Appkernel Documentation

Release 1.0

Csaba Tamas

Sep 15, 2018

Contents

CHAPTER 1

What is Appkernel?

A super-easy microservice and API framework, which enables API development from zero to production within minutes (no kidding: literally within minutes).

It provides data serialisation, transformation, validation, security, ORM, RPC and service mash functions out of the box ([check out the roadmap for more details](docs/roadmap.md)).

The codebase is thoroughly tested under Python 3.7 (Python 2.7 support was dropped somewhere on the road).

CHAPTER 2

Read the docs :)

2.1 Features & Roadmap

The framework is supposed to cover the requirements of the Microservice Patterns documented by Chris Richardson.

2.1.1 Model features

A `Model` is a central data object, representing the domain of our business logic (eg. User, Project, Task, etc.).

- [x] validation on the data model using multiple custom validators
- [x] json serialisation support
- [x] json schema generator
- [x] value generators
- [x] value converters
- [x] wire-format marshaller
- [x] omitted fields

2.1.2 ORM features

Appkernel features a thin and beautiful Object Relational Mapping (ORM/a.k.a database access layer / repository) implementation, making access to your data a super-simple task.

- [x] basic CRUD (create/update/delete) operations
- [x] easy to use active record style queries
- [x] automatically generated prefixed database ID
- [x] index management (unique index, text index, etc.) on the database
- [x] database schema validation and schema management

- [x] builtin converters for serialising or deserialising the model to and from various other formats
- [x] audited fields (eg. automatically added created, updated, updated_by fields)
- [x] document versioning
- [x] Bulk Inserts
- [x] Atomic updates
- [ ] Optimistic locking
- [ ] Concurrency and transaction control
- [ ] Predefined Database Filters
- [ ] Projections
- [ ] Internal Resources

2.1.3 REST Service Endpoints

- [x] REST services (GET, PUT, POST, PATCH, DELETE)
- [x] HATEOAS actions on model
- [x] model metadata and json schema
- [x] URL query interface
- [x] Read-only by default
- [x] role based account management (RBAC)
- [x] basic authentication and JWT token support
- [x] customised, machine readable error messages
- [ ] OpenApi support
- [ ] File Storage
- [ ] JSONP
- [ ] graphql support
- [ ] Conditional Requests
- [ ] OAUTH
- [ ] rate limiting and circuit breaker
- [ ] API Versioning
- [ ] GeoJSON

2.1.4 Performance controls

- [ ] Data Integrity and Concurrency Control
- [ ] Resource-level Cache Control

2.1.5 Microservice Infrastructure

- [x] externalized configuration
- [ ] scheduler and background task executor
- [ ] health checks
- [ ] simplified logging
- [ ] enhanced logging for ops teams
- [ ] circuit breakers
- [ ] CQRS
- [ ] Event sourcing
- [ ] SAGA Pattern
- [ ] metrics
- [ ] service registration and discovery
- [ ] webflow a web state machine

2.2 How does it works?

2.2.1 Base Model

AppKernel is built around the concepts of Domain Driven Design. You can start the project by laying out the model (the Enitity).

Note: All the example code below was tested in Python's interactive console so that you can follow this documentation by trying out the code snippets. A pre-requisite is a working Mongodb (at least version 3.6.4 to enjoy all the features) and preferably an activated virtual-environment with the appkernel and its dependencies installed;

The Model class represents the data in our application and stands at the heart of the architecture. As a first step we define the *Properties* (fields) of our new domain model object as static class variable.

```
class User(Model):
    id = Property(str)
    name = Property(str)
    email = Property(str)
    password = Property(str)
    roles = Property(list, sub_type=str)
```

By this time we've got for free a keyword argument constructor (__init__ method), a json and dict representation of the class:

```
u = User(name='some name', email='some name')
u.password='some pass'

str(u)
'<User> {"email": "some name", "name": "some name", "password": "some pass"}'

u.dumps()
'{"email": "some name", "name": "some name", "password": "some pass"}'
```

Or in case we want a pretty printed Json we can do:

```
print(u.dumps(pretty_print=True))
{
    "email": "some name",
    "name": "some name",
    "password": "some pass"
}'
```

As a next step we can add some validation rules and a few default values, just to make life a bit easier:

```
class User(Model):
    id = Property(str)
    name = Property(str, required=True)
    email = Property(str, validators=[Email])
    password = Property(str, required=True)
    roles = Property(list, sub_type=str, default_value=['Login'])
```

And let's try to list the properties again:

```
u = User(name='some name', email='some name')
str(u)
'<User> {"email": "some name", "name": "some name"}'
u.dumps()
ValidationException: REGEXP on type str - The property email cannot be validated␣
↪against (?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:[{-!
↪#-[]-]|\[{--])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-
↪z0-9])?|\[(?:(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9]))\.){3}(?:(2(5[0-
↪5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9])|[a-z0-9-]*[a-z0-9]:(?:[{-!-ZS-]|\[{--])+)\])
```

Whoops… that's quite a big fat error message with a long regular expression :P …but wait a minute, that's desired behaviour since we didn't provided a proper e-mail address. Let's try it again:

```
u.email='user@acme.com'
u.dumps()
PropertyRequiredException: The property [password] on class [User] is required.
```

Yeah, that's expected too, since we missed the required password. Final round:

```
u.password='some pass'
print(u.dumps(pretty_print=True))
{
    "email": "user@acme.com",
    "name": "some name",
    "password": "some pass",
    "roles": [
        "Login"
    ]
}
```

Observe how the default value on the *roles* property is added automagically :)

So far so good, but what if I just want to validate the class in my business logic, without generating json? Despair not my friend, there's a handy method for it as well:

```
u.finalise_and_validate()
```

Ohh, that looks nice… but wait a minute, why it is called finalise and validate, why not just validate?

Well, because we have thought that there are few more use-cases beyond validation and default value generation. We thought it makes sense to add some finalisation methods called *generators* and *value converters*, where:

- a generator will generate some value upon the finalisation of the object (eg. custom IDs or instance creation dates)
- the converters will convert already existing values to something else (eg. passwords are hashed, date-times are converted back and forth to/from UNIX time, etc.)

Let's add a bit more magic to the mix :)

```
class User(Model):
    id = Property(str, generator=create_uuid_generator('U'))
    name = Property(str, required=True)
    email = Property(str, validators=[Email])
    password = Property(str, required=True, converter=content_hasher())
    roles = Property(list, sub_type=str, default_value=['Login'])

u = User(name='some name', email='user@acme.com', password='some pass')
print(u.dumps(pretty_print=True))
```

… generating the following output:

```
{
    "email": "user@acme.com",
    "id": "U013333e7-9f23-4e9d-80de-480505535cad",
    "name": "some name",
    "password": "$pbkdf2-sha256$20000$C0GI8f4/B2AsRah1LiWE8A
↪$2KBVlwBMtaoy1c2dhNORCETNEwssKMnYvB5NAPbkg1s",
    "roles": [
        "Login"
    ]
}
```

whoaaa.. what happened here:

- the **id** field got autogenerated and whenever we will receive a sample json we will know that describes a User model object, since the ID starts with 'U';
- more interesting is the change happened to the **password** property: it was hashed, so it is all secured :)

2.2.2 Service classes

Now that we have our beautiful data encapsulating Model classes, let's do something useful with them (such as save in the database or expose them as REST services).

Repository

We start by adding a pinch of augmentation with a few utility classes:

- extend the Repository class (or its descendants) to add ORM functionality to the model (CRUD, Schema Generation, Indexing, etc.);
- extend the Service class (or its descendants) to expose the model as a REST services (create new instances with POST, retrieve existing ones with GET or DELETE them);

Let's *restart our interactive python console* * and add a short configuration and an import section to explore the features of a Repository. According to the Domain Driven Design specification: "the Repository contains methods for retrieving domain objects such that alternative storage implementations may be easily interchanged.

```
from appkernel import Model, MongoRepository, Property, content_hasher, create_uuid_
↪generator, Email
from appkernel.configuration import config
from pymongo import MongoClient

config.mongo_database=MongoClient(host='localhost')['tutorial']

class User(Model, MongoRepository):
    id = Property(str, generator=create_uuid_generator('U'))
    name = Property(str, required=True)
    email = Property(str, validators=[Email])
    password = Property(str, required=True, converter=content_hasher())
    roles = Property(list, sub_type=str, default_value=['Login'])

u = User(name='some name', email='user@acme.com', password='some pass')
u.save()
u'U7ebc9ae7-d33c-458e-af56-d08283dcabb7'
```

It returns the ID of the saved Model object. Now let's try to return it from the repository:

```
loaded_user = User.find_by_id(u.id)
print(loaded_user)
<User> {"email": "user@acme.com", "id": "Ua727d463-26c8-4a47-9402-5683430d1bd0", "name
↪": "some name", "password": "$pbkdf2-sha256$20000$KaW0lnKuNSakdG4NQcjZOw$9Nk4RWeszS.
↪PWkNoW4slQdg7K376tsg610prUfjK3n8", "roles": ["Login"]}
```

Ok, let's try a more advanced query:

```
user_at_acme = User.where(User.email=='user@acme.com').find_one()
print(user_at_acme.dumps(pretty_print=True))
```

Giving the following output:

```
{
    "email": "user@acme.com",
    "id": "Ueeb4139a-1e35-43cd-ab69-7bc3b9104ae4",
    "name": "some name",
    "password": "$pbkdf2-sha256$20000$lrJ2jpEyhpCSUmpNaY1RSg$n13u6quqZA9FBVV.
↪oDVD6GzjcKshac.3gDDm11QfFE0",
    "roles": [
        "Login"
    ]
}
```

Getting rid of this user instance would be as simple as *user_at_acme.delete()*, however we won't do it yet, since I want to show a few more tricks.

