Resource API

Release 3.1.1

March 23, 2015

Contents

Resource API is a framework that allows developers declaratively define resources and relationships between each other. The framework maintains referential integrity for all relations and resources. The integrity is maintained regardless of data storage model.

Also the framework provides means to expose HTTP interface to the resources via a simple JSON document.

CHAPTER 1

> Tutorial <

The aim of this tutorial is to provide a comprehensive overview of Resource API components and how they are supposed to be used to implement real-life applications.

Note: Some of the sections and examples of this tutorial intersect with the information provided by the actual code documentation. This *is* intentional.

1.1 Intro

For the sake of this tutorial lets try to design a simple school course management system backed with a primitive python shelve backend that is reasonable to use during prototype phase. We shall implement a granular authorization using means of Resource API framework. After the implementation is done we shall play with direct Python API and HTTP interface.

1.2 1: collect the use cases

Our course simple management system must be able to support the following actions:

- students can sign up to courses
- teachers can kick the students out from the courses
- teachers can grade the students during the courses
- students can make comments about the course and rate its teacher respectively
- teachers are supposed to have information like qualification, diploma, etc.

1.3 2: determine entities

Based on usecases mentioned above we can list the following entities (in singular):

- student
- teacher
- course
- comment

- student grade
- teacher rating

Also the following links are supposed to exist:

- student can sign up for MANY courses and each course can be attended by MANY students
- teacher can teach MULTIPLE courses but each course can be taught only by a SINGLE teacher
- teacher can grade a student ONCE for every course the teacher owns
- every student can grade every teacher ONCE for every course the teacher was teaching
- every student can make comments about the courses as much as he wants

1.4 3: entity diagram

Note: Designing the perfect course management system is not among the goals of this tutorial. It aims to demonstrate how Resource API facilitates dealing with implementation issues.

There are several ways to model the system. For the sake of this example we shall look at teachers and students as at separate entities.

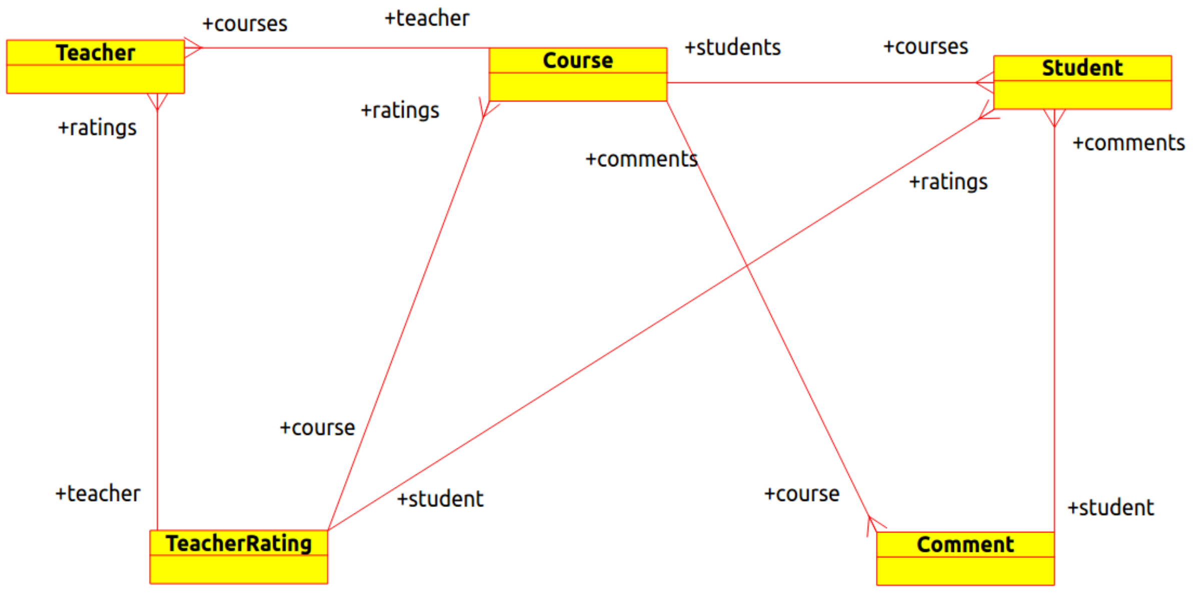

Figure 1.1: Entity diagram

1.5 4: ontology to code

After we managed to come up with an idea on what kind of resources we are supposed to have and how they are expected to link to one another we need to write python code that would correspond to these structures.

```python
from resource_api.interfaces import Resource, Link


class Student(Resource):
    """ A pupil """

    class Links:

        class courses(Link):
            """ Courses the student has ever attended """
            target = "Course"
            related_name = "students"
            master = True

        class comments(Link):
            """ Comments made by the student """
            target = "Comment"
            related_name = "student"

        class ratings(Link):
            """ Ratings given by the student """
            target = "TeacherRating"
            related_name = "student"


class Teacher(Resource):
    """ A lecturer """

    class Links:

        class ratings(Link):
            """ Ratings given to the teacher """
            target = "TeacherRating"
            related_name = "teacher"

        class courses(Link):
            """ Courses the teacher is responsible for """
            target = "Course"
            related_name = "teacher"


class Course(Resource):
    """ An educational unit represinting the lessons for a specific set of topics """

    class Links:

        class teacher(Link):
            """ The lecturer of the course """
            target = "Teacher"
            related_name = "courses"
            cardinality = Link.cardinalities.ONE
            master = True
            required = True

        class comments(Link):
            """ All comments made about the course """
            target = "Comment"
            related_name = "course"
        class ratings(Link):
            """ All ratings that were given to the teachers of the specific course """
            target = "TeacherRating"
            related_name = "course"

        class students(Link):
            """ All pupils who attend the course """
            target = "Student"
            related_name = "courses"


class Comment(Resource):
    """ Student's comment about the course """

    class Links:

        class student(Link):
            """ The pupil who made the comment """
            target = "Student"
            related_name = "comments"
            cardinality = Link.cardinalities.ONE
            master = True
            required = True

        class course(Link):
            """ The subject the comment was made about """
            target = "Course"
            related_name = "comments"
            cardinality = Link.cardinalities.ONE
            master = True
            required = True


class TeacherRating(Resource):
    """ Student's rating about teacher's performance """

    class Links:

        class student(Link):
            """ The pupil who gave the rating to the teacher """
            target = "Student"
            related_name = "ratings"
            cardinality = Link.cardinalities.ONE
            master = True
            required = True

        class course(Link):
            """ The subject with respect to which the rating was given """
            target = "Course"
            related_name = "ratings"
            cardinality = Link.cardinalities.ONE
            master = True
            required = True

        class teacher(Link):
            """ The lecturer to whom the rating is related """
            target = "Teacher"
            related_name = "ratings"
            cardinality = Link.cardinalities.ONE
            master = True
            required = True
```

As you can see every relationship (or link) consists of two edges of a bidirectional relationship. Even though we will dive into implementation details of the relationships later on it is critical to highlight that the **main** purpose of Resource API is to maintain relational integrity in a similar fashion to graph or SQL databases.

There are couple of important things to note in the code above.

First, all links must have a **target** and **related_name** defined. A combination of these two attributes lets the framework bind two edges into a single link entity.

Second, one of the edges must be marked as a **master** one. Structural validation (if there is any structure) and authorization are performed against the master edge only. Storing data in one place is a logical way to save storage space. And the employed approach of authorization lets the following scenario be possible: if someone can add a student to the course then the same user could add the course to the student's course list.

Third, there is a cardinality attribute. There are two possible values for this one. **ONE** and **MANY**. The edge with cardinality **ONE** does not differ from the **MANY** implementation-wise. However, the framework returns a single object for the **ONE** and a collection for **MANY** via *object interface*.

Check *interface documentation* to find out more about link attributes.

1.6 5: structure of the entities

Apart from the relationships between the resources there is another bit of knowledge which is vital for modeling the system. It is the structure of individual resources (e.g.: Student must have an email, first name, last name and a birthday).

Lets expand our graph to include structural information as well.

