Kim Documentation

Release 1.0.0

Mikey Waites
Jack Saunders

March 09, 2018

Contents

Release v1.0.0. (*Installation*)

Introducing Kim:

```
>>> mapper = UserMapper(data=response.json())
>>> mapper.marshal()
User(id='one', name='Bruce Wayne', 'title'='CEO/Super Hero')
>>> user_two = User.query.get('two')
>>> mapper = UserMapper(obj=user_two)
>>> mapper.serialize()
{u'id': 'two', u'name': 'Martha Wayne', 'title': 'Mother of Batman'}
```

CHAPTER 1

Kim Features

Kim is a feature packed framework for handling even the most complex marshaling and serialization requirements.

- Web framework agnostic - Flask, Django, Framework-XXX supported!
- Highly customisable field processing system
- Security focused
- Control included fields with powerful roles system
- Handle mixed data types with polymorphic mappers
- Marshal and Serialize nested objects

Kim officially supports Python 2.7 & 3.3–3.5

CHAPTER 2

The User Guide

Learn all of Kim's features with these simple step-by-step instructions or check out the quickstart guide for a rapid overview to get going quickly.

Introduction

Why Kim?

Installation

This part of the documentation covers the installation of Kim.

Installation via Pip

To install Kim, simply run this command in your terminal of choice:

```
$ pip install py-kim
```

Quickstart

Eager to get going? This page gives an introduction to getting started with Kim.

First, make sure that:

- Kim is *installed*

Defining Mappers

Let's start by defining some mappers. Mappers are the building blocks of kim - They define how JSON output should look and how input JSON should be expected to look.

Mappers consist of Fields. Fields define the shape and nature of the data both when being serialised(output) and marshaled(input).

Mappers must define a `__type__`. This is the python type that will be instantiated if a new object is marshaled through the mapper. `__type__` may be be any object that supports `getattr` and `setattr`, or any dict like object.

```
from kim import Mapper, field

class CompanyMapper(Mapper):
        __type__ = Company
        id = field.String(read_only=False)
        name = field.String()

class UserMapper(Mapper):
        __type__ = User
        id = field.String(read_only=False)
        name = field.String()
        company = field.Nested(CompanyMapper, read_only=True)
```

Further Reading:

- *Defining Mappers - Advanced*
- *Polymorphic Mappers*

Serializing Data

Now we have a mapper defined we can start serializing some objects. To serialize an object we simply pass it to our mapper using the `obj` kwarg.

```
>>> user = get_user()
>>> mapper = UserMapper(obj=user)
>>> mapper.serialize()
{'name': 'Bruce Wayne', 'id': 1, 'company': {'name': 'Wayne Enterprises', 'id': 1}}
```

Serializing Many objects

We can also handle serializing lots of objects at once. Each mapper represents a single datum. When serializing more than one object we use the classmethod `many` from the mapper.

```
>>> user = get_users()
>>> mapper = UserMapper.many(obj=user).serialize()
[{'name': 'Bruce Wayne', 'id': 1, 'company': {'name': 'Wayne Enterprises', 'id': 1}}
{'name': 'Martha Wayne', 'id': 2, 'company': {'name': 'Wayne Enterprises', 'id': 1}}]
```

Further Reading:

- *Custom serialization Pipelines*

Marshaling Data

We've seen how we to serialize our objects back into dicts. Now we want to be able to marshal incoming data into the `__type__` defined on our mappeer. When using our mapper to marshal data, we pass the `data` kwarg.

```
>>> data = {'name': 'Tony Stark'}
>>> mapper = UserMapper(data=data)
>>> mapper.marshal()
User(name='Tony Stark', id=3)
```

As you can see the data we passed the mapper has been converted into our User type.

Marshaling Many Objects

Many objects can be marshaled at once using the `many` method from our mapper.

```
>>> data = [{'name': 'Tony Stark'}, {'name': 'Obadiah Stane'}]
>>> mapper = UserMapper.many(data=data).marshal()
[User(name='Tony Stark', id=3), User(name='Obadiah Stane', id=4)]
```

Handling Validation Errors

When Marshaling, Kim will apply validation via the fields you have used to define your mapper. Field validation and data pipelines are covered in detail in the advanced section, but here's a simple example of handling the errors raised when marshaling.

```
from kim import MappingInvalid

data = {'name': 'Tony Stark'}
mapper = UserMapper(data=data)

try:
        mapper.marshal()
except MappingInvalid as e:
        print(e.errors)
```

Updating Existing Objects

We won't always want to create new objects when marshaling data - Kim supports updating existing objects as well. This is achieved by passing the the existing `obj` to the mapper along with the new data. As with normal marshaling, Kim will raise an error for any missing required fields.

```
>>> obj = User.query.get(2)
>>> data = {'name': 'New Name', 'title': 'New Guy'}
>>> mapper = UserMapper(obj=obj, data=data)
>>> mapper.marshal()
User(name='New Name', id=2, title='New Guy')
```

Partial Updates

We can also partially update objects. This means Kim will not raise an error when required fields are missing from the data passed to the mapper and will instead only process fields that are present in the data provided. This is useful for PATCH requests in a REST API. We pass the *partial=True* kwarg to the Mapper to indicate this is a partial update.

```
>>> obj = User.query.get(4)
>>> data = {'title': 'Super Villain'}
>>> mapper = UserMapper(obj=obj, data=data, partial=True)
>>> mapper.marshal()
User(name='Obadiah Stane', id=4, title='Super Villain')
```

Further Reading:

- *Custom marshaling Pipelines*

Nesting Objects

We have already seen how to define a nested object on one of our mappers. Nesting allows us to specify other mappers that represent nested objects within our data structures. As you can see below, when we serialize our User object Kim also serializes the user's company for us too.

```
>>> user = get_user()
>>> mapper = UserMapper(obj=user)
>>> mapper.serialize()
{'name': 'Bruce Wayne', 'id': 1, 'company': {'name': 'Wayne Enterprises', 'id': 1}}
```