More you can find in the Repository section of this guide;

Rest Service

Let's **restart again our python console** so we can expose the User model over REST permitting the creation and deletion from client applications. Doing so is super simple: the User class needs to extend the `Service` and we are all set.

```
from appkernel import Model, MongoRepository, Property, content_hasher, create_uuid_
↪generator, Email, Service
from pymongo import MongoClient
from flask import Flask
from appkernel import AppKernelEngine

app = Flask('demo app')
kernel = AppKernelEngine('demo app', app=app, enable_defaults=True)

class User(Model, MongoRepository, Service):
    id = Property(str, generator=create_uuid_generator('U'))
    name = Property(str, required=True)
    email = Property(str, validators=[Email])
    password = Property(str, required=True, converter=content_hasher())
    roles = Property(list, sub_type=str, default_value=['Login'])

u = User(name='some name', email='user@acme.com', password='some pass')
u.save()

kernel.register(User)
kernel.run()
```

Expected output:

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

At this moment we have a running REST service exposed on the http://127.0.0.1:5000/. Let's try out the main functions in a sjel terminal console with **curl**:

```
curl -X GET http://127.0.0.1:5000/users/
```

Provides you the following output:

```
{
  "_items": [
    {
      "_type": "User",
      "email": "user@acme.com",
      "id": "U9c6785f5-b8b1-4801-a09c-a45109af1222",
      "name": "some name",
      "password": "$pbkdf2-sha256$20000$6z2nVMq5N8b4P8eYs1aK0Q
↪$011JYdBICbRUr4YjI7QXJOkPm9X8PHLccVknwqQoQoA",
      "roles": [
        "Login"
      ]
    }
  ],
  "_links": {
    "self": {
      "href": "/users/"
    }
  }
}
```

Or one could search the database for users where the name contains the word 'some'

```
curl -X GET "http://127.0.0.1:5000/users/?name=~some"
```

Or check the Model's Json schema (which can be used for validation or user-interface generation):

```
curl -X GET http://127.0.0.1:5000/users/schema
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "additionalProperties": true,
  "properties": {
    "email": {
      "format": "email",
      "type": "string"
    },
    "id": {
      "type": "string"
    },
    "name": {
      "type": "string"
    },
    "password": {
      "type": "string"
    },
    "roles": {
      "items": {
        "type": "string"
      },
      "type": "array"
    }
  },
  "required": [
    "password",
    "name"
  ],
  "title": "User",
  "type": "object"
}
```

There's an alternative proprietary meta-data format further optimised for being used with Single Page Applications, which describes the Model in a way that is easy to be consumed by a frontend rendering logic:

```
curl -X GET http://127.0.0.1:5000/users/meta
{
  "email": {
    "label": "User.email",
    "required": false,
    "type": "str",
    "validators": [
      {
        "type": "Email"
      }
    ]
  },
  "id": {
    "label": "User.id",
    "required": false,
    "type": "str"
  },
  "name": {
    "label": "User.name",
    "required": true,
```

(continues on next page)

(continued from previous page)

```
    "type": "str"
  },
  "password": {
    "label": "User.password",
    "required": true,
    "type": "str"
  },
  "roles": {
    "default_value": [
      "Login"
    ],
    "label": "User.roles",
    "required": false,
    "sub_type": "str",
    "type": "list"
  }
}
```

How beautiful is that? There's way more to it (such as field translation, detailed support for validation rules), described in the **Service Section**.

Let's try to delete the previously sored User object (**please note:** the ID at the end of the URL will be different in your case, you need to copy paste from the previous request.)

```
curl -X DELETE "http://127.0.0.1:5000/users/U9c6785f5-b8b1-4801-a09c-a45109af1222"
{
  "_type": "ErrorMessage",
  "code": 405,
  "message": "MethodNotAllowed/The method is not allowed for the requested URL."
}
```

Hmmm, why is that happening? the reason is that we didn't explicitly defined the HTTP methods supported when we have registered the User Model and the default behaviour is to allow only 'GET' methods by default. In order to support DELETE and other methods we would need to register the model class with the series of desired methods.

```
kernel.register(User, methods=['GET', 'PUT', 'POST', 'PATCH', 'DELETE'])
```

Now we are ready to retry the deletion of the object.

```
curl -X DELETE "http://127.0.0.1:5000/users/U9c6785f5-b8b1-4801-a09c-a45109af1222"
{
  "_type": "OperationResult",
  "result": 1
}
```

The OperationResult 1 shows that the deletion was successful.

Service Hooks

Once a Model is exposed as a REST service, CRUD operations

```
class Order(Model, MongoRepository, Service):
    id = Property(str, generator=create_uuid_generator('O'))
    products = Property(list, sub_type=Product, required=True)
    order_date = Property(datetime, required=True, generator=date_now_generator)
```

(continues on next page)

(continued from previous page)

```
    @classmethod
    def before_post(cls, *args, **kwargs):
        order = kwargs['model']
        client = HttpClientServiceProxy('http://127.0.0.1:5000/')
        status_code, rsp_dict = client.reservation.post(Reservation(order_id=order.id,
↪ products=order.products))
        order.update(reservation_id=rsp_dict.get('result'))

if __name__ == '__main__':
    app_id = f"{Order.__name__} Service"
    kernel = AppKernelEngine(app_id, app=Flask(app_id), development=True)
    kernel.register(Order, methods=['GET', 'POST', 'DELETE'])
    kernel.run()
```

Now that you got the taste of **Appkernel** feel free to dig deeper an deeper using this documentation.

2.3 Installation

Warning: Work in progress section of documentation.

2.3.1 The short story

```
pip install appkernel
```

2.3.2 The long story

Create your project folder:

```
mkdir my_project && cd my_project
```

Install virtual environment (a dedicated workspace where you will install your dependency libraries and these won't enter in conflict with other projects):

```
pip install --user pipenv
virtualenv -p python3 venv
source venv/bin/activate
```

Note: depending on your development environment, you might need to use the command pip3 instead of pip;

Check the python version:

```
python --version
Python 3.7.0
```

If all went good you should have a Python 3.X version (please note: the software is tested with Pyhton 3.6 and 3.7).

Now we are ready to install Appernel and all of its dependencies:

```
pip install appkernel
```

2.4 Creating a microservice

Take the following sample as a minimalist microservice (offering CRUD operations for an Order Model). Save it into the orderservice.py file

```
from datetime import datetime
from flask import Flask
from appkernel import AppKernelEngine, Model, MongoRepository, Property, create_uuid_
↪generator, date_now_generator


class Order(Model, MongoRepository):
    id = Property(str, generator=create_uuid_generator('O'))
    products = Property(list, required=True)
    order_date = Property(datetime, required=True, generator=date_now_generator)


if __name__ == '__main__':
    app_id = f'{Order.__name__} Service'
    kernel = AppKernelEngine(app_id)
    kernel.register(Order, methods=['GET', 'POST', 'DELETE'])
    kernel.run()
```

2.4.1 Create a minimalistic configuration file

2.4.2 Create docker file

Dump the following content in a file named: order_service_docker_file.

```
FROM python:3.7-alpine

RUN apk update && apk upgrade
RUN apk add --update \
    python \
    python-dev \
    py-pip \
    build-base \
  && pip install virtualenv \
  && rm -rf /var/cache/apk/*
RUN apk --no-cache add libxml2-dev libxslt-dev libffi-dev openssl-dev␣
↪python3-dev
RUN apk --no-cache add --virtual build-dependencies
RUN pip install appkernel gevent
WORKDIR /app
COPY . /app

EXPOSE 5000
CMD ["python", "orderservice.py"]
```

The third parameter in the command section is the address of the Mongo docker image. One can check the address of his own installation with the following command:

```
docker inspect bridge |grep -A 5 mongo
```

2.4.3 Build the image

Let's build the docker image in the current service directory:

```
docker build -t order_service_image -f order_service_docker_file .
```

2.4.4 Run the image

And as a last stap we start the service

```
docker run --name orderservice -d -p 5000:5000 order_service_image
```

You can list the log file:

```
docker exec -it orderservice tail -fn 300 /order_service.log
```

