitions

4.7.1 Registering State Machine Provider

To register a state machine provider for your resource model, simply register based on following example:

```
import typing
from dataclasses import dataclass
import morpfw
from .app import App
from .model import PageModel


class PageStateMachine(morpfw.StateMachine):

    states = ['new', 'pending', 'approved']
    transitions = [
        {'trigger': 'approve', 'source': [
            'new', 'pending'], 'dest': 'approved'},
        {'trigger': 'submit', 'source': 'new', 'dest': 'pending'}
    ]
```

(continues on next page)

(continued from previous page)

```
@App.statemachine(model=PageModel)
def get_pagemodel_statemachine(context):
    return PageStateMachine(context)
```

4.8 Extended Attribute

Resource models have the ability to be extended with extended attributes. Extended attributes helps in the situation where you are building a generic feature, but you wish to allow downstream application to add additional data fields into the generic resource model without having to redefine the storage implementation with additional fields.

For example, a generic `Event` model would probably have `title`, `start_datetime` and `end_datetime` on its model, and you have created multiple views to display the Event, such as `ical_view` and `xml_view`. Now, you are doing a project for CustomerA, which you need to add additional data fields to `Event` model, eg: `department_name`. Normally you would have to modify `EventSchema` and its respective db schema with additional fields, but with extended attributes, you can simply register a extended attribute provider for `Event` model which would store the value of `department_name`.

class morpfw.interfaces.**IXattrProvider**

abstract as_dict()

Returns dictionary representation of the data, where python objects such as `datetime` remains as python objects

abstract as_json()

Returns JSON-safe dictionary representation of the data. Any python objects are serialized into JSON-safe data type

abstract jsonschema() → dict

Returns JSON Schema for Xattr

abstract process_update(*newdata: dict*)

Validate received data and then update the extended attributes

abstract update(*newdata: dict*)

Update extended attributes

abstract property schema: Type[Any]

Schema to use for data validation

4.8.1 Registering Extended Attribute Provider

MorpFW provides a default implementation for extended attribute provider called `FieldXattrProvider` which stores extended attributes in `xattr` field of the resource.

To register an extended attribute provider for your model using `FieldXattrProvider`, add the following code:

```
import typing
from dataclasses import dataclass
from morpfw.crud.xattrprovider.fieldxattrprovider import FieldXattrProvider
import morpfw
```

(continues on next page)

(continued from previous page)

```
from .app import App
from .model import PageModel


@dataclass
class PageXattrSchema(morpfw.BaseSchema):

    field1: typing.Optional[str] = None
    field2: typing.Optional[str] = None


class PageXattrProvider(FieldXattrProvider):

    schema = PageXattrSchema
    additional_properties = False


@App.xattrprovider(model=PageModel)
def get_xattrprovider(context):
    return PageXattrProvider(context)
```

4.9 Authentication

By default, morpfw uses no authentication.

Available authentication modules are:

- `morpfw.authn.noauth:AuthnPolicy` - NOAUTH policy
- `morpfw.authn.pas:AuthnPolicy` - Pluggable Auth Service which authenticate either using JWToken or `X-API-KEY` header. Requires *Pluggable Auth Service* to be enabled as the API key and token management comes from that module.
- `morpfw.authn.useridparam:AuthnPolicy` - Default. Gets username from `user.id` parameter in `GET`. Validates remote address against `morpfw.security.allowed_nets` to only trust provded hosts
- `morpfw.authn.remoteuser:AuthnPolicy` - Gets username from `REMOTE_USER` environment variable. Validates remote address against `morpfw.security.allowed_nets` to only trust provded hosts

To change to a different authentication module, update `morpfw.authn.policy` configuration in `settings.yml`. Eg:

```
configuration:
   morpfw.authn.policy: morpfw.authn.remoteuser:AuthnPolicy
   morpfw.authn.policy.settings: {}
```

4.10 Authorization

Authorization in MorpFW make use of Morepath's `permission_rule` directive.

A practice in MorpFW is that we create assign permission rules on a separate, permission rule only application which is then made as a superclass of the current application. This manner allows us to create multiple authorization policies which can be chosen from.

Built-in authorization policies are:

- `morpfw.authz.pas:DefaultAuthzPolicy` - Default policy which rejects access to all resources except for administrator user and user-specific APIs. This policy requires the Pluggable Auth Service to be enabled in the application.

To learn more about MorpFW authorization features, we suggest heading to Morepath's security documentation.