As you can see from the relationship between **Course** and **Student**, links may have attributes as well. Resource API supports link properties in a similar fashion as Neo4j.

1.7 6: structure to code

Full version of the code can be seen `here`. Below we shall focus on the critical bits of the implementation.

Lets have a look at Student's *schema* :

```
class Student(Resource):
    """ A pupil """

    class Schema:
        email = StringField(regex="[^@]+@[^@]+\.[^@]+", pk=True,
                            description="Addess to which the notifications shall be sent")
        first_name = StringField(description="Given name(s)")
        last_name = StringField(description="Family name(s)")
```

As you can see the structure of the entity is exposed declaratively. Instead of writing multiple functions to validate the input we just say what the input is supposed to be. This approach of describing entities' structures is similar to the one used by Django models for example.

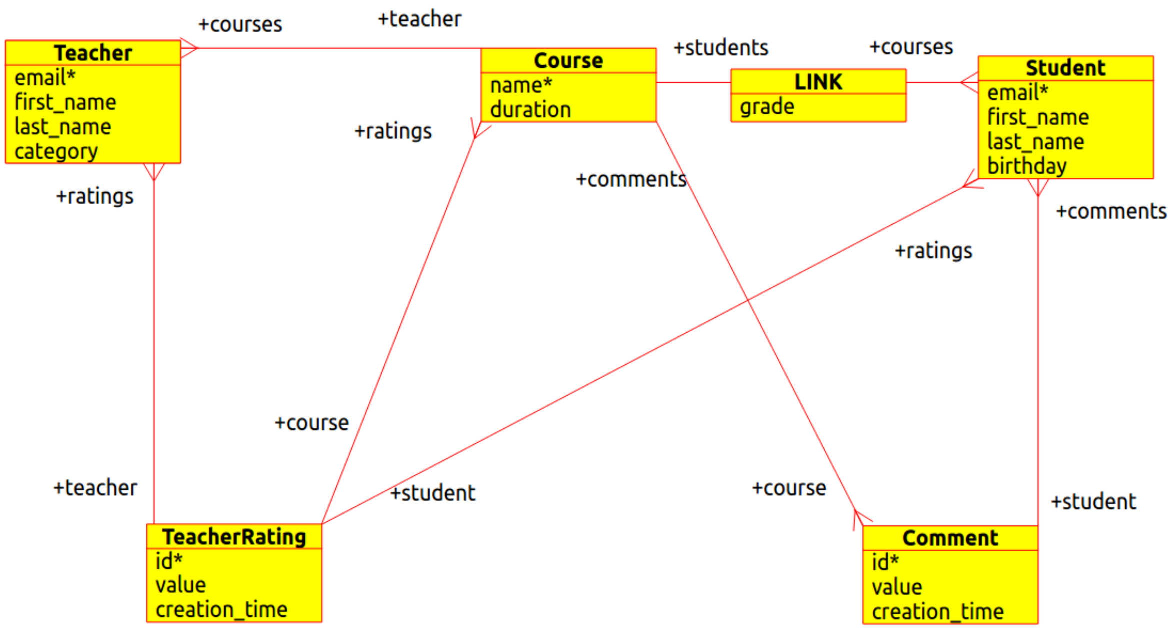

Figure 1.2: Entity diagram with structure

The key difference from Django's approach is usage of the inner class called *Schema*. The nested class exists to prevent naming collisions between user defined fields and the fields that are used by framework internals. Links are defined in a separate nested class for the same reason.

1.8 7: persistence

When we have resource structures defined in our code it is still not enough to make the entities do anything useful. We need to program how these entities are supposed to be stored an fetched to/from the persistence layer, file system, etc.

To make the components work we need to implement *Resource/Link interface*.

A full version of the implementation can be found `here`. Below we shall focus on critical implementation details.

First, lets have a look at Resource implementation:

```
class Resource(BaseResource):

    def __init__(self, context):
        super(Resource, self).__init__(context)
        self._storage = context["storage"]

    def exists(self, user, pk):
        return pk in self._storage.get(self.get_name(), {})

    def get_data(self, user, pk):
        return self._storage.get(self.get_name(), {}).get(pk)

    def delete(self, user, pk):
        self._storage.get(self.get_name(), {}).pop(pk)
        self._storage.sync()

    def create(self, user, pk, data):
        if self.get_name() not in self._storage:
            self._storage[self.get_name()] = {}
        self._storage[self.get_name()][pk] = data
        self._storage.sync()

    def update(self, user, pk, data):
        self._storage[self.get_name()][pk].update(data)
        self._storage.sync()

    def get_uris(self, user, params=None):
        return self._storage.get(self.get_name(), {}).keys()

    def get_count(self, user, params=None):
        return len(self.get_uris(params))
```

Next, lets check Link implementation:

```
class Link(BaseLink):

    def __init__(self, context):
        super(Link, self).__init__(context)
        self._storage = context["storage"]

    def exists(self, user, pk, rel_pk):
        return rel_pk in self._storage.get((pk, self.get_name()), {})

    def get_data(self, user, pk, rel_pk):
        return self._storage.get((pk, self.get_name()), {}).get(rel_pk)

    def create(self, user, pk, rel_pk, data=None):
        key = (pk, self.get_name())
        if key not in self._storage:
            self._storage[key] = {}
        self._storage[key][rel_pk] = data
        self._storage.sync()

    def update(self, user, pk, rel_pk, data):
        self._storage[key][rel_pk].update(data)
        self._storage.sync()

    def delete(self, user, pk, rel_pk):
        self._storage.get((pk, self.get_name()), {}).pop(rel_pk)
        self._storage.sync()

    def get_uris(self, user, pk, params=None):
        return self._storage.get((pk, self.get_name()), {}).keys()

    def get_count(self, user, pk, params=None):
        return len(self.get_uris(pk, params))
```

As you can see each abstract method of both interfaces are implemented to use Shelve database.

Warning: Note, compensating transactions are one of the TODO features to be added to Resource API in the future. Now any error in the implementation of the resource when creating/deleting the entity with multiple associated links has high chances to cause relational inconsistency.

Now let examine the service class:

```
class ShelveService(Service):

    def __init__(self):
        super(ShelveService, self).__init__()
        self._storage = shelve.open(SHELVE_PATH, writeback=True)

    def _get_context(self):
        return {"storage": self._storage}

    def _get_user(self, data):
        return None

    def __del__(self):
        self._storage.close()
```

ShelveService implements abstract `Service`. We had to override two abstract methods.

First, *_get_context*. This method must return an object that shall be passed to all resources during initialization. The context shall be available as a *context* attribute of resource objects. It makes sense to put service-wide singletons like DB connections, persistence layers or open sockets into the context.

Second, *_get_user*. More on it later. But in short it is expected to return a user that would be used for authorization later on.

In addition to service and entity implementations there are a few more important lines:

```
srv = ShelveService()
srv.register(Student)
srv.register(Teacher)
srv.register(Course)
srv.register(Comment)
```

The lines above provide an overview of how to notify the system that certain resources are supposed to be exposed. Each resource must be registered with a respective method call in order to become a part of the API.

Also notice a *setup()* method call. It must be invoked after all the required resources are registered. The method validates that resource relationships point to registered resources. Meaning: if we registered a *Student* but did not register a *Course* - Resource API would raise a `ResourceDeclarationError`.

1.9 8: primary key

When addressing the resource Resource API follows the standards and employs a URI concept. In the example above the URI is represented by a field marked as a primary key. What the framework does by default - it takes the value of the field and passes it to resources `set method`. Resource is a synonym of word *entity* within the context of Resource API.

In contrast with a *student* resource the following *Comment* entity has a primary key that does not have any direct value (unlike the *Student's email*) for the end user. Thus passing it together with the rest of the data during entity creation does not make sense. For this purpose we need to override URI creation mechanism.

1.10 9: custom UriPolicy

To change the way URI is generated and processed for a specific resource we need to subclass `UriPolicy` and implement a bunch of its methods.

A full version of the service with custom UriPolicy can be found here. Below we shall focus on important details of the implementation.