Marshaling Nested Objects

Our Nested company object is specified as `read_only=True`. This means Kim will ignore any data present for that field when marshaling. To demonstrate marshaling with a Nested object let's first add a new field to our UserMapper.

```
from kim import Mapper
from kim import field

def user_getter(session):
"""Fetch a user by id from json data
"""
if session.data and 'id' in session.data:
    return User.get_by_id(session.data['id'])

class CompanyMapper(Mapper):
        __type__ = Company
        id = field.String(read_only=False)
        name = field.String()

class UserMapper(Mapper):
        __type__ = User
        id = field.String(read_only=False)
        name = field.String()
        company = field.Nested(CompanyMapper, read_only=True)
        sidekick = field.Nested('UserMapper', required=False, getter=user_getter)
```

Note: Nested mappers can be passed as a string class name as well as a mapper class directly.

A few things have happened here. We have added another Nested field but this time we've also specified a `getter` kwarg. The getter function will be called when we pass a nested object to the User mapper for the mapper to marshal.

A getter function is responsible for taking the data passed into the nested object and returning another type, typically a database object. If the object is not found or not permitted to be accessed, it should return None, which will cause a validation error to be raised.

The role of Nested getter functions is to provide a simple point at which you can validate the authenticity of the data before inflating it into a nested object. It also means that virtually any datastore can be used to expand nested objects.

```
>>> data = {'name': 'Tony Stark', 'sidekick': {'id': 5, 'name': 'Pepper Potts'}}
>>> mapper = UserMapper(data=data)
>>> obj = mapper.marshal()
>>> obj
User(name='Tony Stark', id=3)
>>> obj.sidekick
User(name='Pepper Potts', id=5)
```

Further Reading:

- *Nested fields*

Roles: Changing the shape of the data

Kim provides a powerful system for controlling what fields are available during marshaling and serialization called *roles*. Roles are defined against a *Mapper* and can be provided as a `whitelist` set of permitted fields or a `blacklist` set of private fields. (It's also possible to combine the two concepts which is covered in more detail in the advanced section).

To define roles on your mapper use the `__roles__` property.

```
from kim import Mapper, field, whitelist, blacklist

class CompanyMapper(Mapper):
        __type__ = Company
        id = field.String(read_only=False)
        name = field.String()

class UserMapper(Mapper):
        __type__ = User
        id = field.String(read_only=False)
        name = field.String()
        company = field.Nested(CompanyMapper, read_only=True)

        __roles__ = {
            'id_only': whitelist('id'),
            'public': blacklist('id')
        }
```

We've defined two roles on our UserMapper. These roles can now be used when marshaling and serializing by passing the `role` kwargs to the methods *kim.mapper.Mapper.serialize* or *kim.mapper.Mapper.marshal*.

Let's use the `id_only` role to serialize a user and only return the id field.

```
>>> user = get_user()
>>> mapper = UserMapper(obj=user)
>>> mapper.serialize(role='id_only')
{'id': 1}
```

Next Steps

The quickstart covers the bare minimum to give you a basic understanding of how to use Kim. Kim offers heaps more functionality so why not head over to the *Advanced Section* to read more about all of Kim's features.

Advanced Topics

This section gives a more detailed explanation of the features of Kim. If you're looking for a quick overview or if this is your first time using Kim, please check out the *quickstart guide*.

Mappers

Polymorphic Mappers

It's not uncommon to have collections of objects that are not all the same. Perhaps you have an `Activity` type that has two sub types `Task` and `Event`. Their serialization requirements differ slightly meaning you'd typically serialize two lists and manually munge them together.

Kim provides support for Polymorphic Mapper to solve this problem.

Polymorphic Mappers are defined like a normal mapper with a few small differences. Firstly we define our base "type". This is the Mapper all of our Polymorphic types extend from. Our base type should inherit from *kim.mapper.PolymorphicMapper* instead of *kim.mapper.Mapper*.

```
from kim import PolymorphicMapper, field

class ActivityMapper(PolymorphicMapper):

    __type__ = Activity

    id = field.String()
    name = field.String()
    object_type = field.String(choices=['event', 'task'])
    created_at = field.DateTime(read_only=True)

    __mapper_args__ = {
        'polymorphic_on': object_type,
    }
```

For users of SQLAlchemy, this API will feel very familiar. We've specified our base mapper with the `__mapper_args__` property. The `polymorphic_on` key is given a referrence to the field used to indentify our polymorphic types. This can also be passed as a string.

```
__mapper_args__ = {
    'polymorphic_on': 'object_type'
}
```

Now we need to define our types.

```
class TaskMapper(ActivityMapper):

    __type__ = Task

    status = field.String(read_only=True)
    is_complete = field.Boolean()

    __mapper_args__ = {
        'polymorphic_name': 'task'
    }


class EventMapper(ActivityMapper):

    __type__ = Event

    location = field.String(read_only=True)

    __mapper_args__ = {
        'polymorphic_name': 'event'
    }
```

Our types inherit from our base `ActivityMapper` and also specify the `__mapper_args__` property. Our types provide the `polymorphic_name` key which indentifies the type to the base mapper.