4.11 Pluggable Auth Service

MorpFW provides a built-in user, group & apikey management module and we called it the Pluggable Auth Service (PAS). PAS by default uses SQLAlchemy backend, but it is possible to override the storage engine used for it. Auth token is handled through JWT.

PAS provides several key API endpoints such as registration, login, logout, user management, group management, and api key management.

To enable PAS, your application have to be a subclass of `morpfw.SQLApp`. `App.hook_auth_models()` method should then be called to register PAS related views.

```
import morpfw
from morpfw.authz.pas import DefaultAuthzPolicy


class App(morpfw.SQLApp, DefaultAuthzPolicy):
    pass


App.hook_auth_models(prefix="/api/v1/auth")
```

Afterwards, load the PAS authentication policy in your application

```
configuration:
  morpfw.authn.policy: morpfw.authn.pas.policy:DefaultAuthnPolicy
```

4.12 PAS REST API

4.12.1 Authentication

POST /auth/user/+login

Log into the system

Example request:

```
POST /auth/user/+login HTTP/1.1
Content-Type: application/json

{
    "username": "admin",
    "password": "password"
}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Authorization: JWT {jwtoken}

{"status":"success"}
```

`GET /auth/self/+refresh_token`

Return new token

Example request:

```
GET /auth/self/+refresh_token HTTP/1.1
Authorization: JWT {jwtoken}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Authorization: JWT {new_jwtoken}

{"status":"success"}
```

4.12.2 User Management

`POST /auth/user/+register`

Register user

Example request:

```
POST /auth/user/+register HTTP/1.1
Content-Type: application/json

{
    "username": "demouser",
    "email": "demouser@example.com",
    "password": "password",
    "password_validate": "password"
}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
```

(continues on next page)

(continued from previous page)

```
{"status":"success"}
```

`POST /auth/user/{username}/+change_password`

Change password

Example request:

```
POST /auth/user/demouser/+change_password HTTP/1.1
Content-Type: application/json

{
    "new_password": "password",
    "new_password_validate": "password"
}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{"status":"success"}
```

Note: individual user resource management api is the same as model rest api.

`GET /auth/self`

Get current logged in user. This model inherits from `UserModel` so all the views from `UserModel` is inherited.

`POST /auth/self/+change_password`

Change password

Example request:

```
POST /auth/self/+change_password HTTP/1.1
Content-Type: application/json

{
    "password": "oldpassword",
    "new_password": "password",
    "new_password_validate": "password"
}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{"status":"success"}
```

4.12.3 Group Management

POST /auth/group/{groupname}/+grant

Grant role

Example request:

```
POST /auth/group/demogroup/+grant HTTP/1.1
Content-Type: application/json

{
    "mapping": [
        {"user": {"username": "demouser"},
        "roles": ["member"]}
    ]
}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{"status":"success"}
```

POST /auth/group/{groupname}/+revoke

Revoke role

Example request:

```
POST /auth/group/demogroup/+grant HTTP/1.1
Content-Type: application/json

{
    "mapping": [
        {"user": {"username": "demouser"},
        "roles": ["member"]}
    ]
}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{"status":"success"}
```

GET /auth/group/{groupname}/+members

List members and their roles

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
```

(continues on next page)

(continued from previous page)


```
    "users": [
        {
            "username": "demouser",
            "userid": "demouser",
            "roles": ["member"],
            "links": [
                {"rel": "self",
                 "type": "GET",
                 "href": "http://localhost:5000/auth/user/demouser"}
            ]
        }
    ]
}
```


Note: individual group resource management api is the same as model rest api.

4.12.4 API Key Management

POST /auth/apikey/

Create API key for current logged in user

Example request:

```
POST /auth/apikey HTTP/1.1
Content-Type: application/json

{
    "password": "password",
    "label": "apikey label"
}
```

Example response:


```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "data": {
        "id": 1,
        "uuid": "3aed93a0844e482ca0997d20ab0a1b2a",
        "creator": "285969eefd7547d38fb3a5d06996f93e",
        "created": "2019-01-29T08:37:48.653715",
        "modified": "2019-01-29T08:37:48.653715",
        "state": null,
        "deleted": null,
        "userid": "demouser",
        "label": "apikey label",
        "apikey_identity": "cfaa53c9f583434b9a56ed7a8889f32e",
        "apikey_secret": "df2e6b0f112843bdaa8c695f7ac6603b",
    },
```


(continues on next page)

(continued from previous page)