Alternative status output check could be done with the following command (note: by default this won't show you anything, since appkernel is not writing to the standard output if it is set to production mode):

```
docker logs orderservice
```

Alternatively you can run the image in interactive mode

```
docker run -it --rm --name order-service order_service_image sh
```

2.4.5 Optionally you can create a config file

Just create a file under the name cfg.yml and place it next to your service initiator script:

```
appkernel:
  logging:
    file_name: myapp.log # the name of the log file
    max_size: 5048 # the maximum size of a log file
    backup_count: 5 # the max. number of log files
  server:
    address: 0.0.0.0 # the bind address
    port: 8080 # the port to expose the services
    shutdown_timeout: 10 # the time left to finish current jobs upon shutdown
    backlog: 100 # the number of connection accepted after the current threads are␣
↪busy
  mongo:
    host: localhost # the address of the mongo service
    db: appkernel # the name of the database in the mongo instance
  i18n:
    #languages: ['en','en-US' ,'de', 'de-DE']
    languages: ['en-US','de-DE'] # the supported translatio nlanguages
```

2.5 The Model Class

A child class extending the `Model` becomes a data-holder object with some out-of the box features (json schema, validation, factory methods). A Model corresponds to the *Entity* from the domain driven design concept. A Model is persisted in the database and/or sent through the wire between two or more services. A Model is also similar to the Python Data Class (will appear in 3.6) but way more powerful.

Warning: This section discusses the Model and its features in great detail. For a quick overview on the most notable features visit the *How does it works?* section if you didn't read that yet.

2.5.1 Features of a Model

- *Introduction to the Model Class*
- *Extensible Data Validation*
- *Default Values and Generators*
- *Converters*
- *Dict and Json Converters*
- *Marshallers*
- *JSON Schema*
- *Meta-data generator*

2.5.2 Introduction to the Model Class

Note: All the examples below were tested with Python's interactive console using the set of imports from below;

from datetime import datetime from appkernel import Model, MongoRepository, Property, Email, UniqueIndex, NotEmpty, Past, create_uuid_generator, date_now_generator, content_hasher

The following example showcases the most notable features of a **Model** class:

```
class User(Model, MongoRepository):
    id = Property(str, required=True, generator=create_uuid_generator('U'))
    name = Property(str, required=True, index=UniqueIndex)
    email = Property(str, validators=[Email], index=UniqueIndex)
    password = Property(str, validators=[NotEmpty],
                         converter=content_hasher(), omit=True)
    roles = Property(list, sub_type=str, default_value=['Login'])
    registration = Property(datetime, validators=[Past], generator=date_now_generator)


user = User(name='some user', email='some@acme.com', password='some pass')
user.save()
print(user.dumps(pretty_print=True))
```

It will generate the following output:

```
{
    "email": "some@acme.com",
    "id": "U943a5699-fa7c-4431-949d-3763ce92b847",
    "name": "some user",
    "registration": "2018-06-03T13:32:51.636770",
    "roles": [
        "Login"
    ]
}
```

Let's have a look on what just have happened. The defined user class can be persisted in MongoDB with the following properties:

- **database ID**: gets auto-generated upon saving the instance (the UUID generator support random value prefixing, so later will be simple to identify Model classes by their IDs);
- **name**: which is validated upon saving (*required=True*) and a unique index will be added to the Users collection (duplicate names won't be allowed);
- **email**: also a unique value, additionally will be validated against a regular expression pattern which makes sure that the value follows the format of an e-mail address (must contain '@' and '.' characters);
- **password**: will be converted to a hashed value upon saving, so we maintain proper security practices; Observe the *omit=True* parameter which will cause the exclusion of this property from the JSON (and other wire-format) representation of the Model;
- **role**: will have a default value *['Login']* upon save (or by calling the builtin method *finalise_and_validate()*) even though we have omitted to specify any role upon instance creation;
- **registration**: will take the value of the date time of the moment of persistence;

Note: Observe that the User class has now a keyword based constructor even-though we didn't defined one before.

Adding more roles to the User is also pretty straightforward:

```
user.append_to(roles=['Admin', 'Support'])
print(user.dumps(pretty_print=True))

{
    "email": "some@acme.com",
    "id": "U943a5699-fa7c-4431-949d-3763ce92b847",
    "name": "some user",
    "registration": "2018-06-03T13:32:51.636770",
    "roles": [
        "Login",
        "Admin",
        "Support"
    ]
}
```

Or let's say we've changed our mind and we would like to remove one element from the role list:

```
user.remove_from(roles='Admin')
```

You also got a nice representation function for free:

```
print(user)
<User> {"email": "some@acme.com", "enabled": true, "id": "U943a5699-fa7c-4431-949d-
↪3763ce92b847", "name": "some user", "registration": "2018-06-03T13:32:51.636770",
↪"roles": ["Login", "Support"]}
```

New properties can also be added to the class (as expected in python):

```
user.enabled=True
print(user.dumps(pretty_print=True))
{
    "email": "some@acme.com",
    "enabled": true,
    "id": "U943a5699-fa7c-4431-949d-3763ce92b847",
    "name": "some user",
    "registration": "2018-06-03T13:32:51.636770",
    "roles": [
        "Login",
        "Admin",
        "Support"
    ]
}
```

But what if we would create a User object which is not valid?

```
incomplete_user = User()
incomplete_user.finalise_and_validate()
```

Of course, it will raise the following Exception:

```
PropertyRequiredException: The property [name] on class [User] is required.
```

Do we have your attention? let's explore the details :)

Extensible Data Validation

We tried to make the boring task of validation a simple and fun experience. Therefore all properties have a builtin **required** field which - if set to True - will check the existence of a property. But in some cases this is far from enough.

For example you might want to make sure that a property value is a valid e-mail address (by using the Email validator), or make sure that the value is lower than 10 (using the Max validator). You can use none, one or more validators for one single property, or you can add your very own custom validator by extending the `Validator` base class;

Built-in validators

`NotEmpty` - checks that the property value is defined and not empty;

```
name = Property(str, validators=[NotEmpty]
```

`Regexp` - checks if the property value matches a regular expression;

```
just_numbers = Property(str, required=True, validators=[Regexp('^[0-9]+$')])
```

`Email` - a specialisation of the Regexp validator, providing a basic e-mail regexp pattern;

```
email = Property(str, validators=[Email])
```

`Min` and `Max` - the field should be numeric one and the value should be between the specified Min and Max values;

```
sequence = Property(int, validators=[Min(1), Max(100)])
```

`Past` and `Future` - the field should be a temporal one and the value should be in the past or in the future;

```
updated = Property(datetime, validators=[Past])
```

`Unique` - the field value should be unique in the collection of this Model object (it will install a unique index in the Mongo database and will cause cause a special unique property in the Json schema;

One of specific validator

Sometimes your Model requires a very special conditional validator, specific to the model, where's no need for building a generic one. In such cases it is enough to implement a method called *validate()*. Take the example of a Payment class, where the method (credit card or alternative payment method) defines the validation conditions:

```
class Payment(Model):
    method = Property(PaymentMethod, required=True)
    customer_id = Property(str, required=True, validators=[NotEmpty])
    customer_secret = Property(str, required=True, validators=[NotEmpty])

    def validate(self):
        if self.method in (PaymentMethod.MASTER, PaymentMethod.VISA):
            if len(self.customer_id) < 16 or len(self.customer_secret) < 3:
                raise ValidationException('The card number must be 16 character long␣
↪and the CVC 3.')
        elif self.method in (PaymentMethod.PAYPAL, PaymentMethod.DIRECT_DEBIT):
            if len(self.customer_id) < 22:
                raise ValidationException('The IBAN must be at least 22 character␣
↪long.')
```

Write your own custom validator

In case you would like to create a new type of validator, you just need to extend the `Validator` base class and implement the **validate** method:

```
class CustomValidator(Validator):
    def __init__(self, value):
        # initialise the extended class
        super(CustomValidator, self).__init__('CustomValidator', value)

    def validate(self, param_name, param_value):
        # implement your custom validation logic
        # below there's a simple equality logic as an example
        if self.value != param_value:
            raise ValidationException(self.type, param_value,
                                      _('The Property %(pname)s cannot be␣
↪validated against %(value)s', pname=param_name,
                                                                                ␣
↪                value=self.value))
```

Note: The validate function should not return any value but raise a `ValidationException` when the value is does not met the predefined conditions.

