APIMAS Documentation

Release 0.3

GRNET

Sep 27, 2017

Contents

APIMAS provides a flexible way for building, modifying and extending your application without the cumbersome management due to the complexity and the size of it.

CHAPTER 1

Trying it out

apimas

Explore apimas package to find out how to model your REST API, and build and deploy your application.

First, create a virtualenv:

```
virtualenv virtualenv-apimas
source virtualenv-apimas/bin/activate
```

then, install apimas via pip:

```
pip install apimas
```

apimas-drf

For apimas support for building django applications, you should checkout apimas-drf package.

In a virtualenv run:

```
pip install apimas-drf
```

CHAPTER 2

Contents

Overview

APIMAS assumes applications are set up like this:

1. There is a REST API at a location `/<prefix>/api/*` with any static files available at `/<prefix>/static/*`

   This API includes multiple collections of REST resources.

   Each resource is defined as a collection of objects of the same type. The object type is defined as a set of fields with a name and type, including field types that refer to fields of other resources.

   For each resource collection there are built in operations that accept and return JSON data.

   - `POST /<prefix>/api/<resource>/ data: {key:val...} -> <url>`

     Creates a new resource in the collection. Its identifier is returned in the form:

     `/<prefix>/api/<resource>/<id>`

     The resource is initialized to the fields provided.

   - `GET /<prefix>/api/<resource>/ filter: {}`

     List the resource collection according to filters, ordering, and pagination input.

   - `PUT /<prefix>/api/<resource/<id> data: {key:val...}`

     Update (or create) resource with fields from input.

   - `GET /<prefix>/api/<resource>/<id> -> data: {key:val...}`

     Retrieve a `data: {key:val...}` representation of the resource.

   - `DELETE /<prefix>/api/<resource>/<id>`

     Remove the identified resource from the collection.

- `GET /<prefix>/api/<resource>/<id>/fields/<field> -> val`

  Resource fields are accessible recursively by their name under `fields/<field>`. The value when retrieving is the same that would be retrieved from the parent resource under the corresponding field key (`{<field>:<value>}`)

- `PUT /<prefix>/api/<resource>/<id>/fields/<field> val`

  Update a field of a resource with a new value. This value is identical to the one that would be provided by a PUT on the parent resource under the field key `{<field>:<val>}`

- `[Collection] /<prefix>/api/<resource>/<id>/fields/<field>/*`

  If a resource field is a collection then all above operations are optionally available recursively.

  Resource fields that are themselves collections are not equivalent to top-level resources. They are embedded on their parent resource and are retrieved and set along with the parent. Designers of APIs that need their collections to scale should make them top-level. Nested collections are supported to help data organization and convenience.

  Built-in operations can be overridden for customization. The application may also create new named actions:

- `POST /<prefix>/api/<resource>/<id>/actions/<action> data:{key:val...}`

  Execute application-provided actions with input.

2. Each top-level REST resource corresponds to a Data View that connects the API with the data store.

   The data view modelling completely defines the REST behaviour of the API locations. The data view may be linked and triggered in various interfaces (e.g. `GET /search/by-name/<name>/` being equivalent to `GET /resource/<some-id>`). Moreover, the underlying data may have arbitrary representation in actual storage. The data view is responsible for connecting the two layers.

3. The Data Storage layer models the actual representation of data in storage. There should be no native way to store data. The Data View layer should support adapting various storage layers for exposition to the REST API locations. For example, Django models may be one way to model actual storage representation.

4. The primary responsibility of the application is to hook at the Data View layer and provide storage and business logic. This logic must implement the hooks corresponding to all actions defined in 1.

5. However, the framework should provide various data store-to-view adapters (e.g. for Django, mongodb, S3). These adapters should themselves automatically hook to all actions defined in 1 and provide further contextualized hooks to applications.

   The store-to-view adapter must at least provide these hooks for modelling:

   (a) To expose arbitrary store-level fields to the REST API as-is

   (b) To create REST-level fields that do not directly correspond to store-level fields, but may be a function or transformation of one or multiple of them.

   and these hooks for business logic:

   (a) A hook immediately after validation of REST input that can affect the communication with the backend store.

   (b) A hook after communication with the backend store that can affect the response.

APIMAS Specification

APIMAS considers interfaces made from a hierarchy of Collections, Resources, and Properties.

Collections are made from an arbitrary number of resources of the same type. Resources in a collection are indexed by a primary property within each resource. Collections may have properties directly attached to them.

Resources are made from a set of properties with predefined names. Resources are of the same type if their properties have the same name and data.

Properties are named data items. The type of each item can be a simple value (e.g. text, integer) or it can be another resource or another collection.

Example:

```
/webstore/api/products/*/reviews/*/text/body
```

where:

- `'products'` is a collection
- `'*'` denotes collection resources
- `'reviews'` is a nested collection property
- `'text'` is a nested resource property
- `'body'` is a value property

Given an API location path as above, a rule of thumb is for each segment in the path:

- The last segment is a value
- If a segment has `'*'` in it, it is a resource
- If a segment has `'*'` in it, its parent is a collection
- All remaining segments are probably nested resources

In APIMAS, the specification of an API is a document object following the hierarchical structure of the API itself. This document may be encoded as a python dict, a JSON object, a yaml document, or a namespace with path-value pairs, or any other equivalent representation. The documents tool-kit in APIMAS uses the python dict form and can also convert to and from the namespace representation.

For example, in "namespace" representation:

```
spec_namespace = {
    'webstore/.endpoint': {},
    'webstore/api/products/.collection': {},
    'webstore/api/products/*/reviews/.collection': {},
    'webstore/api/products/*/reviews/*/text/body': '.unicode',
}
```

in Python `dict` representation:

```
spec_dict = {
    'webstore': {
        '.endpoint': {},
        'api': {
            'products': {
                '.collection': {},
                '*': {
                    'reviews': {
                        '.collection': {},
                        '*': {
                            'text': {
                                'body': '.unicode',
```

```
                            },
                        },
                    },
                },
            },
        },
    },
}
```

in YAML representation:

```
spec_yaml = yaml.load("""
    webstore:
      .endpoint: {}
      api:
        products:
          .collection: {}
          '*':
            reviews:
              .collection: {}
              '*':
                text: {body: .unicode}
""")
```

APIMAS predicates

Each node in the specification document contains structural items which appear in the API locations (e.g. `'products'`) and other metadata items that attach semantics to the containing nodes (e.g. `.unicode`). All metadata keys begin with a dot `.` to distinguish them from structural keys.