Lets have a look at *Comment* definition:

```
class Comment(Resource):
    """ Student's comment about the course """

    UriPolicy = AutoGenSha1UriPolicy
```

In order to override URI creation mechanism we explicitly changed *UriPolicy* from the default one to *AutoGenSha1UriPolicy*.

Lets have a closer look at *AutoGenSha1UriPolicy*:

```
class AutoGenSha1UriPolicy(AbstractUriPolicy):
    """ Uses a randomly generated sha1 as a primary key """

    @property
    def type(self):
        return "autogen_policy"

    def generate_pk(self, data):
        return os.urandom(16).encode('hex')

    def serialize(self, pk):
        return pk

    def deserialize(self, pk):
        if not isinstance(pk, basestring):
            raise ValidationError("Has to be string")
        if not RE_SHA1.match(value):
            raise ValidationError("PK is not a valid SHA1")
        return pk
```

There are three abstract methods that were implemented.

First, *getnerate_pk*. It returns a random SHA1 string.

Second, *serialize* method. Since we do not change the URI anyhow when storing the resource we return it as is.

Third, *deserialize* method. Here we validated that input value is a string and that it fits a SHA1 regular expression.

1.11 10: authorization

Since we want to limit the access to various resources only to specific categories of users, we need to implement authorization using granular *can_* methods of *Link* and *Resource* subclasses.

Full implementation of authorization can be seen here. Below we shall focus on authorization details.

Lets have a look at the methods that limit read-only access to the entities only for authenticated users.

```
def can_get_data(self, user, pk, data):
    """ Only authenticated users can access data """
    if user.is_anonymous:
        return False
    else:
        return True
def can_get_uris(self, user):
    """ Only authenticated users can access data """
    if user.is_anonymous:
        return False
    else:
        return True
```

We just return *False* if a user is *anonymous*. We shall see how the user object should be created later on.

Since we wanted to let *Students* and *Teachers* update only their own info, we encapsulated authorization logic within a *Person* class.

```
class Person(Resource):

    class Schema:
        email = StringField(regex="[^@]+@[^@]+\.[^@]+", pk=True,
                            description="Addess to which the notifications shall be sent")
        first_name = StringField(description="Given name(s)")
        last_name = StringField(description="Family name(s)")

    def can_update(self, user, pk):
        """ Only a person himself can update his own information """
        return user.email == pk or user.is_admin

    def can_delete(self, user, pk):
        """ Only admins can delete people """
        return user.is_admin
```

And both *Student* and *Teacher* inherit from the *Person*:

```
class Student(Person):
    """ A pupil """

    class Schema(Person.Schema):
        birthday = DateTimeField()

class Teacher(Person):
    """ A lecturer """

    class Schema(Person.Schema):
        category = StringField(description="TQS Category",
                               choices=["four", "five", "five plus", "six"])
```

Notice, that we also extracted common bits of the schema into *Person.Schema*. Thus *Student* and *Teacher* schemas inherit from it.

Within the scope of our app it makes sense that teachers can create only courses for themselves and students can make comments only on their own behalf.

In order to enforce this behavior we introduced a *PersonalLink*:

```
class PersonalLink(Link):
    """ Users can link things to their accounts only """

    def can_update(self, user, pk, rel_pk, data):
        return user.email == rel_pk or user.is_admin

    def can_create(self, user, pk, rel_pk, data):
        return user.email == rel_pk or user.is_admin
```

And made *Course*'s link to *Teacher* and *Comment*'s and *TeacherRating*'s link to *Student* inherit from the *PersonalLink*:

```
        class teacher(PersonalLink):
            """ The lecturer of the course """

        class student(PersonalLink):
            """ The pupil who made the comment """

        class student(PersonalLink):
            """ The pupil who gave the rating to the teacher """
```

One last bit of authorization detail required to understand how the implementation is done - a user object. It can be virtually anything. However, it is critical to note that this object is passed to all authorization methods as the first parameter by the framework.

Lets have a look at what the school app does with the user:

```
    def _get_user(self, data):
        if data is None:
            return User(None)
        else:
            return User(data.get("email"))
```

Where class *User* is defined the following way:

```
class User(object):

    def __init__(self, email=None):

        if email is None:
            self.is_anonymous = True
        else:
            self.is_anonymous = False

        if email == "admin@school.com":
            self.is_admin = True
        else:
            self.is_admin = False

        self.email = email
```

As simple as that.

1.12 11: object interface

Object interface provides a python way for traversing the resources.

In order to do the traversal on our school service we need to fetch the entry point.

```
entry_point = srv.entry_point({"email": "admin@school.com"})
student_root_collection = entry_point.get_resource(Student)
student_root_collection.create({"email": "student@school.com",
                                "first_name": "John",
                                "last_name": "Smith",
                                "birthday": "2000-09-25"})
```

Note: Please check *object interface* docs for more detailed information on how to use the direct Python API.

1.13 12. HTTP interface

In order to make HTTP interface work, service instance has to be passed to a WSGI application:

```
srv = ShelveService()
srv.register(Student, "school.Student")
srv.register(Teacher, "school.Teacher")
srv.register(Course, "school.Course")
srv.register(Comment, "school.Comment")
srv.register(TeacherRating, "school.TeacherRating")


from resource_api_http.http import Application
from werkzeug.serving import run_simple
app = Application(srv)
run_simple('localhost', 8080, app, use_reloader=True)
```

Full version of the file (which can be executed as a full featured app) can be found `here`.

Lets have a look at the most significant bits in the declaration.

First, notice how the resources are registered:

```
srv.register(ResourceClass, "namespace.ResourceName")
```

This is in general a good practice to register all entities under a specific name so that the API is not too tightly coupled with Python modules & class names. Module name and class name are used by default as a namespace and a resource name respectively.

Second, we removed *setup()* call. WSGI application does it internally anyways.

Third, the application is passed to Werkzeug's method. Werkzeug is a WSGI library powering Resource API HTTP component.

When the service is up and running it is possible to do HTTP requests with CURL:

Fetch service *descriptor* via OPTIONS request:

```
curl -X OPTIONS 127.0.0.1:8080 | python -m json.tool
```

Fetch a collection of students:

```
curl 127.0.0.1:8080/foo.Student
```

Oh. 403 status code. This is because we did not include authentication information required for authorization.

```
curl --header "email: admin@school.com" 127.0.0.1:8080/school.Student
```

Empty collection. Lets create a student.

```
curl -X POST --header "email: admin@school.com" --header "Content-Type: application/json" \
     -d '{"email":"foo@bar.com","first_name": "John", "last_name": "Smith", "birthday": "1987-02-21T
      127.0.0.1:8080/school.Student
```

Lets fetch Student collection again:

```
curl --header "email: admin@school.com" 127.0.0.1:8080/school.Student
```

As you can see a new student appeared in the list.

Please check *HTTP interface reference* for more information.

CHAPTER 2

Interfaces

There are two major entities in this framework: resources and links between them.

Resource API defines interfaces to be implemented in order to expose the entities.

NOTE: all methods must be implemnted. In case if some of the methods are not supposed to do anything, raise *NotImplemntedError* within their implementations and return *False* for respective authorization methods if needed.

2.1 Resource

Resource concept is similar to the one mentioned in Roy Fielding's desertation.

class `resource_api.interfaces.`**`Resource`**`(`*context*`)`

Represents entity that is supposed to be exposed via public interface

Methods have the following arguments:

pk PK of exisiting resource

data (dict) information to be stored within the resource

params (dict) extra parameters to be used for collection filtering

user (object) entity that corresponds to the user that performs certain operation on the resource

`UriPolicy`

alias of `PkUriPolicy`

`__init__``(`*context*`)`

context (object) entity that is supposed to hold DAL (data access layer) related functionality like database connections, network sockets, etc.