```
    "links": [
        {
            "rel":"self",
            "href": "http://localhost:5000/auth/apikey/
↪3aed93a0844e482ca0997d20ab0a1b2a"
        },
        {
            "rel": "delete",
            "href": "http://localhost:5000/auth/apikey/
↪3aed93a0844e482ca0997d20ab0a1b2a",
            "method": "DELETE"
        }
    ]
}
```

Note: individual API key resource management api is the same as model rest api.

4.13 Distributed Worker & Scheduler

MorpFW integrates with Celery to provide support for running asynchronous & scheduled jobs. The `morpfw` command line tool provides subcomands which makes it easy to start celery worker and scheduler for your project.

In asynchronous tasks triggered through web development API, MorpFW encodes the WSGI request object and pass it to the worker so that you will get a very similar behavior to web development when working with asynchronous tasks.

In scheduled task, a minimal request object is created and passed to the scheduled task function.

4.13.1 Creating Async Task

Asynchronous task is implemented as signals which you can implement a subscriber to the signal.

To create an async task subscribing to a signal, you can use the `async_subscribe` decorator on your `App` object. . The task can then be triggered using `request.async_dispatch`.

Warning: Because request object is passed to the worker, avoid using this in pages with uploads as it involves transfering the upload to the worker.

Following is a simple example implementation

```
import time

import morpfw
import pytest

# lets setup a skeleton app


class App(morpfw.BaseApp):
```

(continues on next page)

(continued from previous page)

```
    pass


class Root(object):
    pass


@App.path(model=Root, path="")
def get_root(request):
    return Root()


# this view will dispatch the event


@App.json(model=Root)
def index(context, request):
    subs = request.async_dispatch("test_signal", obj={"data": 10})
    res = []
    for s in subs:
        res.append(s.get())
    return res


# these are the async handlers


@App.async_subscribe("test_signal")
def handler1(request_options, obj):
    # request_options contain parameters for instantiating a request
    with morpfw.request_factory(**request_options) as request:
        obj["handler"] = "handler1"
        obj["data"] += 1
        return obj


@App.async_subscribe("test_signal")
def handler2(request_options, obj):
    with morpfw.request_factory(**request_options) as request:
        obj["handler"] = "handler2"
        obj["data"] += 5
        return obj
```

4.13.2 Creating Scheduled Job

Scheduled job can be implemented with a similar API style. MorpFW exposes both the cron scheduler and periodic scheduler of Celery in an easy to use API.

Following is a simple example implementation

```
import time

import morpfw
import pytest

# lets setup a skeleton app

class App(morpfw.BaseApp):
    pass


class Root(object):
    pass


@App.path(model=Root, path='')
def get_root(request):
    return Root()


# lets hook up some scheduled job

# run this code every 5 seconds
@App.periodic(name='myproject.every-5-seconds', seconds=5)
def run_5_secs(request_options):
    print('periodic tick!')


# run this code every 1 minute, using cron style scheduling
@App.cron(name='myproject.minutely', minute='*')
def run_every_1_minute(request_options):
    print('cron tick!')
```

4.13.3 Starting Celery Worker & Celery Beat Scheduler

Worker and beat scheduler can be easily started up using:

```
$ # start worker
$ morpfw -s settings.yml worker

$ # start scheduler
$ morpfw -s settings.yml scheduler
```

4.14 Scripting API

We also made it easy to use your developed application as command line script. This is done through creating a `request` object in a script, and use that request as how you would use in a view. This API is provided so that you can easily build automation scripts using data stored in your application, without having to maintain separate mechanism to connect to data and manipulate it.

When a request is instantiated, it will also establish the necessary scaffolding and connection to databases, and when you close a request, data will be committed and connection would be closed.

To instantiate a request object, you may use the following example

```
import morpfw

settings = {
    "application": {
        "title": "My App",  # app title
        "class": "app:App",  # import path to your app
    }
}

with morpfw.request_factory(settings) as request:
    # do something here with the request
    pass
```

Settings provided to `request_factory` will inherit the default settings, so you are not required to provide all options.

CHAPTER

FIVE

ADVANCED FEATURES

5.1 Rules Provider

Rules provider is a convenient plugin API which allows you to register pluggable business rule class for your model following the pattern we have for registering other pluggable providers.

If you wish to have a particular portion of your model processing logic to be overrideable by downstream projects, you may use rules provider to provide the processing logic.

5.1.1 Using Rules Provider