Metadata keys should have well-defined semantics shared by all specifications. We call these well known names as Predicates. Multiple predicates may be applied in the same node. The value of a metadata key is an arbitrary document (with structure and predicates) that parametrizes the semantics of the predicate.

For example a person's name can be specified to be a text of length between 6 and 64 characters, and their age to be an integer between 18 and 65:

```
person_spec = {
    'name': {
        '.text': {
            'minlen': '6',
            'maxlen': '64',
        },
    },
    'age': {
        '.integer': {
            'min': '18',
            'max': '65',
        },
    },
}
```

Predicates are not limited in format or range specifications but can represent any semantics we want them to.

For example, we can introduce a predicate named '.readonly' meaning that users cannot write that value through the API, or '.finalizer' which means that once this property is written, the whole resource becomes immutable.

The benefit of a common format of specification and a common library of predicates is that they offer existing patterns and concepts to address similar API challenges in the design phase, and then offer existing implementations for similar API designs.

Note that each application may introduce its own specific predicates that will not be reused anywhere else. Specification also helps by giving those application-specific concepts and requirements a name and a precise context.

APIMAS Configuration

The composition of structural elements and predicates as above forms a specification for the API that should be completely understood by all applications.

However, implementations of the API will necessarily require additional application-, or even deployment-specific settings, which we will collectively refer to as *configuration*.

Each application may invent its own predicates and build its configuration in a document similar to the specification. For instance, for a Django application a model can be bound to a specific collection like this:

```
conf = {
    'webstore/api/products/.drf_collection/model': 'myapp.models.MyModel',
}
```

Here, `.drf_collection` is a predicate for Django-rest-framework applications and it's has a `model` parameter.

For practicality, the application may choose to merge the specification document and the configuration document into a single working document containing both.

APIMAS Predicates

As explained in the previous section, an APIMAS specification contains structural elements with their respective metadata. This metadata are prefixed with a dot '.' and called *Predicates*. Predicates add semantics to their parent structural elements and therefore, it is a way to change the behaviour of your application.

APIMAS defines a set of predicates whose semantics are understood from every application (both client and server side) and help you create your specification. Below, there is a list of the widely-known predicates:

Structural predicates

The predicates listed below describe basic structural elements of your REST application.

Predicate	Description
`.endpoint`	It defines a location to the web after which there is a set of collections to interact.
`.collection`	It defines that the parent node is a collection of resources of the same type, where each resource can be related to other resources, it is described by some data, and there are actions that can be performed on it.

Action predicates

There are also predicates which delineate what actions or methods can be performed on a collection or a resource individually.

Predicate	Description
`.list`	The list of resources contained to the collection is permitted. It corresponds to: GET <collection name>/
`.retrieve`	A single resource can be retrieved and viewed. It corresponds to: GET <collection name>/<pk>/
`.create`	A new resource of the type defined by the collection can be created. It corresponds to: POST <collection name>/
`.update`	A single resource can be updated. It corresponds to: PUT <collection name>/<pk>/ PATCH <collection name>/<pk>/
`.delete`	A single resource can be deleted from the set of existing. It corresponds to: DELETE <collection name>/<pk>/

Note: The `.update` predicate allows a single resource to be both replaced and partially updated.

Action predicates are specified inside the structural element **actions** of a collection definition.

Example:

```
{
    'foo': {
        '.collection': {},
        'actions': {
            '.list': {},
            '.retrieve': {},
        }
    }
}
```

In the above example, only retrieve and list operations are permitted for collection 'foo'.

Resource description

Each resource contained in a particular collection is described by a field schema with properties and data associated with it. Specifically, each resource is described by a set of fields with specific type and behaviour.

Similarly to the structural and action predicate, there are also predicates to describe the properties of every field. These predicates slit into two categories: **a)**: types, **b)**: properties.

Type Predicates

Predicate	Description
`.integer`	Parent node is an integer.
`.float`	Parent node is a floating point number.
`.string`	Parent node is a string. Small to middle sized strings are supported. **Parameters:** `maxlength`: The upper bound of string's size (optional). The default is 255.
`.text`	Parent node is a text.
`.boolean`	Parent node is either true or false.
`.email`	Parent node is an email address.
`.serial`	Parent node is a serial, non-writable integer number.
`.choices`	Parent node can take a list of allowed values as specified by the parameter `allowed`. **Example**: `'foo': { '.choices': { 'allowed': [1, 'bar'] } }` Parent node can be either 1 or 'bar'. **Parameters:** `allowed`: A list of acceptable values for the parent. (Each item **must** be a literal). `display`: A list of the displayed values of the node (optional). By default is the same as `allowed`.
`.ref`	Parent node points to the web location of another resource. **Parameters:** `to`: Name of the collection where resource is located. This must be a valid name of a collection which have been specified on APIMAS specification too. `many`: true if parent node points to multiple resources, false otherwise.
`.identity`	Parent node points to the web location of this resource. It's actually the REST identifier of the resource. It is non-writable.
`.file`	Parent node is a file.
`.date`	Parent node is a date, represented by a string. **Parameters:** `format`: A list of string representing the allowed input formats of the date. (optional). By default only ISO-8601 is allowed.
`.datetime`	Parent node is a datetime, represented by a string. **Parameters:** `format`: A list of strings representing the allowed. input formats of the datetime. By default only ISO-8601 is allowed.
`.struct`	Parent node is a structure which consists of another field schema, i.e. a set of fields with their types and properties. **Arguments:** A document-like representation with name of fields as key and their description as defined by the use of predicates.

Note: Every field **must** be described with at most one type.

Properties predicates

Properties predicates, typically, describe the behaviour and how can be used on the various actions.

Predicate	Description
`.required`	The parent node is required and **must** be included in every API call associated with create and update operations (e.g. POST and PUT requests).
`.readonly`	The parent node is read-only and its value can be viewed, but it cannot be modified or set.
`.writeonly`	The parent node is write-only and its value can be modified or set, but it cannot be viewed.
`.nullable`	The parent node can have null values.

Note: Some predicates are mutually exclusive. Specifically a node cannot be described as both `.readonly` and `writeonly` or `.required` and `.readonly`.

Example–APIMAS Specification

Before you read this section, you should take a look at the APIMAS specification and predicates.