`can_create``(`*user*, *data*`)`

Returns True if user is allowed to create resource with certain data

`can_delete``(`*user*, *pk*`)`

Returns True if user is allowed to delete the resource

`can_discover``(`*user*, *pk*`)`

Returns False if user is not allowed to know about resoure's existence

`can_get_data``(`*user*, *pk*, *data*`)`

Returns only the fields that user is allowed to fetch

`can_get_uris``(`*user*`)`

Returns True if user is allowed to list the items in the collection or get their count

can_update(*user*, *pk*, *data*)
Returns True if user is allowed to update the resource

create(*user*, *pk*, *data*)
Creates a new instance

delete(*user*, *pk*)
Removes the resource

exists(*user*, *pk*)
Returns True if the resource exists

get_count(*user*, *params=None*)
Returns total amount of items that fit filtering criterias

get_data(*user*, *pk*)
Returns fields of the resource

get_uris(*user*, *params=None*)
Returns an iterable over primary keys

update(*user*, *pk*, *data*)
Updates specified fields of a given instance

URI is represented by a PK (Primary Key) in Resource API.

Resource interface defines two types of methods.

First, DAL related CRUD methods: get_data, get_pks, set, delete, exists

Second, authorization related methods starting with *can_*

Each resource must define a **UriPolicy**:

class resource_api.interfaces.**AbstractUriPolicy**(*resource_instance*)
Defines a way to generate URI based on data that was passed when creating the resource.

__init__(*resource_instance*)

resource_instance (Resource instance) entity that can be used to access previously created items

deserialize(*pk*)
Transforms data sent over the wire into sth. usable inside DAL

pk PK value as it comes over the wire - e.g. string in case of HTTP

@return PK transformed to the data type expected to by DAL in order to fetch data

generate_pk(*data*, *link_data=None*)
Generates a PK based on input data

data (dict): the same data that is passed to Resource's *create* method

link_data (dict): the same link_data that is passed to Resource's *create* method

@return generated PK

get_schema()
Returns meta information (dict) to be included into resource's schema

serialize(*pk*)
Transforms value into sth. ready to transfer over the wire

pk PK value used within DAL to identify stored entries

@return PK transformed into something that can be sent over the wire - e.g. string in case of HTTP

type
A string that would give a hint to the client which PK policy is in use

The default pk policy is this one:

class `resource_api.interfaces.`**`PkUriPolicy`**(*resource_instance*)
Uses value of a field marked as "pk=True" as resource's URI

Note, there are certain cases when the URI is supposed to be generated within peristence (data acess) layer. E.g. via autoincrementing primary key in SQL database. In such case the URI is supposed to be returned by *create* method.

```
class Example(Resource):

    class UriPolicy(AbstractUriPolicy):

        def deserialize(self, pk):
            try:
                return int(pk)
            except ValueError:
                raise ValidationError("URI is not int")

        def serialize(self, pk):
            return pk

        @property
        def type(self):
            return "autoincrement_pk_policy"

        def generate_pk(self, data, link_data=None):
            return None

    def create(self, pk, data):
        # assert pk is None
        row_id = self._sql_database.create_row(data)
        return row_id

    ...
```

2.2 Link

Link concept is derived from RDF triples . RDF is at the same time a part of two big W3C standardized concepts: Linked Data and Semantic Web.

NOTE: Resource API does not aim to follow any of standards defined by the concepts mentioned above. It just uses a portion of interesting ideas that those concepts describe.

class `resource_api.interfaces.`**`Link`**(*context*)
Represents a relationship between two resources that needs to be exposed via public interface

Methods have the following arguments:

pk PK of exisiting source resource (the one that defines link field)

data (dict) extra information to be stored for this relationship

rel_pk (digit|string) PK of exisiting target resource (the one to which we are linking to)

params (dict) extra parameters to be used for collection filtering

user (object) entity that corresponds to the user that performs certain operation on the link `__init__`(*context*)

context (object) entity that is supposed to hold DAL (data access layer) related functionality like database connections, network sockets, etc.

`can_create`(*user*, *pk*, *rel_pk*, *data*)
Returns True if user is allowed to create resource with certain data

`can_delete`(*user*, *pk*, *rel_pk*)
Returns True if user is allowed to delete the resource

`can_discover`(*user*, *pk*, *rel_pk*)
Returns False if user is not allowed to know about resoure's existence

`can_get_data`(*user*, *pk*, *rel_pk*, *data*)
Returns only the fields that user is allowed to fetch

`can_get_uris`(*user*, *pk*)
Returns True if user is allowed to list the items in the collection or get their count

`can_update`(*user*, *pk*, *rel_pk*, *data*)
Returns True if user is allowed to update the resource

`create`(*user*, *pk*, *rel_pk*, *data=None*)
Creates a new link with optional extra data

`delete`(*user*, *pk*, *rel_pk*)
Removes the link. If rel_pk is None - removes all links

`exists`(*user*, *pk*, *rel_pk*)
Returns True if the link exists (is not nullable)

`get_count`(*user*, *pk*, *params=None*)
Returns total amount of items that fit filtering criterias

`get_data`(*user*, *pk*, *rel_pk*)
Returns link data

`get_uris`(*user*, *pk*, *params=None*)
Returns an iterable over target primary keys

`update`(*user*, *pk*, *rel_pk*, *data*)
Updates exisiting link with specified data

Link interface defines two types of methods similar to the ones of Resource interface.

There are conceptual differences between those methods in Link and Resource though.

First, link uses a triple (mentioned above) to address exisiting entities.

Second, data is optional for links.

It is critical to note that *predicate* part of a triple is not passed to any of Link methods. Since all links are supposed to be defined via nested classes in the context of resources they connect - link classes themselves serve as those *predicates*.

2.3 Link declaration

Links between resources are defined using nested classes:

```
class Course(Resource):

    class Links:
        class attendants(Link):
            target = "Student"
            related_name = "active_cources"
            master = True

            def get(self, pk, rel_pk):
                ...

class Student(Resource):

    class Links:

        class active_cources(Link):
            target = "Course"
            related_name = "attendants"

            def get(self, pk, rel_pk):
                ...
```

target has to be a string. It can point to a resource in the same module ("Target") or in any other one ("module.name.Target"). *related_name* must be defined as a string as well and it should equal to the name of a related link.

Also one of the links must be defined as a *master* one. Authorization is done against *master* link. And extra data is stored only in DAL related to *master* link.

Any link can be marked as *changeable = False*. Unchangeable links can be set only upon resource creation. Once the resource is created links cannot be modified (i.e. updated/set or deleted).

All link declarations must be done within *Links* inner class.

2.4 One way links

Note: If the link is marked as *one_way* Resource API will not be able to enforce relational integrity.

One way links do not need a *related_name* nor a *master* flag to be defined. One way links can be declared the following way:

```
class Source(Resource):

    class Links:

        class targets:
            target = "foo.bar.Target"
            one_way = True
```

2.5 Link cardinality

More on relationship cardinality - here.

MANY to ONE relationship can be defined this way:

```
class Target(Resource):

    class Links:

        class sources(Link):
            cardinality = Link.cardinalities.MANY # could be ommited - it is the default one
            target = "Source"
            related_name = "target"

class Source(Resource):

    class Links:

        class target(Link):
            cardinality = Link.cardinalities.ONE
            target = "Target"
            related_name = "sources"
            master = True
...
```

ONE to ONE relationship can be defined this way:

```
class Target(Resource):

    class Links:

        class source(Link):
            cardinality = Link.cardinalities.ONE
            target = "Source"
            related_name = "target"

class Source(Resource):

    class Links:

        class target(Link):
            cardinality = Link.cardinalities.ONE
            target = "Target"
            related_name = "source"
            master = True
...
```

MANY to MANY is the default one but explicitly can be defined this way:

```
class Target(Resource):

    class Links:

        class sources(Link):
            cardinality = Link.cardinalities.MANY # could be ommited - it is the default one
            target = "Source"
            related_name = "targets"

class Source(Resource):

    class Links:

        class targets(Link):
            cardinality = Link.cardinalities.MANY # could be ommited - it is the default one
            target = "Target"
            related_name = "sources"
            master = True
...
```

NOTE: relationships with cardinality ONE can be marked as required:

```
class Target(Resource):

    class Links:

        class sources(Link):
            cardinality = Link.cardinalities.MANY # could be ommited - it is the default one
            target = "Source"
            related_name = "target"

class Source(Resource):

    class Links:

        class target(Link):
            cardinality = Link.cardinalities.ONE
            target = "Target"
            related_name = "sources"
            required = True
            master = True
...
```

Relationships with *MANY* cardinality cannot be marked as *required*.