Serializing Polymorphic Mappers

Serializing Polymorphic Mappers works in the same way as serializing a normal Mapper. When we want to serialize a collection of mixed types we serialzie using the base mapper.

```
>>> activities = Activity.query.all()
>>> ActivityMapper.many(obj=activities).serialize()
[
    {'name': 'My Test Event', 'id': 1, 'object_type': 'event', 'created_at': '2017-03-11T05:14:43+00
    {'name': 'My Test Task', 'id': 1, 'object_type': 'task', 'created_at': '2016-03-11T05:14:43+00:00
]
```

As you would expect, serializing using one of the child types directly will only serialize its own type.

```
>>> activities = Event.query.all()
>>> EventMapper.many(obj=activities).serialize()
[
    {'name': 'My Test Event', 'id': 1, 'object_type': 'event', 'created_at': '2017-03-11T05:14:43+00
]
```

Marshaling Polymorphic Mappers

Marshaling Polymorphic Mappers is also supported but is disabled by default. It is currently considered an experimental feature.

To enable marshaling for Polymorphic Mappers we pass `allow_polymorphic_marshal: True` to the `__mapper_args__` property on the base Polymorphic Mapper.

```
class ActivityMapper(PolymorphicMapper):

    __type__ = Activity

    id = field.String()
    name = field.String()
    object_type = field.String(choices=['event', 'task'])
    created_at = field.DateTime(read_only=True)

    __mapper_args__ = {
        'polymorphic_on': object_type,
        'allow_polymorphic_marshal': True,
    }
```

We can now marshal a collection of mixed object types using the base ActivityMapper.

```
data = [
    {'name': 'My Test Event', 'object_type': 'event', 'created_at': '2017-03-11T05:14:43+00:00', 'loc
    {'name': 'My Test Task', 'object_type': 'task', 'created_at': '2016-03-11T05:14:43+00:00', 'statu
]
>>> ActivityMapper.many(obj=activities).marshal()
[Event(name='My Test Event'), Task(name='My Test Task')]
```

Exception Handling

Kim uses custom exceptions when marshaling to allow you to get at all the errors that ocurred as a result of processing the fields in your mappers marshaling pipeline.

Each pipe in a field's pipeline can raise a *kim.exception.FieldInvalid*. As the pipeline is processed the errors for the field will be stored against the mapper. Once all the fields have been processed the mapper checks to see if any errors occurred. If there are any errors the mapper will raise a *kim.exception.MappingInvalid*.

You should typically only worry about handling the *kim.exception.MappingInvalid* when marshaling.

```
from kim import MappingInvalid

try:
    data = mapper.marshal()
except MappingInvalid as e:
    print(e.errors)
```

The *kim.exception.MappingInvalid* exception raised will have an attribute called errors. Errors is a dictionary containing `field_name: error message`. The errors object can also contain nested error objects when marshaling a *kim.field.Nested* field fails.

Roles

As described in the quickstart, the Roles system provides users with a system for controlling what fields are available during marshaling and serialization.

Role Inheritance

Mappers inherit Roles from their parents automatically. Consider the following example.

```
class MapperA(Mapper):

    __type__ = dict

    field_a = field.String()
    field_b = field.String()

    __roles__ = {
        'ab': whitelist('field_a', 'field_b')
    }


class MapperB(MapperA):

    field_c = field.String()

    __roles__ = {
        'abc': blacklist()
    }
```

MapperB inherits from MapperA and therefore will have access to all the roles defined on MapperA. Equally, MapperB can define the role `ab` to override the fields available for that role.

Combining Roles

Under the hood *kim.role.Role* is a set object. This allows us to combine roles in the ways that sets can be combined. This is useful when you have a role defined on a base type that you need to extend.

When combining whitelist and blacklist roles the order is not important. The blacklist always takes priority. The following examples are equal.

```
>>> role = blacklist('name', 'id') | whitelist('name', 'email')
>>> assert 'email' in role
>>> assert 'name' not in role
>>> assert 'id' not in role
>>> assert role.whitelist

>>> role = whitelist('name', 'id') | blacklist('name', 'email')
>>> assert 'id' in role
>>> assert 'name' not in role
>>> assert 'email' not in role
>>> assert role.whitelist
```

Default Roles

Every mapper has a special hidden role called `__default__`. By default the `__default__` role contains every field defined on your Mapper.

You can override the `__default__` role by specifying it in the `__roles__` property on your Mapper.

```
class MapperA(Mapper):

    __type__ = dict

    field_a = field.String()
    field_b = field.String()

    __roles__ = {
        '__default__': whitelist('field_a')
    }
```

Now whenever we call *kim.mapper.Mapper.marshal* or *kim.mapper.Mapper.serialize* on MapperA without a role, the default role will be used which now only includes field_a.

Note: The __default__ role does not currently inherit from it's parent and must be defined explitly on the all Mappers in the class heirarchy.

Fields

Name and Source

If you'd like the field in your JSON data to have a different name to the field on the object, pass the `source` attribute to `Field`.

```
from kim import Mapper, field

class CompanyMapper(Mapper):
    __type__ = Company
```

```
    title = field.String(source='name')

>>> company = Company(name='Wayne Enterprises')
>>> mapper = CompanyMapper(company)
>>> mapper.serialize()
{'title': 'Wayne Enterprises'}
```

Note: When marshaling, Kim will look for data in the field named in source

Similarly, if you'd like the JSON data to have a different name to the attribute name on the mapper class, pass the `name` attribute to `Field`. This is useful if you have multiple fields in different roles which should serialize to the same field.

```
from kim import Mapper, field, role

class CompanyMapper(Mapper):
    __type__ = Company
    short_title = field.String(name='title')
    long_title = field.String(name='title')

    __roles__ = {
        'simple': role.whitelist('short_title'),
        'full': role.whitelist('long_title')
    }


>>> company = Company(short_title='Wayne', long_title='Wayne Enterprises')
>>> mapper = CompanyMapper(company)
>>> mapper.serialize(role='simple')
{'title': 'Wayne'}
>>> mapper.serialize(role='full')
{'title': 'Wayne Enterprises'}
```

Nested `__self__`

Sometimes your object model may contain flat data but you'd like the JSON output to be nested. You can do this by setting `source='__self__'` on a Nested field.

```
from kim import Mapper, field, role

class AddressMapper(Mapper):
    __type__ = dict

    street = field.String()
    city = field.String()
    zip = field.String()

class CompanyMapper(Mapper):
    __type__ = Company

    name = field.String()
    address = field.Nested(AddressMapper, source='__self__')

>>> company = Company(
    title='Wayne Enterprises',
    street='4 Maple Road',
```

```
    city='Sunview',
    zip='90210')
>>> mapper = CompanyMapper(company)
>>> mapper.serialize()
{'name': 'Wayne Enterprises',
 'address': {'street': '4 Maple Road', 'city': 'Sunview', 'zip': '90210'}}
```

In this example, the address appears as a nested object in the JSON, but it's fields are all sourced from company.