Suppose that you want to build a simple e-shop application which provide a REST API to its clients. A typical e-shop application has the following entities which are described by some fields:

- `users`: Users of the e-shop application which hold carts, make orders and buy products.
- `products`: A list of products described with a product key, price, a name, a description and a quantity.
- `carts`: Carts which contain a list of products.
- `orders`: Orders made by users to purchase a specific cart.

Given the above, representing your application's REST API is an easy task. First, you need to identify the collections of your application.

In this context of e-shop, consider the following collections of resource corresponding to every entity, and the following actions performed on them.

`users`:

```
POST   /api/users/
GET    /api/users/<pk>/
PUT    /api/users/<pk>/
PATCH  /api/users/<pk>/
```

`products`:

```
GET    /api/products/
GET    /api/products/<pk>/
```

`carts`:

```
POST   /api/carts/
GET    /api/carts/
GET    /api/carts/<pk>/
PUT    /api/carts/<pk>/
PATCH  /api/carts/<pk>/
DELETE /api/carts/<pk>/
```

`orders`:

```
POST   /api/orders/
GET    /api/orders/
GET    /api/orders/<pk>/
```

The above REST representation of your application is described by the following specification document.

```
API_SPEC = {
    'api': {
        '.endpoint': {},
        'users': {
            '.collection': {},
            '.actions': {
                '.retrieve': {},
                '.create': {},
                '.update': {},
            }
        },
        'products': {
            '.collection': {},
            '.actions': {
                '.list': {},
                '.retrieve': {},
            }
        },
        'carts': {
            '.collection': {},
            '.actions': {
                '.list': {},
                '.retrieve': {},
                '.create': {},
                '.update': {},
                '.delete': {},
            }
        },
        'orders': {
            '.collection': {},
            '.actions': {
                '.list': {},
                '.retrieve': {},
                '.create': {},
            }
        }
    }
}
```

First of all, we specified `.endpoint: {}` which indicates that a set of collections follows after a prefix `api/`. `.collection: {}` specifies that its parent node is a collection (e.g. *users* is a collection). `.actions` is a namespace predicate within which we define which REST actions are allowed to be performed on the collection.

Next, we need to define the underlying properties of the resources that are included in these collections i.e. their property schema. This is defined within the node '*'. Let's begin with `products` collection as a reference. A product is described by a key, a name, a description, a stock and a price. To expose this information to the REST API, we define something like the following which indicates that the aforementioned properties are *string*, *string*, *string*, *integer* and *float* respectively.

```
'products': {
    '.collection': {},
    '*': {
        'key': {
            '.string': {'max_length': 10}
        },
        'name': {
            '.string': {},
        },
        'description': {
            '.string': {},
        },
        'stock': {
            '.integer': {},
        },
        'price': {
            '.float': {},
        },
    },
    '.actions': {
        '.list': {},
        '.retrieve': {},
    }
}
```

This process can be repeated for all the collections of your application until you form the final specification. APIMAS provides a set of predicates which are used and understood from all the applications (which support APIMAS) to help you create your specification. Finally, we get something like this:

```
API_SPEC = {
    'api': {
        '.endpoint': {},
        'users': {
            '.collection': {},
            '*': {
                'id': {
                    '.serial': {},
                },
                'username': {
                    '.string': {},
                    '.required': {},
                },
                'first_name': {
                    '.string': {},
                    'required': {},
                },
                'last_name': {
                    '.string': {},
                    '.required': {},
                },
                'password': {
                    '.string': {},
```

```
                    '.required': {},
                    '.writeonly': {},
                },
                'email': {
                    '.email': {},
                    '.required': {},
                },
            },
            '.actions': {
                '.create': {},
                '.update': {},
                '.retrieve': {},
            }
        },
        'products': {
            '.collection': {},
            '*': {
                'key': {
                    '.string': {'max_length': 10}
                },
                'name': {
                    '.string': {},
                },
                'description': {
                    '.string': {},
                },
                'stock': {
                    '.integer': {},
                },
                'price': {
                    '.float': {},
                },
            },
            '.actions': {
                '.list': {},
                '.retrieve': {},
            }
        },
        'carts': {
            '.collection': {},
            '*': {
                'customer': {
                    '.required': {},
                    '.ref': {'to': 'api/users'},
                },
                'ordered': {
                    '.boolean': {},
                    '.readonly': {},
                },
                'products': {
                    '.readonly': {},
                    '.structarray': {
                        'key': {
                            '.string': {},
                        },
                        'name': {
                            '.string': {},
                        },
```

```
                        'price': {
                            '.float': {},
                        },
                    }
                },
            },
            '.actions': {
                '.list': {},
                '.retrieve': {},
                '.create': {},
                '.update': {},
                '.delete': {},
            },
        },
        'orders': {
            '.collection': {},
            '*': {
                'id': {
                    '.serial': {},
                    '.readonly': {},
                },
                'address': {
                    '.required': {},
                    '.string': {},
                },
                'date': {
                    '.datetime': {'format': ['%Y-%m-%d %H:%M']},
                    '.required': {},
                },
                'cart': {
                    '.ref': {'to': 'api/carts'},
                    '.required': {},
                }
            },
            '.actions': {
                '.list': {},
                '.create': {},
                '.update': {},
                '.delete': {},
                '.retrieve': {},
            }
        },
    }
}
```

Note: `cart` field of collection `orders` points to a resource of another collection, i.e. `carts` as specified in the *'api/carts'* location of specification.

See also:

For the full reference, see APIMAS predicates.

Django Applications

You can easily build a server-side application by using an APIMAS backend. Currently, the only backend supported is `apimas-drf` which is uses django rest framework to build REST APIs on top of a django application.

Installation

In a virtualenv, run the following command to install apimas-drf:

```
pip install apimas-drf
```

Quickstart-Create a django application

At this point, we assume that you are familiar with django basic concepts and have some experience with developing django applications.

Starting point

As a starting point, you have to define your django models. Based on your models and your specification, APIMAS will create the classes implementing the application's REST API.