In case of required relationships, data for them must be passed together with main resource data during creation phase.

2.6 Schema and QuerySchema

Resources and Links may define schema that shall be used via Resource API for input validation.

The schema is defined the following way:

```
class CustomResource(Resource):

    class Schema:
        name = schema.StringField(pk=True)
        count = schema.IntegerField()

    class Links:

        class target(Link):
            class Schema:
                timestamp = schema.DateTimeField()
```

Schema fields are defined within a nested class with a reserved name *Schema*. A comprehensive reference for built-in fields can be found *here*.

Additionally both Resources and Links may define query schema to validate all parameters that client uses for filtering the collections.

Query schema is defined the following way:

```
class CustomResource(Resource):

    class QuerySchema:
        name = schema.StringField(pk=True)
        count = schema.IntegerField()

    class Links:

        class target(Link):
            class QuerySchema:
                timestamp = schema.DateTimeField()
```

Query parameters are defined in a similar manner as *Schema* ones but inside *QuerySchema* nested subclass. The key functional difference between two schemas is the fact that *Schema* may have **required** fields and *QuerySchema* may not.

NOTE: it is not necessary for *Schema* and *QuerySchema* inner classes to inherit from *Schema* class. Resource API adds this inheritance automatically.

CHAPTER 3

Service and registry

3.1 Service class

The entity main in Resource API is called *Service*.

class `resource_api.service.`**`Service`**

Entity responsible for holding a registry of all resources that are supposed to be exposed

Service has to be subclassed in order to implement usecase specific *_get_context* and *_get_user* methods.

NOTE: do not override any of the public methods - it may cause framework's misbehavior.

`_get_context`()

MUST BE OVERRIDEN IN A SUBCLASS

Must return an object holding all database connections, sockets etc. It is later on passed to all individual resources.

`_get_user`(*data*)

MUST BE OVERRIDEN IN A SUBCLASS

Must return an object representing currently authenticated user. It is later on passed to individual *can_??* methods of various resources for authorization purposes.

`get_entry_point`(*data*)

Returns `entry point`

data information to be used to construct user object via *_get_user* method

`get_schema`(*human=True*)

Returns schema for all registered resources.

human (bool = True) if True it returns schema with namespaces used during registration if False it returns schema with resource module names as namespaces

`register`(*resource*, *name=None*)

Add resource to the registry

resource (`Resource` subclass) entity to be added to the registry

name (string) string to be used for resource registration, by default it is resource's module name + class name with "." as a delimiter

`setup`()

Finalizes resource registration.

MUST be called after all desired resources are registered.

3.2 Resource registration

Lets say that there are multiple resources declared somewhere. In this case they can be registered the following way:

```
class MultiSQLService(Service):

    def _get_context(self):
        return {
            "db1": create_connection(...),
            "db2": create_connection(...)
        }

srv = MultiSQLService()
srv.register(Student)
srv.register(Teacher)
srv.register(Course)
srv.setup()
```

3.3 Entry point

In order to get access to the *object interface* user must call *get_entry_point* method.

```
entry_point = srv.get_entry_point({"username": "FOO"})
```

Object interface

Object interface is accessible via so called *entry point*. Check *here* to understand how entry points are obtained.

NOTE: Whenever user performs an operation and the operation fails one of *built-in exceptions* is raised.

class `resource_api.service.`**`EntryPoint`**`(`*service*, *user*`)`

Represents user specific means of access to object interface.

`get_resource``(`*resource_class*`)`

resource_class (`Resource` subclass)

```
>>> entry_point.get_resource(Student)
<RootResourceCollection object>
```

`get_resource_by_name``(`*resource_name*`)`

resource_name (string) namespace + ”.” + resource_name, where namespace can be a custom namespace or resource’s module name

```
>>> entry_point.get_resource_by_name("school.Student")
<RootResourceCollection object>
>>> entry_point.get_resource_by_name("com.example.module.education.Student")
<RootResourceCollection object>
```

`user`

User object returned by `Service._get_user` method

4.1 Root resource collection

class `resource_api.resource.`**`RootResourceCollection`**`(`*entry_point*, *resource_interface*, *params=None*`)`

Root resource collection is actually a normal resource collection with two extra methods: *create* and *get*.

`create``(`*data*, *link_data=None*`)`

```
>>> student_collection = entry_point.get_resource(Student)
>>> new_student = student_collection.create({"first_name": "John",
>>>                                          "last_name": "Smith",
>>>                                          "email": "foo@bar.com",
>>>                                          "birthday": "1987-02-21T22:22:22"},
>>>                                         {"courses": [{"@target": "Maths", "grade": 4},
>>>                                                      {"@target": "Sports"}]})
```

get (*pk*)

```
>>> student_collection = entry_point.get_resource(Student)
>>> existing_student = student_collection.get("john@example.com")
```

4.2 Resource collection

class `resource_api.resource.`**`ResourceCollection`** (*entry_point*, *resource_interface*, *params=None*)

The entity that represents a pile of resources.

```
>>> student_collection = entry_point.get_resource(Student)
```

The collection is iterable:

```
>>> for student in student_collection:
>>>     ...
```

If `Resource.get_uris` is implemented to return an indexable entity the collection elements can be accessed by index as well:

```
>>> student = student_collection[15]
```

count ()

Returns count of all items within the system that satisfy filtering criterias.

NOTE: `len(collection)` is supposed to return the same result as `collection.count()`. The key difference between them is that `len` needs to fetch all items in the collection meanwhile `collection.count()` relies on `Resource.get_count`

```
>>> len(student_collection)
4569
>>> student_collection.count()
4569
```

filter (*params=None*)

Filtering options can be applied to collections to return new collections that contain a subset of original items:

NOTE: filtering operations applied to root collections return normal collections

```
>>> student_collection = entry_point.get_resource(Student)
>>> new_collection = student_collection.filter(params={"name__startswith": "Abr"})
```

4.3 Resource item

class `resource_api.resource.`**`ResourceInstance`** (*entry_point*, *resource_interface*, *pk*)

Whenever `creating new or fetching existing` resources resource instances are returned. Resource instances are also returned whenever iterating over `resource collections`.

data

Returns data associated with the resource

```
>>> student.data
{"first_name": "John", "last_name": "Smith", "email": "foo@bar.com", "birthday": "1987-02-21
```

delete()

Removes the resource

```
>>> student.delete()
>>> student.data
...
DoesNotExist: ...
```

links

Returns a `link holder`

pk

Returns PK of the resource

```
>>> student.pk
"foo@bar.com"
```

update(*data*)

Changes specified fields of the resource

```
>>> student.update({"first_name": "Looper"})
>>> student.data
{"first_name": "Looper", "last_name": "Smith", "email": "foo@bar.com", "birthday": "1987-02-
```

4.4 Link holder

class `resource_api.link.`**`LinkHolder`**(*entry_point*, *resource_instance*, *pk*)

Accessor for all the links associated with the resource

For link with cardinality "MANY" `RootLinkCollection` is returned:

```
>>> student.links.courses
<RootLinkCollection object>
```

For link with cardinality "ONE" `LinkToOne` is returned:

```
>>> course.links.teacher
<LinkToOne object>
```

4.5 Root link collection

class `resource_api.link.`**`RootLinkCollection`**(*target_collection*, *forward_link_instance*, *backward_link_instance*, *source_pk*, *params=None*)

Root link collection is actually a normal link collection with two extra methods: *create* and *get*.

create(*data*)

data (dict) has to have at least one key called **@target** - its value must be a PK of target resource instance

```
>>> student_courses = student.links.courses
>>> new_link_to_course = student_courses.create({"@target": "Maths"})
```

get(*target_pk*)

target_pk PK of target resource instance

```
>>> student_courses = student.links.courses
>>> exisiting_link_to_course = student_courses.get("Biology")
```

4.6 Link collection

class `resource_api.link.`**`LinkCollection`**`(`*target_collection*, *forward_link_instance*, *backward_link_instance*, *source_pk*, *params=None*`)`

The entity that represents a pile of resource links.

```
>>> student_courses = student.links.courses
```

The collection is iterable:

```
>>> for link in student_courses:
>>>     ...
```

If `Link.get_uris` is implemented to return an indexable entity the collection elements can be accessed by index as well:

```
>>> link = student_courses[15]
```

`count``()`

Returns count of all items within the system that satisfy filtering criterias.