Note: In the example above we used the _() function from *Babel* in order to provide translation support for to the validation error message;

An alternative way could be the implementation of the *validate_objects* which receives all the fields of the object. This is useful to build conditional validators:

```
class CreditCardValidator(Validator):
def __init__(self):
    super().__init__('CreditCardValidators')

def validate_objects(self, parameter_name: str, instance_parameters: list):
    card_number = instance_parameters.get(parameter_name)
    if instance_parameters.get('payment_method') == 'VISA':
        self.__visa_luhn_check(card_number)
    else:
        self.__mastercard_luhn_check(card_number)

def __visa_luhn_check(self, card_number):
    ...

def __mastercard_luhn_check(self, card_number):
    ...
```

Default Values and Generators

Sometimes field values can be automatically generated upon persisting the model object (eg. a database ID or date values related to the creation or current used id in case of need for auditing function) or sensible defaults can be provided in design time (eg. the role 'Login' might be safely added to all users); Take the following example:

```
id = Property(str, required=True, generator=create_uuid_generator('U'))
```

In this case the id property will take a generated value upon saving (or running the *finalise_and_validate()* method on the model) if another value is not provided already; Writing custom generators is easy: any global function with a return value would suffice. In case the generator requires an input argument (like the create_uuid_generator in our case), one would create a method which returns another method:

```
def uuid_generator(prefix=None):
    def generate_id():
        return '{}{}'.format(prefix, str(uuid.uuid4()))

return generate_id
```

This type of ID generator enables you to prefix the IDs of your different Models, making easier the job of the support teams: one will know immediately know in which collection to sarch for even if he only has an ID (given that the User model ID is prefixed with 'U' and the Customer Model ID is prefixed with 'CT';

Built-in generators

UUID Generator: generates a globally unique id. In case a prefix parameter is provided it will be added in-front of the result

```
id = Property(str, generator=create_uuid_generator('U'))
```

Date generator: generate the date-time value of the finalisation moment:

```
registration = Property(datetime, generator=date_now_generator)
```

Current user generator: used to add the authenticated user, useful to automatically register ownership on data object or audit activities.

```
owner = Property(datetime, generator=current_user_generator)
```

Converters

It is also needed to change already existing field values in way or another. Think about the following use-cases:

- passwords need to be hashed before saving it into the database;
- dates could be converted to and from UNIX time before saving or sending it over the wire so one needs to deal less with the data format;
- some sensitive data fragments (such as GDPR controlled private data) might be encrypted/hashed upon saving as well;

Therefore any function which returns a function with the property value as input parameter can be used as a converter. In case the converter works only in one direction (like the password hasher), None can be returned as the second method. Here's the code of a hasher which an be used to secure passwords:

```
def content_hasher(rounds=20000, salt_size=16):
    def hash_content(content):
        # type: (str) -> str
        if content.startswith('$pbkdf2-sha256'):
            return content
        else:
            return pbkdf2_sha256.encrypt(content, rounds=rounds, salt_size=salt_size)

return hash_content
```

2.5.3 Dict and Json Converters

All Models can be easily converted back and forth to and from dict or JSON representation (a.k.a wireformat). Writing JSON is as simple as:

```
user.dumps()
```

The dumps method takes 2 optional parameter:

- *validate* is set to True by default (it will check the class parameters against the validators and the required parameter;
- *pretty_print* is set to False by default (one would need to set it explicitly to True one nice indented JSON output is favoured;

Let's try it out:

```
print(user.dumps(pretty_print=True))
{
    "email": "some@acme.com",
    "id": "Uf112dc8a-d75e-405c-ba8f-c15d1bf438f9",
    "name": "some user",
    "registration": "2018-06-03T17:39:54.125991",
    "roles": [
        "Login"
    ]
}
```

Observe that the password property is missing from the JSON output however the the instance contains a hashed password. That is happening due to the fact that we set the password field to *omit=True*, which means that it will be excluded from all string representations.

```
password = Property(str, converter=content_hasher(), omit=True)
```

What if we want to use a *dict* or any different format as output. In such cases comes handy the static method:

```
def to_dict(instance, convert_id=False, validate=True, skip_omitted_fields=False)
```

And can be used in the following way:

```
User.to_dict(user)
```

In case one wants to prepare some low level MongoDB persistence and we want to convert any property name **id** to **_id** as Mongo expects it. Im such cases the *convert_id=True* parameter come handy.

Of course the opposite would work by using:

```
User.from_dict(some_dict_object)
```

One can use the **set_unmanaged_parameters=False** if values from the dict which do not belong to the Model should be ignored.

Marshallers

Sometimes it is required to maintain different format on the instance and on the wire. An example is when the datetime instance is converted in unix timestampt in order to avoid possible complications due to date format conversions. Marshaller comes handy in such cases.

Timestamp marshaller

In the example below the *last_login* property of `datetime` is converted to unix timestamp of type `float` when generating JSON or upon saving it in the database. When converting JSON back (or loading from the repository) the timestamp will be converted back to `datetime`.

```
class User(Model, MongoRepository):
    last_login = Property(datetime, marshaller=TimestampMarshaller)
```

Date to datetime marshaller

Mongo will throw an exception while trying to save documents (Model instances) wu=ith properties of type date, while this is not supported by Mongo's internal BSON type. In such cases you have two options: either refrain from the use of `date` or use the built-in `MongoDateTimeMarshaller`, which will automatically convert the date to datetime before saving in the database and convert it back to date upon loading:

```
class Application(Model, MongoRepository):
    id = Property(str, required=True, generator=create_uuid_generator())
    application_date = Property(date, required=True,␣
↪marshaller=MongoDateTimeMarshaller)
```

Writing your own mashaller

Writing your own marshaller is as simple as extending the builtin `Marshaller` class and implement it's two method to convert to and from wire-format.

```
class MongoDateTimeMarshaller(Marshaller):
    def to_wireformat(self, instance_value):
        # the instance value is provided and the method should return the one to be␣
↪sent over the wire (JSON or BSON)
        ...

    def from_wire_format(self, wire_value):
        # the value received from the wire and to be converted to the format expected␣
↪by the Model instance
        ...
```

2.5.4 JSON Schema

So now we would want to validate objects when they are received on the wire or we would like to use it for validation in Mongo. Simple as that:

```
User.get_json_schema()
```

In case you would like not to allow more properties on the wire than the ones already defined on the class you can set the **additional_properties=False** which will remove the **'additionalProperties':True,** from the schema, does not allow any json document which contains more properties than the saved ones

In case you would like to use the schema as source of document validation in MongoDB, you would need to use **mongo_compatibility=True**, because the way Mongo handles dates and several other objects on the scope.

2.5.5 Meta-data generator

The JSON schema is a great standard format, however sometimes is harder to parse and it is fairly limited in features when it comes to generate user interfaces from the schema definition on the fly. Therefore we've built a proprietary format which is thought to be easy to be parsed by Javascript.

```
print(json.dumps(User.get_parameter_spec(), indent=4))
{
        "name": {
        "required": true,
```

(continues on next page)

(continued from previous page)

```
            "type": "str",
            "label": "User.name"
        },
        "roles": {
            "default_value": [
                "Login"
            ],
            "required": false,
            "type": "list",
            "sub_type": "str",
            "label": "User.roles"
        },
        "email": {
            "validators": [
                {
                    "type": "Email"
                }
            ],
            "required": false,
            "type": "str",
            "label": "User.email"
        },
        "registration": {
            "validators": [
                {
                    "type": "Past"
                }
            ],
            "required": false,
            "type": "datetime",
            "label": "User.registration"
        },
        "password": {
            "validators": [
                {
                    "type": "NotEmpty"
                }
            ],
            "required": false,
            "type": "str",
            "label": "User.password"
        },
        "id": {
            "required": true,
            "type": "str",
            "label": "User.id"
        }
}
```

2.6 Repositories

The design of the repository API is influenced by peewee, a nice and small python framework focusing on relational databases (sqlite, MySQL, PostgreSQL). The major difference between peewee and the built-in **Appkernel** ORM is that the later is optimised (and till this time) implemented only for MongoDB. However, it is possible to create your own implementation for SQL or any other database.

2.6.1 Basic CRUD (Created, Update, Delete) operations

Note: You can follow all the examples in the Python's interactive interpreter using the imports and the configuration snippet from below.

The following example is only required for the interactive interpreter or for unit tests. In this case we will use the MongoDB instance accessible on the **localhost** and will create a database called **tutorial**.

```
from appkernel import AppKernelEngine, Model, MongoRepository, Property, content_
↪hasher, create_uuid_generator, Email, AuditableRepository, NotEmpty, date_now_
↪generator, Past
from appkernel.configuration import config
from pymongo import MongoClient
from enum import Enum
from datetime import datetime, date, timedelta
from flask import Flask


config.mongo_database=MongoClient(host='localhost')['tutorial']
```

For use in development or production you can choose between the following 2 options for configuration :

- use the built-in *Default configuration*, where the Mongo database must be available on *localhost* and the database name will be **app**;
- or use the built-in *File based configuration* management to provide more fine grained configuration;

Default configuration

Once the `AppKernelEngine` is initialised with no specific configuration and the **enable_defaults** parameter set to *True*, sensible defaults are used (localhost and **app** as database).

```
app_id='demo'
app = Flask(app_id)
kernel = AppKernelEngine(app_id, app=app, enable_defaults=True)
```

File based configuration

Upon initialisation **Appkernel** looks for a file *../cfg.yml*, where the following parameters define a specific database connection:

```
appkernel:
  mongo:
    host: localhost
    db: appkernel
```

The **host** variable may contain the user and password parameters using the *mongodb://* url schema.