```
import typing
import morpfw
from .app import App
from .model import PageModel


class PageRulesProvider(morpfw.RulesProvider):

    def calculate_value_offset(self):
        return self.context['value'] + 1


@App.rulesprovider(model=PageModel)
def get_rulesprovider(context):
    return PageRulesProvider(context)


@App.json(model=PageModel, name='get-value-offset')
def get_value_offset(context, request):
    return context.rulesprovider().calculate_value_offset()
```

5.2 Search Provider

When building application with large dataset, it is common to not use your primary storage to search for resources, but rather search through an external indexing service such as ElasticSearch.

MorpFW provides a overrideable search provider API for you to intercept the search mechanism and put your own search logic.

class morpfw.interfaces.**ISearchProvider**

abstract parse_query(*qs: str*) → dict

Parse query string from search query and convert it into rulez query dictionary. The dictionary would be passed to search method afterwards, which you can then parse into your backend search query.

abstract search(*query: Optional[dict] = None*, *offset: int = 0*, *limit: Optional[int] = None*, *order_by: Union[None, list, tuple] = None*) → List[*IModel*]

Execute search and return list of model objects

5.2.1 Overriding Search Provider

```
import typing
import morpfw
from .app import App
from .model import PageCollection


class PagesSearchProvider(morpfw.SearchProvider):

    def search(self, query=None, offset=0, limit=None, order_by=None):
        """search for resources and return list of resource model objects"""
        result = []
        # do something here
        return result


@App.searchprovider(model=PageCollection)
def get_searchprovider(context):
    return PagesSearchProvider(context)
```

5.3 Aggregate Provider

When building application with large dataset, it is common to pre-aggregate data on the data management layer and only query for aggregate from precomputed aggregate storage.

MorpFW provides a overrideable aggregate provider API for you to intercept the aggregate mechanism and put your own aggregate logic.

5.3.1 Overriding Aggregate Provider

```
import typing
import morpfw
from .app import App
from .model import PageCollection


class PagesAggregateProvider(morpfw.AggregateProvider):

    def aggregate(self, query=None, group=None, order_by=None):
        """search for resources, aggregate and return aggregate result"""
        result = []
        # do something here
        return result


@App.aggregateprovider(model=PageCollection)
def get_aggregateprovider(context):
    return PagesAggregateProvider(context)
```

5.4 Event Signal

Key events in resource management lifecycle would trigger signals which can be subscribed to. Key signals triggered by the type system includes:

- `morpfw.crud.signals.OBJECT_CREATED` - triggered after resource creation
- `morpfw.crud.signals.OBJECT_UPDATED` - triggered after resource is updated
- `morpfw.crud.signals.OBJECT_TOBEDELETED` - triggered before deletion of resource

5.4.1 Registering Signal Subscriber

To hook up a function that subscribe to a signal, you can use the `subscribe` method on your `App` object:

```
from morpfw.crud import signals
from .app import App
from .model import PageModel


@App.subscribe(model=PageModel, signal=signals.OBJECT_CREATED)
def handle_create(request, context, signal):
    print("Object created")


@App.subscribe(model=PageModel, signal=signals.OBJECT_UPDATED)
def handle_update(request, context, signal):
    print("Object updated")
```

(continues on next page)

(continued from previous page)

```
@App.subscribe(model=PageModel, signal=signals.OBJECT_TOBEDELETED)
def handle_deleted(request, context, signal):
    print("Deleting object")
    # raising exception here will prevent the deletion
```

5.4.2 Publishing Custom Event Signal

Custom event signal can be triggered using `signal_publish` method of `App`:

```
from morpfw.crud import signals
from .app import App
from .model import PageModel

MYSIGNAL = 'my_signal'


@App.view(model=PageModel, name='dispatch')
def dispatch(request, context):
    request.app.dispatcher(MYSIGNAL).dispatch(request, context)


@App.subscribe(model=PageModel, signal=MYSIGNAL)
def handle_signal(request, context, signal):
    print("hello world!")
```

CHAPTER

SIX

COMMUNITY

Our community is still in infancy, and we hangout mostly on Telegram. Come join us at MorpFW Telegram Channel if you have any questions.

6.1 Contributors

- Izhar Firdaus (primary author)
- Adi Syukri

6.2 Backers

- Abyres Enterprise Technologies

HTTP ROUTING TABLE

/auth

/page

/pages

INDEX

R

S

T

U

X