NOTE: `len(collection)` is supposed to return the same result as `collection.count()`. The key difference between them is that `len` needs to fetch all items in the collection meanwhile `collection.count()` relies on `Link.get_count`

```
>>> len(student_courses)
4569
>>> student_courses.count()
4569
```

`filter``(`*params=None*`)`

Filtering options can be applied to collections to return new collections that contain a subset of original items:

NOTE: filtering operations applied to root collections return normal collections

```
>>> student_courses = student.links.courses
>>> new_link_collection = student_courses.filter(grade__gte=3)
```

4.7 Link instance

class `resource_api.link.`**`LinkInstance`**`(`*target_collection*, *forward_link_instance*, *backward_link_instance*, *source_pk*, *target_pk*`)`

Whenever `creating new or fetching existing` links link instances are returned. Link instances are also returned whenever iterating over `link collections`.

`data`

Returns data associated with the link

```
>>> link.data
{"grade": 3}
```

delete()

Removes the link

```
>>> link.delete()
>>> link.data
...
DoesNotExist: ...
```

target

Returns a `ResourceInstance` associated with target resource.

```
>>> link.target.pk
"Maths"
```

update(*data*)

Changes specified fields of the link

```
>>> link.update({"grade": 4})
>>> link.data
{"grade": 4}
```

NOTE: CANNOT be used to change @*target*

4.8 Link to one

class `resource_api.link.`**LinkToOne**(*target_collection*, *forward_link_instance*, *backward_link_instance*, *source_pk*)

Represents a relationship with cardinality ONE

item

Returns LinkInstance if it exists, raises DoesNotExist error otherwise

```
>>> course.links.teacher.item.delete()
>>> course.links.teacher.item
...
DoesNotExist ...
```

set(*data*)

Does the same thing as `update` method but CAN change the @*target*

```
>>> course.links.teacher.item.target.pk
"Hades"
>>> course.links.teacher.set({"@target": "Zeuz"})
>>> course.links.teacher.item.target.pk
"Zeus"
```

CHAPTER 5

Schema

A collection of fields is represented by:

class `resource_api.schema.`**`Schema`**(*validate_required_constraint=True*, *with_errors=True*)

Base class for containers that would hold one or many fields.

it has one class attribute that may be used to alter shcema's validation flow

has_additional_fields (bool = False) If *True* it shall be possible to have extra fields inside input data that will not be validated

NOTE: when defining schemas do not use any of the following reserved keywords:

- find_fields
- deserialize
- get_schema
- serialize
- has_additional_fields

5.1 Schema example

A schema for a page with title, text, creation timestamp and a 5 star rating would look the following way:

```
class PageSchema(Schema):
    title = StringField(max_length=70)
    text = StringField()
    creation_time = DateTimeField()
    rating = IntegerField(min_value=1, max_value=5)
```

5.2 General field API

All fields inherit from *BaseField* and thus have its attributes in common.

class `resource_api.schema.`**`BaseField`**(*description=None*, *required=True*, ***kwargs*)

Superclass for all fields

description (None|string = None) help text to be shown in schema. This should include the reasons why this field actually needs to exist.

required (bool = False) flag that specifes if the field has to be present

****kwargs** extra parameters that are not programmatically supported

There are two extra parameters supported by Resource API:

readonly (bool=False) if True field cannot be set nor changed but is a logical part of the resource. Resource creation time would be a good example.

changeable (bool=False) if True field can be set during creation but cannot be change later on. User's birth date is a valid example.

5.3 Primitive fields

There are two types of digit fields supported by schema. Integers and floats. Fields that represent them have a common base class:

class `resource_api.schema.`**`DigitField`**(*min_val=None*, *max_val=None*, ***kwargs*)
Base class for fields that represent numbers

min_val (int|long|float = None) Minumum threshold for incoming value

max_val (int|long|float = None) Maximum threshold for imcoming value

The fields representing integers and floats respecively are:

class `resource_api.schema.`**`IntegerField`**(*min_val=None*, *max_val=None*, ***kwargs*)
Transforms input data that could be any number or a string value with that number into *long*

class `resource_api.schema.`**`FloatField`**(*min_val=None*, *max_val=None*, ***kwargs*)
Transforms input data that could be any number or a string value with that number into *float*

Time related fields are represented by:

class `resource_api.schema.`**`DateTimeField`**(*description=None*, *required=True*, ***kwargs*)
datetime object serialized into YYYY-MM-DDThh:mm:ss.sTZD.

E.g.: 2013-09-30T11:32:39.984847

class `resource_api.schema.`**`DateField`**(*description=None*, *required=True*, ***kwargs*)
date object serialized into YYYY-MM-DD.

E.g.: 2013-09-30

class `resource_api.schema.`**`TimeField`**(*description=None*, *required=True*, ***kwargs*)
time object serialized into hh:mm:ssTZD.

E.g.: 11:32:39.984847

class `resource_api.schema.`**`DurationField`**(*description=None*, *required=True*, ***kwargs*)
timedelta object serialized into PnYnMnDTnHnMnS.

E.g.: P105DT9H52M49.448422S

Strings are represented by:

class `resource_api.schema.`**`StringField`**(*regex=None*, *min_length=None*, *max_length=None*, ***kwargs*)
Represents any arbitrary text

regex (string = None) Python regular expression used to validate the string.

min_length (int = None) Minimum size of string value

max_length (int = None) Maximum size of string value

Various boolean flags exist in the schape of:

class `resource_api.schema.`**`BooleanField`**(*default=None*, ***kwargs*)
Expects only a boolean value as incoming data

5.4 Composite fields

class `resource_api.schema.`**`ListField`**(*item_type*, ***kwargs*)
Represents a collection of primitives. Serialized into a list.

item_type (python primitve|Field instance) value is used by list field to validate individual items python primitive are internally mapped to Field instances according to `PRIMITIVE_TYPES_MAP`

```
PRIMITIVE_TYPES_MAP = {
    int: IntegerField,
    float: FloatField,
    str: StringField,
    unicode: StringField,
    basestring: StringField,
    bool: BooleanField
}
```

class `resource_api.schema.`**`ObjectField`**(*schema*, ***kwargs*)
Represents a nested document/mapping of primitives. Serialized into a dict.

schema (class): schema to be used for validation of the nested document, it does not have to be Schema subclass - just a collection of fields

ObjectField can be declared via two different ways.

First, if there is a reusable schema defined elsewhere:

```
>>> class Sample(Schema):
>>>     object_field = ObjectField(ExternalSchema, required=False, description="Zen")
```

Second, if the field is supposed to have a unique custom schema:

```
>>> class Sample(Schema):
>>>     object_field = ObjectField(required=False, description="Zen", schema=dict(
>>>         "foo": StringField()
>>>     ))
```

Built-in exception classes

NOTE: do not raise framework exceptions yourself - otherwise the components might misbehave.

Framework itself raises a bunch of errors that give a descriptive information regarding the status of triggered operations.