Building a base model structure

Let's create a simple project management app with some tasks in it:

```
class Priority(Enum):
    HIGH = 1
    MEDIUM = 2
    LOW = 3

class Task(Model, MongoRepository):
    name = Property(str, required=True, validators=[NotEmpty])
    description = Property(str, validators=[NotEmpty])
    completed = Property(bool, required=True, default_value=False)
    created = Property(datetime, required=True, generator=date_now_generator)
    closed_date = Property(datetime, validators=[Past])
    priority = Property(Priority, required=True, default_value=Priority.MEDIUM)

    def complete(self):
        self.completed = True
        self.closed_date = datetime.now()

class Project(Model, AuditableRepository):
    id = Property(str)
    name = Property(str, required=True, validators=[NotEmpty()])
    tasks = Property(list, sub_type=Task)
    created = Property(datetime, required=True, generator=date_now_generator)
```

Saving and updating

Now we are ready to define our first **Project** with some **Tasks** in it:

```
project = Project(name='some test project')
project.append_to(tasks=Task(name='finish the documentation', priority=Priority.HIGH))
# or if you like one-liners, you can add multiple tasks at once
project.append_to(tasks=[Task(name='finish all todos'), Task(name='complete the unit␣
↪tests')])

project.save()
print(project.dumps(pretty_print=True))
```

And the output looks sleek:

```
{
    "id": "OBJ_5b142be00df7a9647023f0b1",
    "created": "2018-06-03T19:54:06.830307",
    "name": "some test project",
    "tasks": [
        {
```

(continues on next page)

(continued from previous page)

```
            "completed": false,
            "created": "2018-06-03T19:53:38.149125",
            "name": "finish the documentation",
            "priority": "MEDIUM"
        },
        {
            "completed": false,
            "created": "2018-06-03T19:53:51.041349",
            "name": "finish all todos",
            "priority": "MEDIUM"
        },
        {
            "completed": false,
            "created": "2018-06-03T19:53:51.041380",
            "name": "complete the unit tests",
            "priority": "MEDIUM"
        }
    ]
}
```

Now let's complete the first task:

```
project.tasks[0].complete()
project.save()
ObjectId('5b1ee7050df7a9087e0e8952')
print(project.dumps(pretty_print=True))
```

Observe the property **completed** which now is set to True and the **closed_date** having the value of the invocation date of the **complete()** method:

```
{
    "created": "2018-06-11T23:17:57.050000",
    "id": "OBJ_5b1ee7050df7a9087e0e8952",
    "inserted": "2018-06-11T23:17:57.050000",
    "name": "some test project",
    "tasks": [
        {
            "closed_date": "2018-06-11T23:19:39.345000",
            "completed": true,
            "created": "2018-06-11T23:17:57.050000",
            "name": "finish the documentation",
            "priority": "HIGH"
        },
        {
            "completed": false,
            "created": "2018-06-11T23:17:57.050000",
            "name": "finish all todos",
            "priority": "MEDIUM"
        },
        {
            "completed": false,
            "created": "2018-06-11T23:17:57.050000",
            "name": "complete the unit tests",
            "priority": "MEDIUM"
        }
    ],
```

(continues on next page)

(continued from previous page)

```
    "updated": "2018-06-11T23:19:46.428000",
    "version": 2
}
```

Advanced Functionality

Appkernel allows you to perform atomic updates. Let's suppose we need to update some counters. The naive approach would be to write something like this:

```
for stock in Stock.find((Stock.product.code == 'BTX') & (Stock.product.size ==␣
↪ProductSize.L)):
if stock.avaialable > 0:
    stock.avaialable = stock.avaialable - 1
    stock.reserved = stock.reserved + 1
    stock.save()
else:
    raise ReservationException('Not enough products on stock.')
```

Do not do this! Not only is this is slow, but it is also vulnerable to race conditions if multiple processes are updating the available and reserved counters at the same time. Instead, you can update the counters atomically using update():

```
query = Stock.where((Stock.product.code == 'BTX') & (Stock.product.size ==␣
↪ProductSize.L))
res = query.update(available=Stock.available - quantity, reserved=Stock.reserved +␣
↪quantity)
if res == 0:
    raise ReservationException(
        f"There's no stock available for code: BTX and size: L.")
elif res > 1:
    raise ReservationException(f"Multiple product items were reserved ({res}).")
```

You can make these update statements as complex as you like.

Auditable Repository

You might have observed that there are a few extra fields, which we didn't defined on the model explicitly. This is happening due to the **AuditableRepository** class we've used in the very beginning. This will bring a few additional features to the mix:

- *inserted*: the date and time when the object was inserted to the database;
- *updated*: the date and time when the object was updated for the last time;
- *version*: the number of updates on this class;

Of course we could have stayed with the simpler `MongoRepository` in case we are not in need of the extra magic for auditing our data model.

Delete objects

We can check the number of projects quickly:

```
Project.count()
1
```

Once we don't need the project anymore we can issue the **delete** command:

```
project.delete()
1
```

You can delete all projects at once:

```
Project.delete_all()
```

Querying data

Appkernel provides a simple abstraction over the native MongoDB queries, simplifying your job for most of the queries. The query expressions can be provided as parameter to the:

- **find** method: returns a generator, which can be used to iterate over the result set;
- **find_one** method: returns the first hit or None, if nothing matches the query criteria;
- **where** method: returns the `Query` object, which allows the chaining of further expressions, such as **sort**;

A simple example:

```
prj = Project.find_one(Project.name == 'some test project')
print(prj.dumps(pretty_print=True))
```

Or use property name chaining for searching all project which contain the word 'finish' in their task description:

```
prj = Project.find_one(Project.tasks.name % 'finish')
print(prj.dumps(pretty_print=True))
```

An alternative way to achieve the same target:

```
prj2 = Project.find_one(Project.tasks[Task.name == 'finish the documentation'])
```

Or you can iterate through all occurrences…

```
for project in Project.find():
    print(project)
```

Or iterate through the ones which fit a query condition:

```
for prj in Project.find(Project.name == 'some test project'):
    print(prj.dumps(pretty_print=True))
```

… and sort the result in a particular order:

```
query = Project.where(Project.name == 'some test project').sort_by(Project.created.
↪asc())
for prj in query.find():
    print(prj.dumps(pretty_print=True))
```

Adding multiple expressions to the query is also straightforward:

```
yesterday = datetime.combine(date(2018, 6, 10), datetime.min.time())
today = datetime.combine(date(2018, 6, 11), datetime.min.time())
prj = Project.find_one((Project.created > yesterday) & (Project.created < today))
print(prj.dumps(pretty_print=True))
```

Pagination

Sometimes it is a good approach to define a range (a page) which is gonna be queried, in this way we avoid filling up the memory with a huge result set. The following query will return the first 10 Projects from the database:

```
for prj in Project.find(page=0, page_size=10):
    print(prj)
```

Query expressions

Find by ID

Find a project knowing its exact id:

```
prj = Project.find_by_id('5b1ee9930df7a9087e0e8953')
```

Exact match

Returns where the field *name* exactly matches: *'Project A'*:

```
prj = Project.find_one((User.name == 'Project A'))
```

Not equal

Return all projects **except** *'Project A'*:

```
prj = Project.find_one((User.name != 'Project A'))
```

Or

Returns *'Project A'* or *'Project B'*:

```
prj = Project.find_one((Project.name == 'Project A') | (Project.name == 'Project B'))
```

And

Returns every project named *'Project A'* created after yesterday:

```
yesterday = (datetime.now() - timedelta(days=1))
prj = Project.find_one((Project.name == 'Project A') & (Project.created > yesterday))
```

Empty Array

Find all Projects with no tasks:

```
prj = Project.find_one(Project.tasks == None)
```

Contains

Find all projects which has at least one task containing the string 'finish':

```
prj = Project.find_one(Project.tasks.name % 'finish')
```

Also you can query for values in an array. The following query will return all users, who are having the Role **Admin** and **Operator**:

```
User.find(User.roles % ['Admin', 'Operator'])
```

Does not exists

Return all users which have no defined **description** field:

```
User.find(User.description == None)
```

Value exists (not None)

Return all users which has description field:

```
User.find(User.description != None)
```

Smaller and bigger

Return all projects created between a well defined period of time:

```
yesterday = (datetime.now() - timedelta(days=1))
tomorrow = (datetime.now() + timedelta(days=1))
user_iterator = Project.find((User.created > yesterday) & (User.created < tomorrow))
```

Query with custom properties

Sometimes the object model does not contains a property but the field is available in the database. Think about the AuditableRepository which automatically creates extra fields such as object version. In case we'd like to search all documents with version 2, the **custom property** comes handy:

```
project = Project.find_one(Project.custom_property('version') == 2)
```

Native Queries

Appkernel's built-in ORM tries to cover the common use-cases and it will be further developed in the future, however in case there's a need for special and very complex query, we might want to fallback to MongoDB's native query.

```
project.counter=5
project.save()
for p in Project.find_by_query({'counter': {'$gte': 0, '$lt': 10}}):
    print 'Project name: {} and counter: {}'.format(p.name, p.counter)
```

Alternatively you can also access a reference to a PyMongo `Collection` object via the `Model`'s **get_collection** method.

```
mongo_document = Project.get_collection().find_one(filter)
```

For more details on what can you do via the collection reference, please consult the **pymongo** documentation.