According to the guide in section, you can specify a collection of resources named *foo*, where all REST operations are allowed:

```
API_SPEC = {
    'api': {
        '.endpoint': {},
        'foo': {
            '.collection': {},
            '*': {
                'text': {
                    '.string': {}
                },
                'number': {
                    '.integer': {},
                },
            },
            'actions': {
                '.list': {},
                '.retrieve': {},
                '.create': {},
                '.update': {},
                '.delete': {},
            }
        }
    }
}
```

Given the specification above, you have to create the corresponding django-model in the project's `models.py` file.

```
from django.db import models

class Foo(models.Model):
```

```
    text = models.CharField(max_length=20)
    number = models.IntegerField()
```

Enrich APIMAS specification

In order to link the specification of the collection to the django model you have to declare *'foo'* as a *django rest framework* collection and *text* and *number* as fields, using the predicates `.drf_collection` and `.drf_field`, respectively:

```
     API_SPEC = {
    'api': {
        '.endpoint': {},
        'foo': {
            '.collection': {},
            '.drf_collection': {
                'model': 'myapp.models.Foo'
            },
            '*': {
                'text': {
                    '.string': {},
                    '.drf_field': {},
                },
                'number': {
                    '.integer': {},
                    '.drf_field': {},
                },
            },
            'actions': {
                '.list': {},
                '.retrieve': {},
                '.create': {},
                '.update': {},
                '.delete': {},
            }
        }
    }
}
```

In the above example, we introduced two new predicates which are not included in the APIMAS standard predicates: a) `.drf_collection`, b) `.drf_field`. These predicates are understood only by the *django-rest-framework* backend, which is responsible for implementing this specification.

Set permissions

APIMAS provides a mechanism for setting the permissions of your application. You can read more in a next section. However, for this tutorial, we omit the description of this mechanism. Thus, you have to add the following configuration on your specification.

```
     API_SPEC = {
    'api': {
        '.endpoint': {
            'permissions': [
                # That is (collection, action, role, field, state, comment).
                ('foo', '*', 'anonymous', '*', '*', 'Just an example')
            ]
```

```
        },
        'foo': {
            '.collection': {},
            '.drf_collection': {
                'model': 'myapp.models.Foo'
            },
            '*': {
                'text': {
                    '.string': {},
                    '.drf_field': {},
                },
                'number': {
                    '.integer': {},
                    '.drf_field': {},
                },
            },
            'actions': {
                '.list': {},
                '.retrieve': {},
                '.create': {},
                '.update': {},
                '.delete': {},
            }
        }
    }
}
```

This tells APIMAS, that an anonymous user can perform any action ('*' on 2nd column) on collection 'foo', associated with any field ('*' on 4th column) and any state ('*' 5th column). The last column is used to write your comments. More about permissions can be found here.

Use DjangoRestAdapter

Then, APIMAS will create all required code using `DjangoRestAdapter` class. In particular, `DjangoRestAdapter` will create the mapping of URL patterns and views (`urlpatterns`). This mapping is specified specify on your `URLconf` module (typically, the `urls.py` file on your django-project).

For example, in `urls.py` file:

```
from apimas.drf.django_rest import DjangoRestAdapter
from myapp.spec import API_SPEC

adapter = DjangoRestAdapter()
adapter.construct(API_SPEC)

urlpatterns = [
    adapter.urls
]
```

Now, you are ready to test your application, by running:

```
python manage.py runserver
```

You can make some testing calls using `curl`. For example, create a new resource object

```
curl -X POST -d '{"text": "foo", "number": 1}' -H "Content-Type: application/json"␣
↪http://localhost:8000/api/foo/
```

```
{
    "number": 1,
    "text": "foo"
}
```

or, retrieve an existing one:

```
curl -X GET http://localhost:8000/api/foo/1/
```

```
{
    "number": 1,
    "text": "foo"
}
```

django-rest-framework adapter

So far, we have seen a short tutorial on using APIMAS to create a *django* application. We easily created an application which served a REST API, by only defining the storage django-models) and the view (APIMAS specification, i.e. API representation) representation of our application. Typically, apart from the django-models, a django-developer has to create the corresponding django forms and views in order to map url patterns with implementation. Hence, for a typical example a developer has to make the following classes:

`models.py`:

```
from django.db import models

class Foo(models.Model):
    text = models.CharField(max_length=30)
    number = models.IntegerField()
```

`forms.py`

```
from django import forms
from myapp.models import Foo

class FooForm(forms.ModelForm):

    class Meta(object):
        model = Foo
        fields = ('number', 'text',)
```

`views.py`

```
import json
from django.http import HttpResponse
from myapp.forms import FooForm

def view_foo(request):
    form = FooForm()
    return render(request, 'path/to/template', form)
```

Even when using *django-rest-framework* which facilitates the development of the REST API, the developer typically has to create boilerplate such as:

`serializers.py`

```
from rest_framework import serializers
from myapp.models import Foo

class FooSerializer(serializers.ModelSerializer):

    class Meta:
        model = Foo
        fields = ('number', 'text')
```

views.py

```
from rest_framework import viewsets
from myapp.serializers import FooSerializer
from myapp.models import Foo

class FooViewSet(viewsets.ModelViewSet):
    serializer_class = FooSerializer
    queryset = Foo.objects.all()
```

Even though in the above examples things seem to be easy, the management of such an application may become cumbersome if more entities are introduced or the complexity of data representation of an entity is increased, e.g. if we have an entity with 30 fields, and each field behaves differently according to the state of the entity (e.g. non-accessible in read operations).

As already mentioned in a previous section, APIMAS provides a way to describe your application and its data representation on a document. The *django-rest-adapter* reads from the specification and it translates the description of your application into implementation. The *django-rest-adapter* uses *django-rest-framework* behind the scenes and generates at runtime the required `rest_framework.serializers.Serializer` (responsible for the serialization and deserialization of your request data) and `rest_framework.viewsets.ViewSet` classes according to the specification.

In essence, your application consists of your storage and API representation, and each time, you want to change something on your API representation, you simply refer to the corresponding properties of your specification.

django-rest adapter's workflow

The *django-rest* adapter creates the corresponding mapping of url patterns to views based on the storage and API representation of your application. Therefore, for a typical application we have the following work flow:

- In a list operation (`GET <collection name>/`), the list of objects included in the model associated with the collection, is retrieved.
- In a retrieve operation (`GET <collection name>/<pk>/`), a single model instance is displayed based on its API representation.
- In a create operation (`POST <collection name>/`), sent data are validated, and then a model instance is created after serializing data.
- In an update operation (`PUT|PATCH <collection name>/pk/`), sent data are validated, serialized, and the new values of model instance are set.
- In a delete operation (`DELETE <collection name>/pk/`), a model instance, identified by the `<pk>` is deleted.