Note: `__self__` can also be used to marshal nested objects into flat structures

---

Marshaling Nested Fields

Nested fields can be marshaled in a similar manner to serializing, but there are several security concerns you should take into account when using them. Kim's settings default to the most secure and must be overridden to use the full functionality.

Note: This section, and Kim's defaults, assume you are using nested fields to refer to foreign keys (or similar NoSQL relationships) on ORM objects. If you are not using Kim with an ORM, you probably want to enable the `allow_create` and `allow_updates_in_place` options for seamless operation.

In general, there are four things you may want to happen when marshaling a nested field. The following sections describe them, and the input data they expect.

For all examples, assume the Mapper looks like this:

```
from kim import Mapper

class UserMapper(Mapper):
    __type__ = MyUser

    id = field.Integer(read_only=True)
    name = field.String(required=True)
    company = field.Nested('CompanyMapper')  # Set options on this field
```

1. Retrieve by ID only (default)

```
{'id': 1,
 'name': 'Bob Jones',
 'company': {
    'id': 5,  # Will be used to look up Company
    # Any other data here will be ignored
 }}
```

This is the most secure option and the most common thing you will want to do. This means that only the ID of the target object will be used, a `getter` function which you define will be used to retrieve the object with this ID from your database (taking into account security such as ensuring the user has access to the object), and the object returned from the `getter` function will be set on the target attribute.

2. `allow_updates` - Retrieve by ID, allowing updates

```
{'id': 1,
 'name': 'Bob Jones',
 'company': {
    'id': 5,  # Will be used to look up Company
    'name': 'New name',  # Will be set on the Company
 }}
```

This option retrieves the related object via it's ID using a `getter` function as in scenario 1. However, any other fields passed along with the ID will be updated on the related object, according to the role passed. You are strongly encouraged to only use this option with a restrictive role, in order to avoid introducing security holes where users can change fields on objects they should not be able to do, (for example, change the `user` field on an object to change it's ownership).

Use this option like this (`role` is not required):

```
company = field.Nested('CompanyMapper', allow_updates=True, role='restrictive_role')
```

3. `allow_create` - Retrieve by ID, or create object if no ID passed

```
# No ID passed - create new
{'id': 1,
 'name': 'Bob Jones',
 'company': {
    'name': 'My new company',  # Will be set on the new company
 }}
 # ID passed - works as scenario 1
 {'id': 1,
  'name': 'Bob Jones',
  'company': {
     'id': 5,  # Will be used to look up company
     # Any other data here will be ignored
  }}
```

This option uses your `getter` function to look up the related object by ID, but if it is not found (ie. your getter function returns `None`) then a new instance of the object will be created, using the fields passed according to the role.

This option may be combined with `allow_updates` in order to provide a field which will accept an existing object, allow it to be updated and allow a new one to be created.

Once again, you should consider carefully the role you use with this option to avoid unexpected consequences (for example, it being possible to set the `user` field on an object to someone other than the logged-in user.)

Use this option like this (`role` is not required):

```
company = field.Nested('CompanyMapper', allow_create=True, role='restrictive_role')
```

4. `allow_updates_in_place` - Do not use ID, update existing related object

```
# No ID passed - update the existing object if it exists
{'id': 1,
 'name': 'Bob Jones',
 'company': {
    # No ID is required here
```

```
    'name': 'New name',  # Will be updated on existing company
}}
```

In this scenario, no ID field is required and no `getter` function is used. Instead, the fields are simply updated on the existing value of `user.company`, if it exists.

Collections

Collections are used to produce arrays of similar fields in the JSON output. They can be scalar fields or nested fields and work when serializing or marshaling.

To create a collection, wrap any field in `Collection`:

```
from kim import Mapper, field, role


class CompanyMapper(Mapper):
    __type__ = Company

    name = field.String()
    offices = field.Collection(field.String())

>>> mapper = CompanyMapper(company)
>>> mapper.serialize()
{'name': 'Wayne Enterprises',
 'offices': ['London', 'Berlin', 'New York']}
```

You can also wrap nested fields:

```
from kim import Mapper, field, role

class EmployeeMapper(Mapper):
    __type__ = Employee

    name = field.String()
    job = field.String()


class CompanyMapper(Mapper):
    __type__ = Company

    name = field.String()
    employees = field.Collection(field.Nested(EmployeeMapper))

>>> mapper = CompanyMapper(company)
>>> mapper.serialize()
{'name': 'Wayne Enterprises',
 'employees': [
    {'name': 'Jim', 'job': 'Developer'},
    {'name': 'Bob', 'job': 'Manager'},
]}
```

When marshaling, Nested fields can be forced to be unique on a key to avoid duplicates:

```
from kim import Mapper, field, role

class EmployeeMapper(Mapper):
    __type__ = Employee
```

```
    id = field.Integer()
    name = field.String()


class CompanyMapper(Mapper):
    __type__ = Company

    name = field.String()
    employees = field.Collection(
        field.Nested(EmployeeMapper), unique_on='id')

>>> data = {'employees': [{'id': 1, 'name': 'Jim'}, {'id': 1, 'name': 'Bob'}]}
>>> mapper = CompanyMapper(data=data)
>>> mapper.marshal()
MappingInvalid
```

Pipelines

Fields process their data through a series of pipes, called a pipeline. A pipe is passed some data, performs one operation on it and returns the new data. This is then passed to the next pipe in the chain. This concept is similar to Unix pipes.