Bulk insert

Sometimes you're in need to insert (upsert) multiple objects at once:

```
def create_user_batch(urange=51):
users = []
for i in range(1, urange):
    users.append(User().update(name='multi_user_{}'.format(i)).update(password='some␣
↪default password'). \
        append_to(roles=['Admin', 'User', 'Operator']).update(description='some␣
↪description').update(
        sequence=i))
return users
ids = User.bulk_insert(create_user_batch()
```

2.6.2 Index management

In order to speed up lookup for certain fields, you might want to add indexes to certain properties. This can be easily achieved by using the **index** parameter of the `Property` class. Let's redefine the **Project** class:

```
class Project(Model, AuditableRepository):
    ...
    name = Property(str, required=True, validators=[NotEmpty()], index=UniqueIndex)
    created = Property(datetime, required=True, generator=date_now_generator,␣
↪index=Index)
    ...

User.init_indexes()
```

Mind the *index=UniqueIndex* on the **name** property and the *index=Index* on the **created** property. The idea behind the Unique Index is to avoid accidental project name duplication, while the normal Index on the created field will speed up the search and sorting by created date.

Built-in Indexes

- **Index**: used to speed up queries (also will slow insertion, so use it with care);
- **UniqueIndex**: will make sure that the value exists only once in the database;
- **TextIndex**: can be used all string fields and helps with full-text search;

For more information on indexes, please have look on Mongo's documentation;

2.6.3 Schema Installation

MongoDB started its life as a schema less database, however the advantages of applying a schema on a database was soon recognized by the Mongo folks. Data integrity is assured by enforcing validation on inserts and udpates.

MongoDB now supports a subset of JSON Schema which can be used to validate field against type information or matching a regular expression or set of Enum values. The Mongo Specific JSON schema can be generated by Appkernel's `Model` and installed by the childs of `MongoRepository`.

```
Project.add_schema_validation(validation_action='error')
```

The validation_action can take the value:

- *error* - in case an object is not valid, the insertion will be rejected;
- *warning* - in case of a schema validation error, only a log-line is registered in MongoDB;

2.6.4 Supported Repository Types

All repositories are extending the `Repository` base class. This class serves as an Interface (so a sort of an implementation guideline, since the Interface concept is not supported by Python) for all other repository implementations.

- `MongoRepository` - standard repository functionality providing access to MongoDB;
- `AuditableRepository` - an extended repository, which will save the user, document creation date and some other, useful metadata information;

2.6.5 Advanced Functionality

Accessing the native **pymongo** `collection` class opens a lot of new opportunities.

Dropping the collection

Will drop the complete collection:

```
User.get_collection().drop()
```

Check index information

The index information can be retrieved:

```
idx_info = User.get_collection().index_information()
```

... or alternatively:

```
config.mongo_database['Users'].index_information()
```

Aggregation Pipeline

Mongo features a very powerful map-reduce tool called Aggregation Pipeline, very useful for complicated queries:

```
pipeline = [{'$match': ...}, {'$group': ...}]
Project.get_collection().aggregate(pipeline)
```

2.7 Services

2.7.1 REST endpoints over HTTP

Exposing your models over HTTP/REST is easy and *Custom resource endpoints* are supported as well.

Let's assume that we have created a User class extending the `Model`. Now we'd like to expose it as a REST endpoint:

```python
class User(Model, MongoRepository):
    ...

if __name__ == '__main__':
    app = Flask(__name__)
    kernel = AppKernelEngine('demo app', app=app)
    kernel.register(User)
    kernel.run()
```

The *register* method from the above example will expose the *User* entity at *http://localhost/users* with the GET method supported by default. In case we would like to add support for the rest of the HTTP methods (pun intended), we would need to explicitly specify them in the *register* method (for more details check out the *Full range of CRUD operations* section).

```python
kernel.register(User methods=['GET', 'PUT', 'POST', 'PATCH', 'DELETE'])
```

Securing service access is also no-brainer:

```python
kernel.enable_security()
user_service = kernel.register(User methods=['GET', 'PUT', 'POST', 'PATCH', 'DELETE'])
user_service.deny_all().require(Role('user'), methods='GET').require(Role('admin'),
                                                                    methods=['PUT',
↪'POST', 'PATCH', 'DELETE'])
```

The configuration above will permit the access of the GET method to all clients authenticated with the role *user*, however it requires the role *admin* for the rest of the HTTP methods. Check out the details in the *Role Based Access Management* section for more details.

2.7.2 Full range of CRUD operations

Appkernel follows the REST convention for CRUD ((CR)eate(U)pdate(D)elete) operations:

- GET: to retrieve all, some or one model instance (entity);
- POST: to create a new entity or update an existing one;
- PUT: to replace an existing model instance;
- PATCH: to add or remove selected properties from an existing model instance;
- DELETE: to delete an existing model instance;

The path is automatically created from the class-name by convention.

Examples:

```
kernel.register(User)
```

This will expose the User model under: *http://localhost/user*.

The user with ID 12345678912 will be accessible at: *http://localhost/user/12345678912*

In case you would like to use a path prefix (eg. for verioning the API) you can register the model with a *url_base* segment:

```
kernel.register(User, url_base='/api/v1/')
```

In this case the User model is available at *http://localhost/api/v1/user* and *http://localhost/api/v1/user/12345678912* respectively.

Let's check out one example with *curls -X get http://localhost/api/v1/user/U9dbd7a25-8059-4005-8067-09093d9e4b06*:

```
{
    "_links": {
        "collection": {
            "href": "/users/",
            "methods": "GET"
        },
        "self": {
            "href": "/users/U9dbd7a25-8059-4005-8067-09093d9e4b06",
            "methods": [
                "GET"
            ]
        }
    },
    "_type": "User",
    "created": "2018-06-22T21:59:34.812000",
    "id": "U9dbd7a25-8059-4005-8067-09093d9e4b06",
    "name": "some_user"
}
```

In case the ID is not found in the database, a 404 Not found error will be returned.

```
Response: 404 NOT FOUND -> {
    "_type": "ErrorMessage",
    "code": 404,
    "message": "Document with id 1234 is not found."
}
```

Delete Model

Deleting an object is simple as well. Only that the method needs to be changed from GET to DELETE in the request.

```
curl -X DELETE http://localhost/U9dbd7a25-8059-4005-8067-09093d9e4b06
Response: 200 OK -> {
    "_type": "OperationResult",
    "result": 1
}
```

Create (POST)

Use json body for creating new instances:

```
curl -X POST --data {"birth_date": "1980-06-30T00:00:00", "description": "some␣
↪description", "name": "some_user", "password": "some_pass", "roles": ["User", "Admin
↪", "Operator"]} http://localhost/users/

Response: 201 CREATED -> {
    "_type": "OperationResult",
    "result": "U956c0b3c-cf5d-4bf5-beef-370cd7217383"
}
```

Alternatively you can send data as multi-part form data:

```
curl -X POST \
    -F name="some_user" \
    -F description="soe" \
    -F password="some pass" \
    -F birth_date="1980-06-30T00:00:00" \
    -F roles=["User", "Admin", "Operator"] \
    http://localhost/users

Response: 201 CREATED ->
{
    "_type": "OperationResult",
    "result": "U0054c3b6-dc0a-43ef-a10f-1ff705e90c36"
}
```

2.7.3 Filtering and Sorting

Query parameters are added to the end of the URL with a '?' mark. You can use any of the properties defined on the Model class. You can chain multiple parameters with the '&' (and) mark.