Customize your application

If the default behaviour above does not suit the application, you are able to customize and extent it by adding your own logic. Specifically, APIMAS provides two hooks for every action (before interacting with the database and after) for extending the logic of your application or executing arbitrary code (e.g. executing a query or sending an email to an external agent). You can do this as follows:

```
from apimas.drf.mixins import HookMixin

class RestOperations(HookMixin):

    def preprocess_create(self):
        # Code executed after validating data and before creating
        # a new instance.
        ...

    def finalize_create(self):
        # Code executed after creating the model instance and
        # and before serving the response.
        ...
```

If you want to customize the behaviour of your application in other actions, you simply have to add the corresponding methods to your class, e.g.

- `preprocess_<action_name>(self)` (for executing code before interacting with db)
- `finalize_<action_name>(self)` (for executing code before serving the response and after interacting with db).

Customize your application - A simple case scenario

Imagine that we have the following model:

```
from django.db import models

class Foo(models.Model):
    text = models.CharField(max_length=30)
    number = models.IntegerField()
    another_text = models.CharField(max_length=30)
```

and the API specification for this model:

```
API_SPEC = {
    'api': {
        '.endpoint': {},
        'foo': {
            '.drf_collection': {
                'model': 'myapp.models.Foo'
            },
            '*': {
                'text': {
                    '.string': {},
                    '.drf_field': {}
                },
                'number': {
                    '.integer': {},
                    '.drf_field': {}
```

```
                },
            },
            'actions': {
                '.list': {},
                '.retrieve': {},
                '.create': {},
                '.update': {},
                '.delete': {}
            }
        }
    }
}
```

In the above example, the field `another_text` is not exposed to the API, but its value is computed by the server based on the values of `text` and `number`. Therefore, in this case, you may write your hook class like below:

```
from myapp.mymodule.myfunc

class RestOperations(HookMixin):
    def preprocess_create(self):
        context = self.unstash()
        another_text = myfunc(context.validated_data['text'],
                              context.validated_data['number'])
        self.stash(extra={'another_text': another_value})
```

Here we get the context of the action via the `self.unstash()` method, then we compute the value of `another_text` according to some application logic, and finally, we tell APIMAS (`self.stash()`) that it should add extra data to the model instance (`another_text`), in addition to those sent by the client. `self.unstash()` returns a namedtuple with the following fields:

- `instance`: Model instance to interact.
- `data`: Dictionary of raw data, as sent by the client.
- `validated_data`: Dictionary of de-serialized, validated data.
- `extra`: A dictionary with extra data, you wish to add to your model.
- `response`: Response object.

Note that in some cases, there are some context fields that are not initialized. For instance, in the `preprocess_create()` hook, `instance` is not initialized because model instance has not been created yet.

The last part is to declare the use of the hook class. You have to provide an argument to the `hook_class` parameter of the `.drf_collection` predicate.

```
'foo': {
    '.drf_collection': {
        'model': 'myapp.models.Foo',
        'hook_class': 'myapp.hooks.RestOperations',
    },
    # spec as above.
}
```

Write django-rest-framework code

As we have already mentioned, django-rest adapter generates dynamically two classes: a) a serializer class, b) a viewset class according to the specification. If you still wish to customize and override these generated classes,

APIMAS provides various ways to do that:

- Override these classes with your own classes.
- Add additional attributes.

There are two primary reasons to do this:

- django-rest adapter has not abstracted the full functionality of django-rest-framework yet.
- You may have reasons to override the internal functionality of django-rest-framework.

Below, we describe two common cases when you need to write django-rest-framework code.

Deal with structures

In your API, you may have structural fields, that is, all fields characterized as `.struct` or `.structarray`. django-rest-framework backend does not support write operations, because they are read-only by default. Hence, if you want to be able to perform write operations on these fields, you have to override the `create()` or/and `update()` methods, provided by each serializer class.

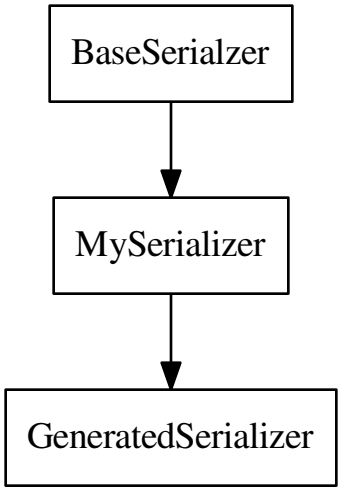
Example:

```
from rest_framework.serializers import BaseSerialzer

class MySerializer(BaseSerializer):

    def create(self, validated_data):
        # Your code
        ...

    def update(self, instance, validated_data):
        # Your code.
        ...
```

Then, in your specification, specify the following parameter in `.drf_collection` predicate:

```
'foo': {
    '.drf_collection': {
        'model': 'myapp.models.Foo',
        'model_serializers': ['myapp.serializers.MySerializer'],
    },
    # spec as above.
}
```

`model_serializers` tells APIMAS that the classes specified should be base classes for the generated serializer class, which are placed to the lowest level of the inheritance hierarchy. Therefore, in the above example, the hierarchy of the generated class is as follows:



If you specify more than one classes on your `model_serializers`, then the classes on the right will inherit the classes on the left.

Further information about writable structure fields can be found in the official documentation of django-rest-framework, here.

Add more actions to your API

You can have additional actions to your API apart from the CRUD ones you declare in the specification. For example:

```
POST foo/1/myaction/
```

To implement `myaction` you need to write your own ViewSet class that includes a method with the action's name. For instance:

```
from rest_framework.decorators import detail_route
from rest_framework.viewsets import GenericViewSet

class MyViewSet(GenericViewSet):

    @detail_route(methods=['post'])
    def myaction(self, request, pk):
        # My code.
        ..
```

Next, you need to include the module path of your ViewSet mixin class in the `mixins` parameter of your `.drf_collection` predicate. APIMAS will inherit from your class and the extra action method will appear in the generated final ViewSet class.

```
'foo': {
    '.drf_collection': {
        'model': 'myapp.models.Foo',
        'mixins': ['myapp.mixins.MyViewSet'],
```

```
    },
    # spec as above.
}
```

You can find more information about extra actions here.

Note: Specifying bases and mixins for the generated viewse class enhances the resusability of your code. For instance, you may have a custom ViewSet class which is shared amongst all your collections. Instead of copying the same code over and over across different hooks, you can declare a common mixin for all of them within your specification.