There are separate pipelines for serializing and marshaling.

For example, here is the marhal pipeline for the `String` field. Pipes are grouped into four stages - input, validation, process and output.

```
input_pipes = [read_only, get_data_from_name]
validation_pipes = [is_valid_string, is_valid_choice, ]
process_pipes = []
output_pipes = [update_output_to_source]

# Order of execution is:
read_only ->                  # Stop execution if field is ready only
get_data_from_name ->         # Get the data for this field from the JSON
is_valid_string ->            # Raise exception if data is not a string
is_valid_choice ->            # If choices=[] set on field, raise exception if not valid choice
update_output_to_source ->    # Update the object with this data
```

Custom Fields and Pipelines

To define a custom field, you need to create the Field class and its corresponding Pipline. It's usually easiest to inherit from an existing Field/Pipeline, rather than defining an entirely new one.

This example defines a new field with a custom pipeline to convert its output to uppercase:

```
from kim import pipe, String, Mapper
from kim.pipelines.string import StringSerializePipeline


@pipe()
def to_upper(session):
    if session.data is not None:
        session.data = session.data.upper()
    return session.data

class UpperCaseStringSerializePipeline(StringSerializePipeline):
```

```
    process_pipes = StringSerializePipeline.process_pipes + [to_upper]

class UpperCaseString(String):
    serialize_pipeline = UpperCaseStringSerializePipeline

class MyMapper(Mapper):
    __type__ = dict

    name = UpperCaseString()
```

Note: This is a contrived example, for simple transforms like this see `extra_marshal_pipelines` below

Note that we have only overridden the `process_pipes` stage of StringSerializePipeline. Everything else remains the same. We have extended the `process_pipes` list from the parent object in order to retain it's functionality, and just added our new pipe at the end.

Pipes should find and set their data on `session.data`. The session object also provides access to the field, the current output object, the parent field (if nested) and the mapper. See the API docs for details.

Custom Validation - extra_marshal_pipes

If you just want to change the pipeline used by a particular instance of a Field on a Mapper, for example to add custom validation logic, you don't need to define an entirely new field. Instead you can pass `extra_marshal_pipes`:

`extra_marshal_pipes` are pushed onto the existing list of pipes defined on the field at compile time once each time a Field is instantiated.

```
from kim import Mapper, String, Integer, pipe


@pipe()
def check_age(session):
    if session.data is not None and session.data < 18:
        raise session.field.invalid('not_old_enough')

    return session.data


class MyMapper(Mapper):
    __type__ = dict

    name = String()
    age = Integer(
        extra_marshal_pipes={
            'validation': [check_age],
        },
        error_msgs={'not_old_enough': 'You must be over 18'}
    )
```

`extra_marshal_pipes` takes a dict of the format `{stage: [pipe, pipe, pipe]}`. Any pipes pased will be added at the end of their respective stage.

CHAPTER 3

The API Documentation / Guide

Detailed class and method documentation

Developer Interface

This part of the documentation covers all the interfaces of Kim.

Mappers

class kim.mapper.**Mapper**(*obj=None*, *data=None*, *partial=False*, *raw=False*, *parent=None*)

Mappers are the building blocks of Kim - they define how JSON output should look and how input JSON should be expected to look.

Mappers consist of Fields. Fields define the shape and nature of the data both when being serialised(output) and marshaled(input).

Mappers must define a __type__. This is the type that will be instantiated if a new object is marshaled through the mapper. __type__ may be be any object that supports getattr and setattr, or any dict like object.

Usage:

```
from kim import Mapper, field

class UserMapper(Mapper):
    __type__ = User

    id = field.Integer(read_only=True)
    name = field.String(required=True)
    company = field.Nested('myapp.mappers.CompanyMapper')
```

Initialise a Mapper with the object and/or the data to be serialzed/marshaled. Mappers must be instantiated once per object/data. At least one of obj or data must be passed.

Parameters

- **obj** – the object to be serialized, or updated by marshaling
- **data** – input data to be used for marshaling
- **raw** – the mapper will instruct fields to populate themselves using __dunder__ field names where required.

- **partial** – allow pipelines to pull data from an existing source or fall back to standard checks.
- **parent** – The parent of this Mapper. Set internally when a Mapper is being used as a nested field.

Raises MapperError

Returns None

Return type None

_get_mapper_type()
Return the specified type for this Mapper. If no __type__ is defined a MapperError is raised

Raises MapperError

Returns The specified __type__ for the mapper.

_get_obj()
Return self.obj or create a new instance of self.__type__

Returns self.obj or new instance of self.__type__

_get_role(*name_or_role*, *deferred_role=None*)
Resolve a string to a role and check it exists, or check a directly passed role is a Role instance and return it.

You may also affect the fields returned from a role at read time using deferred_role. deferred_role is used to provide the intersection between the role specified at name_or_role and the deferred_role.

Usage:

```
class FooMapper(Mapper):
    __type__ = dict
    name = field.String()
    id = field.String()
    secret = field.String()

    __roles__ = {
        'overview': whitelist('id', 'name'),
    }

mapper._get_role('overview', deferred_role=whitelist('id'))
```

Deferred roles can be used for things like allowing end users to provide a list of fields they want back from your API but only if they appear in a role you've specified.