Between

Search users with a birth date between date:

```
curl http://localhost/users/?birth_date=>1980-06-30&birth_date=<1985-08-01&logic=AND
```

Contains

Search for users which contain *Jane* in the name property:

```
curl http://localhost/users/?name=~Jane
```

You can also search values within an array

```
curl http://localhost/users/?roles=~Admin
```

In

Search value within an array:

```
curl http://localhost/users/?name=[Jane,John]
```

Or

You can search for *Jane* or *John*:

```
curl http://localhost/users/?name=Jane&name=John&logic=OR
```

or:

```
curl http://localhost/users/?name=~Jane&&enabled=false
```

Not equal

Search all users which does not contain *Max* in the name property:

```
curl http://localhost/users/?name=!Max
```

Using Mongo query expression

Native Mongo Queries can be always provided as query parameters:

```
curl http://localhost/users/?query={"$or":[{"name":"John"}, {"name":"Jane"}]}
```

Sort

Sorting the result set is also easy, by using the *sort_by* expression:

```
curl http://localhost/users/?birth_date=>1980-06-30&sort_by=birth_date
```

Additionally you can specify the sort order:

```
curl http://localhost/users/?birth_date=>1980-06-30&sort_by=sequence&sort_order=DESC
```

2.7.4 Pagination

Pagination is supported with the use of *page* and *page_size*:

```
curl http://localhost/users/?page=1&page_size=5
```

. . . and of course sorting can be used in combination with pagination:

```
curl http://localhost/users/?page=1&page_size=5&sort_by=sequence&sort_order=DESC
```

Mongo Aggregation Pipeline

Additionally to native queries, Aggregation Pipeline is supported too:

```
curl http://localhost/users/aggregate/?pipe=[{"$match":{"name": "Jane"}}]
```

2.7.5 Custom resource endpoints

The built-in CRUD operations might be a good start for your application, however we would quickly run into situation where custom functionality needs to be exposed to the API consumers. In such cases the *@action* decorator comes handy. Let's suppose we need to provide the result of a specific method on the User:

```
class User(Model, MongoRepository):
    ...

    @action(require=Anonymous())
    def get_description(self):
        return self.description
```

And we're ready to go, you have a new endpoint returning the description property of the value and any user with the role *Anonymous* can access it:

```
curl http://localhost/users/U32268472-d9e3-46d9-86a2-a80926bd770b/get_description
```

Now one can argue, that this example is not utterly useful, a statement which in this case might not be very far from the common perception. However there's much more into it. Let's say that we'd like to enable the user and the admin to change the password for the User:

```
@action(method='POST', require=[CurrentSubject(), Role('admin')])
def change_password(self, current_password, new_password):
    if not pbkdf2_sha256.verify(current_password, self.password):
        raise ServiceException(403, _('Current password is not correct'))
    else:
        self.password = new_password
        self.save()
    return _('Password changed')
```

The `CurrentSubject` and `Role` authority controls who can access the method:

- **CurrentSubject**: in case the JWT token subject is identical with the model id, the access to the method is granted;
- **Role**: enables any user having the required role type call the method;

2.7.6 HATEOAS

By default HATEOAS support is enabled when a domain object is registered with Appkernel (*kernel.register(User)*). This means the return result-set includes browseable urls, exposing the existing methods to your API consumer.

```
{
  "_links": {
    "change_password": {
      "args": [
        "current_password",
        "new_password"
      ],
      "href": "/users/Ua4453112-0e7a-4f10-b95b-0d9b88493193/change_password",
      "methods": "POST"
    },
    "collection": {
      "href": "/users/",
      "methods": "GET"
    },
    "get_description": {
      "href": "/users/Ua4453112-0e7a-4f10-b95b-0d9b88493193/get_description",
      "methods": "GET"
    },
    "self": {
      "href": "/users/Ua4453112-0e7a-4f10-b95b-0d9b88493193",
      "methods": [
        "GET",
        "PUT",
        "POST",
        "PATCH",
        "DELETE"
      ]
    }
  },
  "_type": "User",
  "created": "2018-07-08T16:05:25.539000",
  "description": "test description",
  "id": "Ua4453112-0e7a-4f10-b95b-0d9b88493193",
  "name": "test user",
  "roles": [
    "Admin",
    "User",
    "Operator"
  ]
}
```

Would you not want to use the HATEOAS feature, you can chose to disable it at the Model registration phase *kernel.register(User, enable_hateoas=False)*.

2.7.7 HTTP Method Hooks

before_'http-method' **after_**'http-method'

So for post we could implement:

@classmethod def before_post(cls, *args, **kwargs):

order = kwargs['model']

2.7.8 Shema and metadata

All models provide JSON schema and a metatada to help frontend UI generation and data validation in frontends. Accessing the JSON schema is easy by calling **"http://root_url/{model_name}/schema"**

```
curl http://localhost/users/schema
```

Accessing the metadata by calling **"http://root_url/{model_name}/meta"** is easy too:

```
curl http://localhost/users/meta
```

2.7.9 Powered by Flask

The REST service engine uses Flask under the hood, therefore the reference to the flask app is always available at *kernel.app*.

2.8 Transparent REST client proxies

Warning: Work in progress section of documentation

2.9 I18n (Internationalisation)

Warning: Work in progress section of documentation

The translation support of **appkernel** is built on babel and flask-babel.

Strings which need to be internationalized should be marked with the babel specific '_' (underscore) marker function (eg. _('username')).

```
from flask_babel import _

def some_function():
    raise ServiceException(403, _('Current password is not correct'))
```

This method will work within the request context of flask, however when in need to work outside the request context we might need to use the `lazy_gettext` function:

```
from flask_babel import lazy_gettext as _l

class LoginForm(FlaskForm):
    username = StringField(_l('Username'), validators=[DataRequired()])
```

One can also add notes to the messages, which will be extracted into the translation files, helping the translator with context information.

```
# NOTE: This is a comment about `Foo Bar`
_('Foo Bar')
```

2.9.1 Preparation

You need to add a small configuration file, called *babel.cfg* to the root folder of your project:

```
[model_messages: **.py]
extract_messages = _l

;[python: **.py]
;extract_messages = _l

[jinja2: app/templates/**.html]
extensions=jinja2.ext.autoescape,jinja2.ext.with_\
```

The first two lines define the filename patterns for Python. We won't use the built-in python extractor because that is not reading the `Parameter` classes.

The third section defines two extensions provided by the Jinja2 template engine that help Flask-Babel properly parse template files. **Mind the path definition in the configuration file for the python and jinja file.**

2.9.2 Generating the translation files

To extract all the texts to the .pot file, you can use the following command (make sure that your're switched to your virtual environment):

```
(venv) $ pybabel extract -F babel.cfg -k _l -o messages.pot .
(venv) $ pybabel init -i messages.pot -d ./translations -l en
(venv) $ pybabel init -i messages.pot -d ./translations -l de
(venv) $ pybabel compile -d ./translations
```

The first command will create a list of key-strings in the local directory and write it into a messages.pot file. In our case it will search for strings marked with the babel specific _() function. The second and third command will copy the keys from the messages.pot into a folder called *en* and *de* inside of the folder translations. This is where the actual actual translation should take place. Once you're ready with the localisation, you are good to execute the *compile* command.

We need one more step, namely to add the supported languages to our configuration:

```
appkernel:
    i18n:
    languages: ['en-US','de-DE']
```

2.9.3 Updating the translation files

Once you add new text to your source-files, you can re-generate the translation source files with the following commands

```
(venv) $ pybabel extract -F babel.cfg -k _l -o messages.pot .
(venv) $ pybabel update -i messages.pot -d ./translations
(venv) $ pybabel compile -d ./translations
```

One can also use the

```
python ./setup.py compile_catalog --help
```

2.10 Role Based Access Management

Warning: Work in progress section of documentation

2.10.1 JWT Token

Appkernel uses JWT Token for authentication and authorisation of service calls. In order to generate a Token we need a Model class which extends the `IdentityMixin` and contains an *id* and a list of *roles*, which the principal (user or api client) holds:

```
class User(..., IdentityMixin):
    ...
    id = Property(str, required=True, generator=create_uuid_generator('U'))
    roles = Property(list, sub_type=str)
```

With this setup the User class will have a property, called auth_token

```
print(('token: {}'.format(user.auth_token))

{
    "created": "2018-07-08T20:29:23.154563",
    "description": "test description",
    "id": "Ue92d3b52-dd8b-4f10-a496-31b342b19cc9",
    "name": "test user",
    "roles": [
        "Admin",
        "User",
        "Operator"
    ]
}
```

The token is digitally signed with an RS256 algorithm.