---

django-rest-framework fields

By default, the django-rest adapter reads all REST resource properties predicated with `.drf_field` and tries to map each of them to an attribute or function on your django model. It is not necessary to have 1 to 1 mapping between your API and storage configuration. For instance, you may want to:

- expose a field with different name as that specified in your model.
- define fields in your API which are not intended to be stored in your db.
- create responses with arbitrary structure.

Examples:

Define the name of source field explicitly

In this example, we create an `api_text` property on a REST resource that is mapped to a differently named `text` field on a django model, using the `source` parameter of the `.drf_field` predicate:

```
from django.db import models

class Foo(models.Model):
    text = models.CharField(max_length=30)
    number = models.IntegerField()
```

```
'foo': {
    '.drf_collection': {
        'model': 'myapp.models.Foo',
    },
    '*' {
        'api_text': {
            '.string': {},
            '.drf_field': {
                'source': 'text'
            }
        },
        'number': {
            '.integer': {},
            '.drf_field': {},
        },
    },
}
```

Use non-model fields

You can create REST resource properties that are not mapped to any of the django model fields. In the following example, we add a string property named "extra_field" to our specification that is not to be saved to or retrieved from the model, by specifying `onmodel: False` to the `.drf_field` predicate.

```
'foo': {
    '.drf_collection': {
        'model': 'myapp.models.Foo',
    },
    '*' {
        'api_text': {
            '.string': {},
            '.drf_field': {
                'source': 'text'
            }
        },
        'number': {
            '.integer': {},
            '.drf_field': {},
        },
        'extra-field': {
            '.string': {},
            '.drf_field': {
                'onmodel': False,
            },
        },
    },
}
```

A non-model property is validated but there is no automatic handling of it during write actions. You have to handle it via the hooks provided by APIMAS.

When processing read actions such as list or retrieve, the django-rest adapter will seek to call a function to extract the value of non-model properties since there is no model for them. If you want non-model fields to be readable, you must provide an argument to the `instance_source` parameter on the `.drf_field` predicate. The parameter is enabled only when `onmodel` is False. `instance_source` must be the module path of a function that accepts a model instance as input and returns the property value.

```
def myfunc(instance):
    # Code which retrieves the value of a non-model field based on
    # the instance.
    pk = instance.pk

    # Open a file, identified by the pk of the instance and
    # extract the desired value.
    with open('file_%s.txt' % (str(pk)), 'r') as myfile:
        data = myfile.read()
    return data
```

```
'foo': {
    '.drf_collection': {
        'model': 'myapp.models.Foo',
    },
    '*' {
        'api_text': {
            '.string': {},
```

```
            '.drf_field': {
                'source': 'text'
            }
        },
        'number': {
            '.integer': {},
            '.drf_field': {},
        },
        'extra-field': {
            '.string': {},
            '.drf_field': {
                'onmodel': False,
                'instance_source': 'myapp.mymodule.myfunc'
            },
        },
    },
}
```

Create structured responses

Apart from the things already mentioned, one additional reason for having non-model fields is to create responses with arbitrary structure. For instance, instead of returning the following response:

```
{
    "text": "foo",
    "number": 10
}
```

you wish to return this:

```
{
    "data": {
        "text": "foo",
        "number": 10
    }
}
```

Your django-model is not aware of the node "data". Therefore, you need to format your specification as:

```
'foo': {
    '.drf_collection': {
        'model': 'myapp.models.Foo',
    },
    '*' {
        'data': {
            '.drf_field': {'onmodel': False},
            '.struct': {
                'api_text': {
                    '.string': {},
                    '.drf_field': {
                        'source': 'text'
                    }
                },
                'number': {
                    '.integer': {},
                    '.drf_field': {},
```

```
                    },
                }
            }
        },
}
```

where node "data" is a structured non-model property consisting of model fields "api_text" and "number".

Warning: All fields on a model must be exposed to the same REST location. They must not be scattered among different nodes in the specification.

APIMAS permissions

APIMAS implements a built-in mechanism for setting permissions to your server-side application. The permissions of your application consist of a set of rules. Each rule contains the following information:

- `collection`: The name of the collection to which the rule is applied.
- `action`: The name of the action for which the rule is valid.
- `role`: The role of the user (entity who performs the request) who is authorized to make request calls.
- `field`: The set of fields that are allowed to be handled in this request (either for writing or retrieval).
- `state`: The state of the collection which **must** be valid when the request is performed.
- `comment`: Any comment for documentation reasons.

Set permission rules

Consider the following example rule:

```
rule = ('foo', 'create', 'admin', 'text', 'open', 'section 1.1')
```

The rule indicates that a request for the collection *foo*, which is asking to *create* a new resource, and is issued by an *admin*, is allowed to create a *text* property when the collection is in an *open* state. *section 1.1* is a comment made by the developer and it is ignored.

To enable writing another field *number*, write one more rule:

```
rule = ('foo', 'create', 'admin', 'text', 'open', 'section 1.1')
rule2 = ('foo', 'create', 'admin', 'number', 'open', 'section 1.1')
```

or write a pattern to match the two properties:

```
rule = ('foo', 'create', 'admin', 'text|number', 'open', 'section 1.1')
```

Supported APIMAS operators for matching are:

- `*`: Any pattern.
- `?`: Pattern indicated by a regular expression.
- `_`: Pattern starts with the given input.
- `!`: NOT operation.
- `&`: AND operation.

- |: OR.

For example, the following rule reveals that an admin or a member ('admin|member') can perform any ('*') action any on collection starts wit 'foo' ('_foo'), provided that they handle fields matched with a particular expression ('?ition$') and the state is 'open' and 'valid' at the same time ('open&valid').

```
rule = ('_foo', '*', 'admin|member', '?ition$', 'open&valid', 'section 1.1')
```

The set of your rules must be declared in your specification as a parameter to the `.endpoint` predicate.