Parameters

- **deferred_role** – provide a role containing fields to dynamically change the permitted fields for the role specified in name_or_role
- **name_or_role** – role name as a string or a Role instance

Raises MapperError

Returns Role instance

Return type Role

_get_fields(*name_or_role*, *deferred_role=None*, *for_marshal=False*)
Returns a list of Field instances providing they are registered in the specified Role.

If the provided name_or_role is not found in the Mappers role list an error will be raised.

Parameters

- **deferred_role** – an instance of role used to dynamically a new role.
- **name_or_role** – the name of a role as a string or a `Role` instance.
- **for_marshal** – Indicate that the mapper is marshaling data.

Raises `MapperError`

Returns list of `Field` instances

Return type list

get_mapper_session(*data*, *output*)

Populate and return a new instance of `MapperSession`

Parameters

- **data** – data being Mapped
- **output** – obj mapper is mapping too

Returns `MapperSession` object

Return type `MapperSession` object

classmethod many(***mapper_params*)

Provide access to a `MapperIterator` to allow multiple items to be mapped by a mapper.

Parameters **mapper_params** – dict of params passed to each new instance of the mapper.

Returns `MapperIterator` object

Return type `MapperIterator`

Usage:

```
>>> mapper = Mapper.many(data=data).marshal()
```

marshal(*role='__default__'*)

Marshal `self.data` into `self.obj` according to the fields defined on this Mapper.

Returns Object of `__type__` populated with data

serialize(*role='__default__'*, *raw=False*, *deferred_role=None*)

Serialize `self.obj` into a dict according to the fields defined on this Mapper.

Parameters

- **role** – specify the role to use when serializing this mapper
- **raw** – instruct the mapper to transform the data before serializing. This option overrides the Mapper.raw setting.

Raises `FieldInvalid` `MapperError`

Returns dict containing serialized object

Return type mixed

Usage::

```
>>> mapper = UserMapper(obj=user)
>>> mapper.serialize(role='public')
```

See also:

`transform_data`

transform_data(*data*)

Transform a flat list of key names into a nested data structure by inflating dunder_score key name into objects.

Parameters **data** – The object or data being transformed

Returns transformed data

Return type dict

Usage:

```
>>> data = ['id', 'name', 'contact_details__phone',
            'contact_details__address__postcode']
>>> mapper.transform_data(data)
{
    'id': x,
    'name': x,
    'contact_details': {
        'phone': x,
        'address': {
            'postcode': x
        }
    }
}
```

validate(*output*)

Mappers may subclass this method to perform top-level validation on multiple related fields, raising *FieldInvalid* or *MappingInvalid* if any problems are found.

Raises FieldInvalid

Raises MappingInvalid

class kim.mapper.**PolymorphicMapper**(*obj=None*, *data=None*, *partial=False*, *raw=False*, *parent=None*)

PolymorphicMappers build on the normal Mapper system to provide functionality for serializing and marshaling collections of different objects with different data structures.

Usage:

```
from kim import Mapper, field

class ActivityMapper(PolymorphicMapper):

    __type__ = Activity

    id = field.String()
    name = field.String()
    object_type = field.String(choices=['event', 'task'])
    created_at = field.DateTime(read_only=True)

    __mapper_args__ = {
        'polymorphic_on': object_type,
    }


class TaskMapper(ActivityMapper):
```

```
        __type__ = Task

        status = field.String(read_only=True)
        is_complete = field.Boolean()

        __mapper_args__ = {
            'polymorphic_name': 'task'
        }


    class EventMapper(ActivityMapper):

        __type__ = Event

        location = field.String(read_only=True)

        __mapper_args__ = {
            'polymorphic_name': 'event'
        }
```

Initialise a Mapper with the object and/or the data to be serialzed/marshaled. Mappers must be instantiated once per object/data. At least one of obj or data must be passed.

Parameters

- **obj** – the object to be serialized, or updated by marshaling
- **data** – input data to be used for marshaling
- **raw** – the mapper will instruct fields to populate themselves using __dunder__ field names where required.
- **partial** – allow pipelines to pull data from an existing source or fall back to standard checks.
- **parent** – The parent of this Mapper. Set internally when a Mapper is being used as a nested field.

Raises `MapperError`

Returns None

Return type None

get_mapper_session(*data*, *output*)

Populate and return a new instance of `MapperSession`

Parameters

- **data** – data being Mapped
- **output** – obj mapper is mapping too

Returns `MapperSession` object

Return type `MapperSession` object

classmethod get_polymorphic_identity(*key*)

Return the polymorphic mapper stored at `key`.

Parameters **key** – The name of a polymoprhic indentity

Raises *kim.exception.MapperError*

Return type *kim.mapper.Mapper*

Returns the Mapper stored against key

classmethod get_polymorphic_key(*obj=None*, *data=None*)
Return the value from obj when serializing or from data when marshaling for the polymorphic_on key.

Parameters

- **data** – datum being marshaled by the Mapper
- **obj** – obj being serialized by the Mapper

Returns the polymorphic type name

Return type str

Raises *kim.exception.FieldInvalid* *kim.exception.MappingInvalid*

classmethod is_polymorphic_base()
Return a boolean indicating if this cls is the base type in the class hierarchy

Returns True if the class is the base type, otherwise False

Return type boolean

many(***mapper_params*)
Provide access to a `MapperIterator` to allow multiple items to be mapped by a mapper.

Parameters **mapper_params** – dict of params passed to each new instance of the mapper.

Returns `MapperIterator` object

Return type `MapperIterator`

Usage:

```
>>> mapper = Mapper.many(data=data).marshal()
```

marshal(*role='__default__'*)
Marshal `self.data` into `self.obj` according to the fields defined on this Mapper.

Returns Object of `__type__` populated with data

serialize(*role='__default__'*, *raw=False*, *deferred_role=None*)
Serialize `self.obj` into a dict according to the fields defined on this Mapper.