NOTE: the framework does not wrap any internal errors within implementations - they are raised without any changes.

exception `resource_api.errors.`**`AuthorizationError`**
Raised when user is not allowed to perform a specific operation with resource instance or resource collection

exception `resource_api.errors.`**`DataConflictError`**
Raised when user tries to perform something that conflicts with a current state of data - create Resource or Link that was already create before

exception `resource_api.errors.`**`DeclarationError`**
Raised by the framework during initialization phase if there are some issues with declarations

exception `resource_api.errors.`**`DoesNotExist`**
Raised when trying to fetch a non-existent Resource or Link instance

exception `resource_api.errors.`**`Forbidden`**
Raised whenever user tries to perform something that is prohibited due to the structure of data - remove required LinkToOne - create one to many link

exception `resource_api.errors.`**`FrameworkError`**
Base class for all the errors that are raised by the framework

exception `resource_api.errors.`**`MultipleFound`**
Raised when user tries to fetch link to one instance and the framework manages to find multiple entries

exception `resource_api.errors.`**`ResourceDeclarationError`**(*resource*, *message*)
Raised by the framework when there are issues with resource declarations

exception `resource_api.errors.`**`ValidationError`**
Raised for any issue related to data sent by user including linking errors

CHAPTER 7

HTTP interface

Resource API can be exposed via HTTP interface with 2nd level of REST Maturity Model

It can be achieved by passing Resource API service instance to `WSGI application`:

```
from werkzeug.serving import run_simple
from custom_app.service import CustomService
from resource_api_http.http import Application

srv = CustomService()
app = Application(srv, debug=True)
run_simple('127.0.0.1', 5000, app, use_debugger=True, use_reloader=True)
```

NOTE: there is no need to call `setup()` method before passing the service to WSGI app. WSGI app shall make the call itself.

7.1 General principles

- structure of the resource is not exposed via URL schema
- links between the resources are exposed via URL schema
- it is not be possible to get links related to the resource via the same url as the resource itself
- there isn't any url to get a collection of the resources' representations. Instead there are URLs to get a list of URIs first and later on use individual HTTP requests to fetch representations of each resource.
- all resources have one and only one URI

7.2 URL schema

Http service has the follwoing URL schema:

```
LINK = RELATIONSHIP
RESOURCE_NAME = NAMESPACE.ACTUAL_RESOURCE_NAME
ID = Individual resource's URI
LINK_NAME = Name of the relationship in a triple
TARGET_ID = URI of the target resource in relationship's triple

# has to contain all required fields
{new_link_data} = {"@target": TARGET_PK, key: value}
```

```
# has to contain only the fields to be changed. Can’t contain "@target".
{partial_link_data} = {key: value}

# has to contain all required fields
{new_resource_data} = {key: value, "@links": {link_to_many_name: [{new_link_data}, ...],
                                              link_to_one_name: {new_link_name}}}
# has to contain only the fields to be changed. Can’t contain "@links".
{partial_resource_data} = {key: value}  # does not have to contain all fields


## Schema

OPTIONS /
>> {service_schema}

## Resource operations

  # create new resource
  POST {new_resource_data} /RESOURCE_NAME
  >> PK, 201 || None, 204

  # get a collection of IDs
  GET /RESOURCE_NAME
  >> [ID1, ID2, ..., IDN], 200

  # get a filtered collection of IDs
  GET /RESOURCE_NAME?query_param=value
  >> [ID1, ID2, ..., IDN], 200

  # get resource’s representation
  GET /RESOURCE_NAME/ID
  >> {key: value}, 200

  # update certain fields of the resource
  PATCH {partial_resource_data} /RESOURCE_NAME/ID
  >> None, 204

  # remove the resource
  DELETE /RESOURCE_NAME/ID
  >> None, 204

  # get number of resources
  GET /RESOURCE_NAME:count
  >> integer count, 200

  # get number of resources with filtering
  GET /RESOURCE_NAME:count?query_param=value
  >> integer count, 200

## Link operations

  ### Link to one operations

    NOTE: "link" string is a part of the URL

    # get target resource’s ID
    GET /RESOURCE_NAME/ID/LINK_NAME/item
    >> TARGET_ID, 200
    # update the link
    PATCH /RESOURCE_NAME/ID/LINK_NAME/item {partial_link_data}
    >> None, 204

    # create a new link or completely overwrite the existing one
    PUT /RESOURCE_NAME/ID/LINK_NAME {new_link_data}
    >> None, 204

    # get data related to the link
    GET /RESOURCE_NAME/ID/LINK_NAME/item:data
    >> {key: value}, 200

    # remove the link
    DELETE /RESOURCE_NAME/ID/LINK_NAME/item
    >> None, 204

  ### Link to many operations

    # get a collection of TARGET_IDs
    GET /RESOURCE_NAME/ID/LINK_NAME
    >> [TARGET_ID1, TARGET_ID2, ...], 200

    # get a filtered collection of TARGET_IDs
    GET /RESOURCE_NAME/ID/LINK_NAME?query_param=value
    >> [TARGET_ID1, TARGET_ID2, ...], 200

    # get number of links
    GET /RESOURCE_NAME/ID/LINK_NAME:count
    >> integer count, 200

    # get number of links with filtering
    GET /RESOURCE_NAME/ID/LINK_NAME:count?query_param=value
    >> integer count, 200

    # create a new link
    POST /RESOURCE_NAME/ID/LINK_NAME {new_link_data}
    >> None, 204

    # get data related to the link
    GET /RESOURCE_NAME/ID/LINK_NAME/TARGET_ID:data
    >> {partial_link_data}, 200

    # update the link
    PATCH /RESOURCE_NAME/ID/LINK_NAME/TARGET_ID {partial_link_data}
    >> None, 204

    # remove the link
    DELETE /RESOURCE_NAME/ID/LINK_NAME/TARGET_ID
    >> None, 204
```

7.3 Error status codes

- **400** in case if request body is invalid
- **403** for any issue related to user authentication/authorization. E.g. if user has no permission to change certain fields.

- **404** if the resource/link being accessed does not exist
- **405** when some HTTP method is not allowed with a specific URL
- **409** when trying to perform the operation that causes conflicts
- **501** when some functionality is not implemented
- **500** when unknown server error takes place

7.4 WSGI Application reference

class `resource_api_http.http.`**`Application`**(*service*, *debug=False*)

Plain WSGI application for Resource API service

service (`Service's` subclass instance) Service to generate HTTP interface for

debug (bool) If True 500 responses will include detailed traceback describing the error

CHAPTER 8

HTTP client

HTTP clinet interface is similar in its design to *object interface*.

class `resource_api_http_client.client.`**`Client`**(*base_url*, *transport_client*)

Client side entry point.

It can be instanciated the following way with a URL of the HTTP service and authentication headers as parameters:

```
>>> client = Client.create(base_url="http://example.com/api",
                           auth_headers={"auth_token": "foo-bar-17"})
```

classmethod **`create`**(*base_url*, *auth_headers=None*)

Instanciates the client

base_url (string) URL of Resource API server (e.g.: "http://example.com/api")

auth_headers (dict || None) Dictionary with fields that are later on used to *construct user object <resource_api.service.Service_get_user>*

`get_resource_by_name`(*resource_name*)

resource_name (string) E.g.: "school.Student"

@return <RootResourceCollection instance>

`schema`

Contains Resource API schema

8.1 Root resource collection

class `resource_api_http_client.client.`**`RootResourceCollection`**(*client*, *name*, *params=None*)

Root resource collection is actually a normal resource collection with two extra methods: *create* and *get*.

`create`(*data*, *link_data=None*)

```
>>> student_collection = client.get_resource_by_name("school.Student")
>>> new_student = student_collection.create({"first_name": "John", "last_name": "Smith", "em
>>>                                           "birthday": "1987-02-21T22:22:22"})
```

`get`(*pk*)

```
>>> student_collection = client.get_resource_by_name("school.Student")
>>> existing_student = student_collection.get("john@example.com")
```

8.2 Resource collection

class `resource_api_http_client.client.`**`ResourceCollection`**(*client*, *name*, *params=None*)

The entity that represents a pile of resources.

```
>>> student_collection = client.get_resource_by_name("school.Student")
```

The collection is iterable:

```
>>> for student in student_collection:
>>>     ...
```

`count`()

Returns count of all items within the system that satisfy filtering criterias.