Example:

```
{
    'api': {
        '.endpoint': {
            'permissions': [
                ('foo', 'create', 'admin', 'text', 'open', 'section 1.1'),
                # More rules...
                ...
            ]
        }
    },
}
```

APIMAS permissions – Roles

In order to check against the roles specified in permission rules, you have assign to roles to an authenticated user by setting them as a list of strings named `apimas_roles` on your user instance as in:

```
request.user.apimas_roles = ['admin', 'dev']
```

```
class User(models.Model):
     ...

     @property
     def apimas_roles(self):
         ...
```

Unauthenticated users

Requests by unauthenticated users are matched by the `anonymous` role in permission rules. Using anonymous roles you can make part of your API public. For example, the following rule allows anyone to create `foo` resources as long as `foo` is in an `open` state:

```
rule = ('foo', 'create', 'anonymous', '*', 'open', 'section 1.1')
```

APIMAS permissions – Fields

The 'field' column of a rule, corresponding to field, indicates which field(s) are allowed to be handled. For instance:

- For a write-operation, only the fields defined in your rules are allowed to be written. Thus, if someone sent some data that are not validated against your rules, they would be ignored.

- For a read-operation, only the fields defined in your rules can be accessed. The rest are not displayed to the client.

APIMAS permissions – States

States are matched if calling a class method on the model associated with the request returns true. There is a different method for checking a state for collection (list, create) versus resource requests. The names and signatures of the methods are as follows:

```
@classmethod
def check_collection_state_<state name>(cls, row, request, view):
    # your code. Return True or False.
    ...

@classmethod
def check_resource_state_<state name>(cls, obj, row, request, view):
    # your code. Return True or False.
    ...
```

For example, imagine you have the following permission rules:

```
rule = ('foo', 'create', 'anonymous', '*', 'open', 'section 1.1')
rule2 = ('foo', 'update', 'anonymous', 'number', 'submitted', 'section 1.1')
```

In the above example, in the case of an update operation, the methods listed below will be triggered to check if states 'open' or 'submitted' are satisfied:

- `check_state_collection_open()`
- `check_state_resource_submitted()`

If none of the states is matched, then an HTTP_403 error is returned. If only one state is matched, then the django-rest adapter checks which fields can be handled in this state, e.g. when the state is 'open', an anonymous user can set all fields, while when the state is 'submitted' only the field 'number' can be updated.

django-rest adapter predicates

Below, there is a list of the predicates introduced by the django-rest adapter along with their semantics.

Predicate	Description
`.drf_collection`	The parent node is a collection of resources of the same type, where each resource can be related to other resources, it is described by some data, and there are actions that can be performed on it. The parent node uses *django-rest-framework* backend. **Parameters:** `model`: String of the django-model corresponding to the storage representation of the collection. `authentication_classes`: (optional) List of classes used for the authentication of the collection. More here. `permission_classes`: (optional) List of the classes responsible for the permissions of the collection. More here. `mixins`: (optional) List of the bases classes of the `ViewSet` class generated by django-rest adapter. `model_serializers`: (optional) List of bases classes of the `ApimasModelSerializer` (class responsible when having model-fields) generated by django-rest adapter. `serializers`: (optional) List of base classes of the `ApimasSerializer` (class responsible when having non-model fields) generated by django-rest adapter. `hook_class`: (optional) A class which implements hooks before and after interacting with db for various actions. See *more*.
`.drf_field`	The parent node is a drf_field. In other words, it is an instance of a django-rest-framework field which is responsible for converting raw value of a field (sent by client) into complex data such as objects, querysets, etc. **Parameters:** `onmodel`: True if field has a storage representation, False otherwise (default: True). `source`: Name of the storage representation of the field (Default is the name of the parent). `instance_source`: A string which points to a function which retrieves the value of the field given the current instance (applicable if `onmodel: False`).

Client-Side Applications

APIMAS supports the creation of client-side applications to interact with the REST API described by your specification. There is a client-adapter which is responsible for the conversion of specification into implementation. The logic behind this conversion is similar with that of server-side applications which use the corresponding adapter.

Apimas Client Adapter

The `ApimasClientAdapter` is the bridge between specification and python objects which represent the client of each collection. In other words, these clients enable you to interact with a REST API programmatically.

Therefore, given the specification below, you use this class to construct client objects. This class is initialized with the root url of the server we want to interact. In the end of the construction process, the client objects have been constructed and you can extract a client object for a particular collection via `adapter.get_client()`. This object provides you the following methods to interact with the API:

- `list()`
- `retrieve()`
- `create()`
- `update()`
- `partial_update()`
- `delete()`

```
from apimas.clients import ApimasClientAdapter

API_SPEC = {
    'api': {
        '.endpoint': {},
        'foo': {
            '.collection': {},
            '*': {
                'text': {
                    '.string': {}
                },
                'number': {
                    '.integer': {},
                },
            },
            'actions': {
                '.list': {},
                '.retrieve': {},
                '.create': {},
                '.update': {},
                '.delete': {},
            }
        }
    }
}

adapter = ApimasClientAdapter('http://localhost:8000')
adapter.construct(API_SPEC)

clients = adapters.clients
foo_client = adapter.get_client('foo')

data = {'text': 'bar', 'integer': 1}

# Create a new foo resource object.
# It performs a POST http://localhost:8000/foo/ request.
response = foo_client.create(data=data)
print response.data, response.status_code
```

```
# List resources of foo collection.
# It performs a GET http://localgost:8000foo/ request.
response = foo_client.list()
print response.data, response.status_code
```

`ApimasClientAdapter` uses python requests to make the necessary HTTP calls and cerberus for validating data.

Authentication

Before you interact with the API, you may want to authenticate your party. For this reason, client objects generated by the client-adapter provide method `set_credentials` with the following signature:

```
def set_credentials(self, auth_type, **credentials):
    ...
```

You have to provide the type of the authentication, e.g. basic, token, etc. and your credentials.

Example:

```
client = adapter.get_client('foo')
client.set_credentials('basic', username='foo',
                       password='passoword')
client.retrieve('1')
```

Before retrieving a single resource, we had to set our credentials according to the specified authentication mode. Each authentication mode supports different credentials schema. For instance, if you use basic authentication, you **must** provide a username and a password.

Supported authentication modes:

Authentication Mode	Credentials Schema
basic	• username • password
token	• token

Create a CLI for your client - ApimasCliAdapter

In case you wish to create a command line interface (CLI) for your client-side application, APIMAS offers a built-in adapter which creates the CLI for you based on your specification. This is `ApimasCliAdapter` class which introduces two new predicates a) `.cli_commands`, b) `.cli_option`.

But first, you have to create a configuration file, say `.apimas` on a directory of your choice, written in yaml syntax.