Parameters

- **role** – specify the role to use when serializing this mapper
- **raw** – instruct the mapper to transform the data before serializing. This option overrides the Mapper.raw setting.

Raises `FieldInvalid` `MapperError`

Returns dict containing serialized object

Return type mixed

Usage::

```
>>> mapper = UserMapper(obj=user)
>>> mapper.serialize(role='public')
```

See also:

`transform_data`

transform_data(*data*)

Transform a flat list of key names into a nested data structure by inflating dunder_score key name into objects.

Parameters **data** – The object or data being transformed

Returns transformed data

Return type dict

Usage:

```
>>> data = ['id', 'name', 'contact_details__phone',
            'contact_details__address__postcode']
>>> mapper.transform_data(data)
{
    'id': x,
    'name': x,
    'contact_details': {
        'phone': x,
        'address': {
            'postcode': x
        }
    }
}
```

validate(*output*)

Mappers may subclass this method to perform top-level validation on multiple related fields, raising *FieldInvalid* or *MappingInvalid* if any problems are found.

Raises FieldInvalid

Raises MappingInvalid

class `kim.mapper.`**MapperIterator**(*mapper*, ***mapper_params*)

Provides a symmetric interface for Mapping many objects in one batch.

A simple example would be seriaizing a list of User objects from a database query or other source.

Usage:

```
from kim import Mapper, field

class UserMapper(Mapper):
    __type__ = User

    id = field.Integer(read_only=True)
    name = field.String(required=True)
    company = field.Nested('myapp.mappers.CompanyMapper')

objs = User.query.all()
results = UserMapper.many().serialize(objs)
```

Constructs a new instance of a MapperIterator.

Parameters

- **mapper** – a *Mapper* to map each item too.
- **mapper_params** – a dict of kwargs passed to each mapper

get_mapper(*data=None*, *obj=None*)
Return a new instance of the provided mapper.

Parameters

- **data** – provide the new mapper with data when marshaling
- **obj** – provide the new mapper with data when serializing

Return type *Mapper*

Returns a new *Mapper*

marshal(*data*, *role='__default__'*)
Marshals each item in `data` creating a new mapper each time.

Parameters

- **objs** – iterable of objects to marshal
- **role** – name of a role to use when marshaling

Returns list of marshaled objects

serialize(*objs*, *role='__default__'*, *deferred_role=None*)
Serializes each item in `objs` creating a new mapper each time.

Parameters

- **objs** – iterable of objects to serialize
- **role** – name of a role to use when serializing

Returns list of serialized objects

class kim.mapper.**MapperSession**(*mapper*, *data*, *output*, *partial=None*)
Object that represents the state of a `Mapper` during the execution of marshaling and serialization `Pipeline`.

Instantiate a new instance of `MapperSession`

Parameters

- **mapper** – `Mapper` instance.
- **data** – The data marshaled by the `Mapper`
- **output** – The object the `Mapper` is outputting to.

Returns None

Return type None

See also:

get_mapper_session method `get_mapper_session`

Fields

class kim.field.**Field**(**args*, ***field_opts*)
Field, as it's name suggests, represents a single key or 'field' inside of your mappings. Much like columns in a database or a csv, they provide a way to represent different data types when pushing data into and out of your Mappers.

A core concept of Kims architecture is that of Pipelines. Every Field makes use of both an Input and Output pipeline which affords users a great level of flexibility when it comes to handling data.

Kim provides a collection of default Field implementations, for more complex cases extending Field to create new field types couldn't be easier.

Usage:

```
from kim import Mapper
from kim import field

class UserMapper(Mapper):

    id = field.Integer(required=True, read_only=True)
    name = field.String(required=True)
```

Constructs a new instance of Field. Each Field accepts a set of kwargs that will be passed directly to the fields defined `FieldOpts`.

Parameters

- **args** – list of arguments passed to the field
- **kwargs** – keyword arguments typically passed to the FieldOpts class attached to this Field.

Raises `FieldOptsError`

Returns None

See also:

`FieldOpts`

opts_class = <class 'kim.field.FieldOpts'>
The `FieldOpts` field config class to use for the Field.

marshal_pipeline = <class 'kim.pipelines.marshaling.MarshalPipeline'>
The Fields marshaling pipeline

serialize_pipeline = <class 'kim.pipelines.serialization.SerializePipeline'>
The Fields serialization pipeline

get_error(*error_type*)
Return the error message for `error_type` from the error messages defined on the fields opts class.

Parameters **error_type** – the key of the error found in self.error_msgs

Returns Error message

Return type string

invalid(*error_type*)
Raise an Exception using the provided error_type for the error message. This method is typically used by pipes to allow `Field` to control how its errors are handled.

Usage:

```
@pipe()
def validate_name(session):
    if session.data and session.data != 'Mike Waites':
        raise session.fied.invalid('not_mike')
```

Parameters **error_type** – The key of the error being raised.

Raises `FieldInvalid`

See also:

`FieldOpts` for an explanation on defining error messags

marshal (*mapper_session*, ***opts*)
Run the marshal `Pipeline` for this field for the given `data` and update the output for this field inside of the mapper_session.

Parameters **mapper_session** – The Mappers marshaling session this field is being run inside of.

Opts kwargs passed to the marshal pipelines run method.

Returns None

See also:

kim.mapper.Mapper.marshal

marshal_pipeline
The Fields marshaling pipeline

alias of `MarshalPipeline`

name
Proxy access to the `FieldOpts` defined for this field.

Return type str

Returns The value of get_name from FieldOpts

Raises `FieldError`

See also:

kim.field.FieldOpts.get_name

opts_class
The `FieldOpts` field config class to use for the Field.

alias of `FieldOpts`

serialize (*mapper_session*, ***opts*)
Run the serialize `Pipeline` for this field for the given *data* and update *output* in for this field inside of the mapper_session.