NOTE: `len(collection)` is supposed to return the same result as `collection.count()`. The key difference between them is that `len` needs to fetch all items in the collection meanwhile `collection.count()` relies on **/<ResourceName>:count** URL

```
>>> len(student_collection)
4569
>>> student_collection.count()
4569
```

`filter`(*params=None*)

Filtering options can be applied to collections to return new collections that contain a subset of original items:

NOTE: filtering operations applied to root collections return normal collections

```
>>> student_collection = client.get_resource_by_name("school.Student")
>>> new_collection = student_collection.filter(params={"name__startswith": "Abr"})
```

8.3 Resource item

class `resource_api_http_client.client.`**`ResourceInstance`**(*client*, *name*, *pk*, *data=None*)

Whenever `creating new or fetching existing` resources resource instances are returned. Resource instances are also returned whenever iterating over `resource collections`.

`data`

Returns data associated with the resource

```
>>> student.data
{"first_name": "John", "last_name": "Smith", "email": "foo@bar.com", "birthday": "1987-02-21
```

`delete`()

Removes the resource

```
>>> student.delete()
>>> student.data
...
DoesNotExist: ...
```

links
Returns a link holder

pk
Returns PK of the resource

```
>>> student.pk
"foo@bar.com"
```

update (*data*)
Changes specified fields of the resource

```
>>> student.update({"first_name": "Looper"})
>>> student.data
{"first_name": "Looper", "last_name": "Smith", "email": "foo@bar.com", "birthday": "1987-02-
```

8.4 Link holder

class `resource_api_http_client.client.`**`LinkHolder`**(*client*, *url*, *schema*)
Accessor for all the links associated with the resource

For link with cardinality “MANY” `RootLinkCollection` is returned:

```
>>> student.links.courses
<RootLinkCollection object>
```

For link with cardinality “ONE” `LinkToOne` is returned:

```
>>> course.links.teacher
<LinkToOne object>
```

8.5 Root link collection

class `resource_api_http_client.client.`**`RootLinkCollection`**(*client*, *base_url*, *target_name*, *params=None*)
Root link collection is actually a normal link collection with two extra methods: *create* and *get*.

create (*data*)

data (dict) has to have at least one key called **@target** - its value must be a PK of target resource instance

```
>>> student_courses = student.links.courses
>>> new_link_to_course = student_courses.create({"@target": "Maths"})
```

get (*target_pk*)

target_pk PK of target resource instance

```
>>> student_courses = student.links.courses
>>> exisiting_link_to_course = student_courses.get("Biology")
```

8.6 Link collection

class `resource_api_http_client.client.`**`LinkCollection`**`(`*client*, *base_url*, *target_name*, *name*, *params=None*`)`

The entity that represents a pile of resource links.

```
>>> student_courses = student.links.courses
```

The collection is iterable:

```
>>> for link in student_courses:
>>>     ...
```

Accessing items by index is also possible: >>> link = student_courses[15]

`count``()`

Returns count of all items within the system that satisfy filtering criterias.

NOTE: `len(collection)` is supposed to return the same result as `collection.count()`. The key difference between them is that `len` needs to fetch all items in the collection meanwhile `collection.count()` relies on **/<ResourceName>:count** URL

```
>>> len(student_courses)
4569
>>> student_courses.count()
4569
```

`filter``(`*params=None*`)`

Filtering options can be applied to collections to return new collections that contain a subset of original items:

NOTE: filtering operations applied to root collections return normal collections

```
>>> student_courses = student.links.courses
>>> new_link_collection = student_courses.filter(grade__gte=3)
```

8.7 Link instance

class `resource_api_http_client.client.`**`LinkInstance`**`(`*client*, *base_url*, *target_name*, *target_pk*, *data=None*, *unique=False*`)`

Whenever `creating new or fetching existing` links link instances are returned. Link instances are also returned whenever iterating over `link collections`.

`data`

Returns data associated with the link

```
>>> link.data
{"grade": 3}
```

`delete``()`

Removes the link

```
>>> link.delete()
>>> link.data
...
DoesNotExist: ...
```

target
Returns a `ResourceInstance` associated with target resource.

```
>>> link.target.pk
"Maths"
```

8.8 Link to one

class `resource_api_http_client.client.`**`LinkToOne`**(*client*, *base_url*, *target_name*, *name*)
Represents a relationship with cardinality ONE

item
Returns LinkInstance if it exists, raises DoesNotExist error otherwise

```
>>> course.links.teacher.item.delete()
>>> course.links.teacher.item
...
DoesNotExist ...
```

set(*data*)
Does the same thing as `update` method but CAN change the @*target*

```
>>> course.links.teacher.item.target.pk
"Hades"
>>> course.links.teacher.set({"@target": "Zeuz"})
>>> course.links.teacher.item.target.pk
"Zeus"
```

Descriptor (schema)

Descriptor is Resource API's way to notify the client about the structure of resources and relationships between them.

Lets say that there is the following collection of resources:

```
class User(Resource):
    """ Represents a person who uses the system """

    class Schema:
        email = StringField(regex="[^@]+@[^@]+\.[^@]+", pk=True)
        name = StringField(max_length=70, description="First name and last name")

    class Links:

        class cars(Link):
            """ All cars that belong to the user """
            target = "Car"
            related_name = "owner"

class Car(Resource):
    """ The item being sold/bought within the system """

    class Schema:
        pk = IntegerField(pk=True, description="Integer identifier")
        model = StringField(description="E.g. BMW")
        brand = StringField(description="E.g. X6")
        year_of_production = DateTimeField(
            description="Time when the car was produced to estimate its age")

    class QuerySchema:
        age = IntegerField(description="Age of the car in years")

    class Links:

        class owner(Link):
            """ User who owns the car """
            target = "User"
            related_name = "cars"
            cardinality = Link.cardinalities.ONE
            master = True

            class Schema:
                acquisition_data = DateTimeField(
                    description="Time when the car changed its owner")
```

And they are registered:

```
srv = Service()
srv.register(User, "auth.User")
srv.register(Car, "shop.Car")
srv.setup()
```

In this case the descriptor shall look like:

```
{
   "shop.Car": {
       "pk_policy": {
           "description": " Uses value of a field marked as \"pk=True\" as resource’s URI "
       },
       "description": " The item being sold/bought within the system ",
       "links": {
           "owner": {
               "related_name": "cars",
               "description": " User who owns the car ",
               "required": false,
               "target": "auth.User",
               "cardinality": "ONE",
               "schema": {
                   "acquisition_data": {
                       "type": "datetime",
                       "description": "Time when the car changed its owner"
                   }
               }
           }
       },
       "schema": {
           "pk": {
               "pk": true,
               "type": "int",
               "description": "Integer identifier"
           },
           "brand": {
               "type": "string",
               "description": "E.g. X6"
           },
           "model": {
               "type": "string",
               "description": "E.g. BMW"
           },
           "year_of_production": {
               "type": "datetime",
               "description": "Time when the car was produced to estimate its age"
           }
       },
       "query_schema": {
           "age": {
               "type": "int",
               "description": "Age of the car in years"
           }
       }
       }
   },
   "auth.User": {
       "pk_policy": {
```

```
                "description": " Uses value of a field marked as \"pk=True\" as resource’s URI "
            },
            "description": " Represents a person who uses the system ",
            "links": {
                "cars": {
                    "related_name": "owner",
                    "description": " All cars that belong to the user ",
                    "required": false,
                    "target": "shop.Car",
                    "cardinality": "MANY",
                    "schema": {
                        "acquisition_data": {
                            "type": "datetime",
                            "description": "Time when the car changed its owner"
                        }
                    }
                }
            },
            "schema": {
                "email": {
                    "regex": "[^@]+@[^@]+\\.[^@]+",
                    "pk": true,
                    "type": "string",
                    "description": null
                },
                "name": {
                    "max_length": 70,
                    "type": "string",
                    "description": "First name and last name"
                }
            }
        }
}
```

As it can be seen, the descriptor is a one to one mapping of the structure declared in python to JSON document.

There is a couple of things to note about the descriptor:

- **description** fields are generated from resources’/links’ docstrings and **description** argument of schema fields. If one of them is missing **description** is intentionally marked as *null*.
- **target** corresponds to the name that was used when registering a specific resource

Python Module Index

Index