For example, in `myloc/.apimas`:

```
root: http:localhost:8000
spec:
    api:
        .endpoint: {}
        foo:
            .collection: {}
            .cli_commands: {}
            '*':
```

```
                text:
                    .cli_option: {}
                    .string: {}
                number:
                    .cli_option: {}
                    .integer: {}
            actions:
                .list: {}
                .retrieve: {}
                .create: {}
                .update: {}
                .delete: {}
```

The CLI-adapter constructs a set of commands for every collection based on that file. For example, for the collection *foo*, we have the following commands corresponding to every action as specified on specification:

- `apimas --config myloc/.apimas api foo-list`
- `apimas --config myloc/.apimas api foo-retrieve`
- `apimas --config myloc/.apimas api foo-create`
- `apimas --config myloc/.apimas api foo-update`
- `apimas --config myloc/.apimas api foo-delete`

Apparently, these five commands use the same client object internally, that is, the client object which is responsible for interacting with the collection `foo`. Option `--config` tells apimas where to find the configuration file. Note that sub-command `api` stands for the endpoint (i.e. `api`) in which collection is located.

Also note that if one action is not specified on specification, the corresponding command is not created. For instance, if we remove the `.list` predicate, there will not be the `apimas foo-list` command.

Generally, the generated command has the following format:

```
apimas <endpoint> <collection>-<action> --<option1> --<option2>
```

Command options

For write-actions, i.e. create and update, you have to pass some data according to the data description of your collection (i.e. fields). For this purpose, you have to create some command options by enriching your specification using `.cli_option` predicate. This tells adapter to create an option for the command, keeping all the other properties of the node. For instance, the presence of `.required` predicate will make the option required, etc.

Example:

```
apimas api foo-create --text foo --number 1
```

In the above example, we use the `foo-create` command to create a new resource of collection foo, setting text as *foo* and number as 1. Also note that it is not necessary for the names of command-line options and fields to be verbatim equal.

Example:

```
root: http:localhost:8000
spec:
    api:
        .endpoint: {}
        foo:
```

```
            .collection: {}
            .cli_commands: {}
            '*':
                text:
                    .cli_option:
                        option_name: text-option
                    .string: {}
                number:
                    .cli_option:
                        option_name: number-option
                    .integer: {}
            actions:
                .list: {}
                .retrieve: {}
                .create: {}
                .update: {}
                .delete: {}
```

In the above example, we specified the parameter `option_name` in `.cli_option` predicate which defines the name of the command option and it creates a mapping with the name of the API field.

```
apimas api foo-create --text-option foo --number-option 1
```

However, the HTTP request which is going to be made by the client, has still the structure as defined by the specification.

Structural fields

Imagine we have two more fields which describe the collection *foo*. One is a `.struct` (i.e. field *"foo"*) and the other is `.structarray` (i.e. field *"bar"*).

```
root: http:localhost:8000
spec:
    api:
        .endpoint: {}
        foo:
            .collection: {}
            .cli_commands: {}
            '*':
                text:
                    .cli_option: {}
                    .string: {}
                number:
                    .cli_option: {}
                    .integer: {}
                foo:
                    .cli_option: {}
                    .struct:
                        age:
                            .cli_option: {}
                            .integer: {}
                        name:
                            .cli_option: {}
                            .string: {}
                bar:
                    .cli_option: {}
```

```
                    .structarray:
                        age:
                            .cli_option: {}
                            .integer: {}
                        name:
                            .cli_option: {}
                            .string: {}
            actions:
                .list: {}
                .retrieve: {}
                .create: {}
                .update: {}
                .delete: {}
```

The command options are created as follows:

- In case of `.struct`, a command option for every nested field prefixed by the name of parent node is created.
- In case of `.structarray`, a single command option is created which takes a JSON as input.

Example:

```
apimas api foo-create --foo-age 1 --foo-name myname --bar '[{"age": 1, "name": "myname
↪"}]'
```

Resource actions

Commands performed on single resources, have a required command argument which is the identifier of the resource to the set of the collection.

Example:

```
apimas api foo-update bar --data foo --number 1
apimas api foo-retrieve bar
apimas api foo-delete bar
```

We performed update, retrieve and delete actions on a resource of collection *foo*, identified by the name **"bar"**.

Authentication

If you want to provide your credentials in order to be authenticated before interacting with your collection, you have to enrich your specification, using `.cli_auth` predicate. The `.cli_auth` predicate creates a new **required** option named `--credentials` for every command of your collection. This command options takes a file path as input. This points to a file where your credentials are provided. The format of your file is indicated by the parameter `format` inside `.cli_auth`. The supported formats are a) yaml, b) json. In addition, this file **must** provide your credentials based on the credentials schema which you have specified on your specification.

Example:

```
root: http:localhost:8000
spec:
    api:
        .endpoint: {}
        foo:
            .collection: {}
            .cli_commands: {}
```

```
            .cli_auth:
                format: yaml
                schema:
                    basic:
                        -username
                        -password
            '*':
                text:
                    .cli_option: {}
                    .string: {}
                number:
                    .cli_option: {}
                    .integer: {}
            actions:
                .list: {}
                .retrieve: {}
                .create: {}
                .update: {}
                .delete: {}
```

Then, your file where your credentials are stored should be as follows:

`mycredentials.yaml`

```
basic:
    username: myusername
    password: mypassword
```

Now you are ready to execute all commands:

```
apimas api foo-list --credentials ~/mycredentials.yaml
apimas api foo-retrieve bar --credentials ~/mycredentials.yaml
apimas api foo-create --text foo --number 1 --credentials ~/mycredentials.yaml
apimas api foo-update bar --text foo --number 1 --credentials ~/mycredentials.yaml
apimas api foo-delete bar --credentials ~/credentials.yaml
```

Multiple Authentication Modes

If you need multiple authentication modes, then you should specify all of them on your specification. Then, you should add the `.cli_auth` predicate to your specification. In the following example, a client can be authenticated with two possible authentication modes, i.e. `basic` and `token`.

```
.cli_auth:
    format: yaml
    schema:
        basic:
            -username
            -password
        token:
            -token
```

In this case, you can provide credentials for both authentication modes on your credentials file. However, only one authentication mode is used each time. You can select which one you want to use by specifying `default`. If default is not specified, then the first authentication mode is used.

For example:

credentials.yaml

```
default: token
basic:
    username: myusername
    password: mypassword
token:
    token: mytoken
```

License