2.10.2 Setup

The JWT token requires a pair of private-public key-pair which can be generated using openssl in the folder {config-folder}/keys

```
# lets create a key to sign these tokens with
openssl genpkey -out appkernel.pem -algorithm rsa -pkeyopt rsa_keygen_bits:2048
# lets generate a public key for it...
openssl rsa -in appkernel.pem -out mykey.pub -pubout
```

2.10.3 Role based authorisation

Configuring the default behaviour can be done right after registering the Model class to be exposed:

```
user_service = kernel.register(User, methods=['GET', 'PUT', 'POST', 'PATCH', 'DELETE
↪'])
user_service.deny_all().require(Role('user'), methods='GET').require(Role('admin'),
                                                                    methods=['PUT',
↪'POST', 'PATCH', 'DELETE'])
```

From now on, one needs the **Authorization** header on the requests with a valid Token containing the role *admin*. Example:

```
'Authorization':'Bearer eyJhbGciOiJSUzI1 ... 1Mjc0MzEzNDd9.'
```

In case there's a custom link method on one of your Model object, the *require* parameter will contain the list of `Permission`-s granting access to the method:

```
@action(method='POST', require=[CurrentSubject(), Role('admin')])
def change_password(self, current_password, new_password):
    ...
```

Current Permissions:

- Role - a permission, which enables a user who holds the named role to access the protected resource;
- Anonymous - a static Role, which grants access to unauthenticated users;
- Denied - a static Role, which should not be given to any user; Therefore permission will be added to all resources which should not be accessed at all;
- CurrentSubject - a special purpose Permission, which allows the access of a method if the object ID and the JWT subject id is the same (can be used for users

to modify their own data);

2.11 Why did we built this?

- We had the need to build a myriad of small services in our daily business, ranging from data-aggregation pipelines, to housekeeping services and other process automation services. These do share similar requirements and the underlying infrastructure needed to be rebuilt and tested over and over again. The question arose: what if we avoid spending valuable time on the boilerplate and focus only on the fun part?
- Often time takes a substantial effort to make a valuable internal hack or proof of concept presentable to customers, until it reaches the maturity in terms reliability, fault tolerance and security. What if all these non-functional requirements would be taken care by an underlying platform?
- There are several initiatives out there (Flask Admin, Flask Rest Extension and so), which do target parts of the problem, but they either need substantial effort to make them play nice together, either they feel complicated and uneasy to use. We wanted something simple and beautiful, which we love working with.
- These were the major driving question, which lead to the development of App Kernel.

2.12 How does it helps you?

We did the heavy lifting so you can focus on the things that matter :)

We believe you wish to focus entirely on delivering business value on day one and being the rockstar of your project. Therefore we took care of the boilerplate: analysed the stack, made the hard choices in terms of

Database/ORM/Security/Rate Limiting and so on, so you don't have to: **just lay back, fasten your seatbelts and enjoy the ride! ;)**

2.13 API Definition

2.13.1 App Kernel Engine

The main application class, which exposes the Service classes, manages Repositories and applies security.

2.13.2 Model

The base class to be extended by all Domain objects (Models). It has a set of useful methods, such as JSON marshaling, metadata (json schema) generation and validation. Example:

```
class User(Model):
        id = Property(str)
        name = Property(str, required=True, index=UniqueIndex)
        email = Property(str, validators=Email, index=UniqueIndex)
        password = Property(str, validators=NotEmpty,
                             converter=content_hasher(), omit=True)
```

2.13.3 Property

2.13.4 Validators

The base Validator class

Not Empty Validator

Regular Expression Validator

Email Validator

Minimum Validator

Maximum Validator

Past Validator

Future Validator

Unique Value Validator

2.13.5 Generators

UUID Generator

Date NOW Generator

Password hasher

2.13.6 Repository

The current implementation is the MongoRepository.

2.13.7 Query

2.13.8 MongoRepository

2.13.9 Auditable Repository

2.13.10 MongoQuery

2.13.11 Service

2.14 Development environment

Clone the project:

```
git clone git@github.com:accelero-cloud/appkernel.git
```

After cloning the project, you might want to setup a virtual environment:

```
cd appkernel
pip install --user pipenv
virtualenv -p python3 venv
source venv/bin/activate
pip install -e .
pip install pytest
pip install pytest-flask
pip install pylint
pip install flake8
```

Since astroid (a dependency of pylint) is not supporting python 3.7 yet, you might need to run the command from above if your pylint analysis ends with `RuntimeError:  generator raised StopIteration`.

```
pip install --pre -U pylint astroid
```

Hint for PyCharm users

- you might want to set the Project Interpreter (in the project settings) to the virtual environment just have created;
- you might want to set to excluded your .idea, appkernel.egg-info and venv folder in case of using Pycharm;

2.14.1 Setup git hooks

The project features pre-commit and pre-push hooks for automatically running tests and pylint:

```
cd .git/hooks
ln -sf ../../hooks/pre-commit ./pre-commit
ln -sf ../../hooks/pre-push ./pre-push
cd ../..
```

2.14.2 Preparing test execution

Some tests require compiled translations:

```
cd tests
pybabel compile -d ./translations
```

And many others a working local mongo db:

```
docker create -v ~/data:/data/db -p 27017:27017 --name mongo mongo
docker start mongo
```

. . . where `~/data` might be replaced by any folder where you would like to store database files;

Hint: the schema installation feature expects a MongoDB version min. 3.6. In case you have an older version you might need to upgrade your mongo image (`docker pull mongo:latest`).

Run the following command in the test folder:

```
pytest
```

2.14.3 Publish the project to PyPi

Make sure yuo have the latest twine version:

```
python3 -m pip install --upgrade twine
```

Make a test run:

```
python setup.py build -vf && python setup.py bdist_wheel
twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

Once we are ready we can upload the package the repo:

```
python setup.py build -vf && python setup.py bdist_wheel
twine upload dist/*
```

In case you have a ~/.pypirc you can use the shortcut names:

```
[distutils]
index-servers=
    pypi
    pypitest

[pypi]
#repository=https://pypi.python.org/pypi
username=user
password=pass

[pypitest]
#repository=https://testpypi.python.org/pypi
username=user
password=pass
```

```
twine upload -r pypitest dist/*
```

2.14.4 Migration to Python3

```
sudo apt install python3-pip
python -m pip install --upgrade pip
sudo update-alternatives --install /usr/bin/python python /usr/bin/python2.7 1
sudo update-alternatives --install /usr/bin/python python /usr/bin/python3.5 2
sudo update-alternatives --install /usr/bin/python python /usr/bin/python3.6 3
update-alternatives --list python
sudo pip install --upgrade pip
virtualenv -p /usr/bin/python3.6 venv3
source ./venv3/bin/activate
pip install pylint
```

2.15 Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

1. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

2. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

3. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

2.16 Crash Course (TL;DR)

Let's build a mini identity service:

```
class User(Model, MongoRepository):
    id = Property(str)
    name = Property(str, required=True, index=UniqueIndex)
    email = Property(str, validators=[Email], index=UniqueIndex)
    password = Property(str, validators=[NotEmpty],
                        converter=content_hasher(), omit=True)
    roles = Property(list, sub_type=str, default_value=['Login'])

kernel = AppKernelEngine('demo app')

if __name__ == '__main__':

    kernel.register(User)

    # let's create a sample user
    user = User(name='Test User', email='test@accelero.cloud', password='some pass')
    user.save()

    kernel.run()
```

Now we can test it by using curl:

```
curl -i -X GET 'http://127.0.0.1:5000/users/'
```

And check out the result

```
{
  "_items": [
    {
      "_type": "User",
      "email": "test@appkernel.cloud",
      "id": "0590e790-46cf-42a0-bdca-07b0694d08e2",
      "name": "Test User",
      "roles": [
        "Login"
      ]
    }
  ],
  "_links": {
    "self": {
      "href": "/users/"
    }
  }
}
```

That's all folks, our user service is ready to roll, the entity is saved, we can re-load the object from the database, or we can request its json schema for validation, or metadata to generate an SPA (Single Page Application). Of course validation and some more goodies are built-in as well :)

CHAPTER 3

Quick overview of some notable features

3.1 Built-in ORM function

Find one user matching the query parameter:

```
user = User.where(name=='Some username').find_one()
```

Return the first 5 users which have the role "Admin":

```
user_generator = User.where(User.roles % 'Admin').find(page=0, page_size=5)
```

Or use native Mongo Query:

```
user_generator = Project.find_by_query({'name': 'user name'})
```

3.2 Some more extras baked into the Model

Generate the ID value automatically using a uuid generator and a prefix 'U':

```
id = Property(..., generator=uuid_generator('U-'))
```

It will generate an ID which gives a hint about the object type (eg. *U-0590e790-46cf-42a0-bdca-07b0694d08e2*)

Add a Unique index to the User's name property:

```
name = Property(..., index=UniqueIndex)
```

Validate the e-mail property, using the NotEmpty and Email validators

```
email = Property(..., validators=[Email, NotEmpty])
```

Add schema validation to the database:

```
User.add_schema_validation(validation_action='error')
```

Hash the password and omit this attribute from the json representation:

```
password = Property(..., converter=content_hasher(rounds=10), omit=True)
```

Run the generators on the attributes and validate the resulting object (usually not needed, since it is implicitly called by save and dumps methods):

```
user.finalise_and_validate()
```

3.3 Setup role based access control

Right after exposing the service as a REST endpoint, security rules can be added to it:

```
user_service = kernel.register(User, methods=['GET', 'PUT', 'POST', 'PATCH', 'DELETE
↪'])
user_service.deny_all().require(Role('user'), methods='GET').
require(Role('admin'), methods=['PUT', 'POST', 'PATCH', 'DELETE'])
```

The configuration above will allow to GET *user* related endpoints by all users who has the **user** role. PUT, POST, PATCH and DELETE method are allowed to be called by users with the **admin** role.

3.3.1 JWT Token

Once the Model object extends the `IdentityMixin`, it will feature a property called **auth_token** which will contain a valid JWT token. All **roles** from the model are added to the token. Accessing the jqt token is simple:

```
token = user.auth_token
```

Python Module Index

Index