Parameters **mapper_session** – The Mappers marshaling session this field is being run inside of.

Opts kwargs passed to the marshal pipelines run method.

Returns None

See also:

kim.mapper.Mapper.serialize

serialize_pipeline
The Fields serialization pipeline

alias of `SerializePipeline`

class `kim.field.`**FieldOpts** (***opts*)
FieldOpts are used to provide configuration options to *Field*. They are designed to allow users to easily provide custom configuration options to *Field* classes.

Custom *FieldOpts* classes are set on *Field* using the `opts_class` property.

```
class MyFieldOpts(FieldOpts):

    def __init__(self, **opts):

        self.some_property = opts.get('some_property', None)
        super(MyFieldOpts, self).__init__(**opts)
```

See also:

Field

Construct a new instance of `FieldOpts` and set config options

Parameters

- **name** – Specify the name of the field for data output
- **required** – This field must be present when marshaling
- **attribute_name** – Specify internal name for this field, set on mapper.fields dict
- **source** – Specify the name of the attribute on the object to use when getting/setting data. May be `__self__` to use entire mapper object as data
- **default** – Specify a default value for this field to apply when serializing or marshaling
- **allow_none** – This option only takes affect if required=False. If allow_none=False and required=False, then Kim will accept either the field being missing completely from the data, or the field being passed with a non-None value. That is, either `{}` or `{'field': 'value'}` but never `{'field': None}`. Default True.
- **read_only** – Specify if this field should be ignored when marshaling
- **error_msgs** – A dict of error_type: error messages.
- **null_default** – Specify the default type to return when a field is null IE None or {} or ''
- **choices** – Specify a list of valid values
- **extra_serialize_pipes** – dict of lists containing extra Pipe functions to be run at the end of each stage when serializing. eg `{'output': [my_pipe, my_other_pipe]}`
- **extra_marshal_pipes** – dict of lists containing extra Pipe functions to be run at the end of each stage when marshaling. eg `{'validate': [my_pipe, my_other_pipe]}`

Raises *FieldOptsError*

Returns None

get_name()

Return the name property set by `set_name`

Return type str

Returns the name of the field to be used in input/output

set_name(*name=None*, *attribute_name=None*, *source=None*)

Programmatically set the name properties for a field.

Names cascade from each other unless they are explicitly overridden.

Example 1: class MyMapper(Mapper):

foo = field.String()

attribute_name = foo name = foo source = foo

Example 2: class MyMapper(Mapper):

foo = field.String(name='bar', source='baz')

attribute_name = foo name = bar source = baz

Parameters

- **name** – value of name property
- **attribute_name** – value of attribute_name property
- **source** – value of source property

Returns None

validate()

Allow users to perform checks for required config options. Concrete classes should raise *FieldError* when invalid configuration is encountered.

A slightly contrived example is requiring all fields to be `read_only=True`

Usage:

```
from kim.field import FieldOpts

class MyOpts(FieldOpts):

    def validate(self):

        if self.read_only is True:
            raise FieldOptsError('Field cannot be read only')
```

Raises *.FieldOptsError*

Returns None

class kim.field.**String**(**args*, ***field_opts*)

`String` represents a value that must be valid when passed to str()

Usage:

```
from kim import Mapper
from kim import field

class UserMapper(Mapper):
    __type__ = User

    name = field.String(required=True)
```

Constructs a new instance of Field. Each Field accepts a set of kwargs that will be passed directly to the fields defined `FieldOpts`.

Parameters

- **args** – list of arguments passed to the field
- **kwargs** – keyword arguments typically passed to the FieldOpts class attached to this Field.

Raises `FieldOptsError`

Returns None

See also:

`FieldOpts`

marshal_pipeline
alias of `StringMarshalPipeline`

opts_class
alias of `StringFieldOpts`

serialize_pipeline
alias of `StringSerializePipeline`

class `kim.field.`**Integer**(**args*, ***field_opts*)
`Integer` represents a value that must be valid when passed to int()

Usage:

```
from kim import Mapper
from kim import field

class UserMapper(Mapper):
    __type__ = User

    id = field.Integer(required=True, min=1, max=10)
```

Constructs a new instance of Field. Each Field accepts a set of kwargs that will be passed directly to the fields defined `FieldOpts`.

Parameters

- **args** – list of arguments passed to the field
- **kwargs** – keyword arguments typically passed to the FieldOpts class attached to this Field.

Raises `FieldOptsError`

Returns None

See also:

`FieldOpts`

marshal_pipeline
alias of `IntegerMarshalPipeline`

opts_class
alias of `IntegerFieldOpts`

serialize_pipeline
alias of `IntegerSerializePipeline`

class `kim.field.`**IntegerFieldOpts**(***kwargs*)
Custom FieldOpts class that provides additional config options for `Integer`.

Construct a new instance of `IntegerFieldOpts` and set config options

Parameters

- **max** – Specify the maximum permitted value
- **min** – Specify the minimum permitted value

Raises `FieldOptsError`

Returns None

class kim.field.**Decimal**(**args*, ***field_opts*)

Decimal represents a value that must be valid when passed to decimal.Decimal()

Usage:

```
from kim import Mapper
from kim import field

class UserMapper(Mapper):
    __type__ = User

    score = field.Decimal(precision=4, min=0, max=1.5)
```

Constructs a new instance of Field. Each Field accepts a set of kwargs that will be passed directly to the fields defined FieldOpts.

Parameters

- **args** – list of arguments passed to the field
- **kwargs** – keyword arguments typically passed to the FieldOpts class attached to this Field.

Raises FieldOptsError

Returns None

See also:

FieldOpts

marshal_pipeline

alias of DecimalMarshalPipeline

opts_class

alias of FloatFieldOpts

serialize_pipeline

alias of DecimalSerializePipeline

class kim.field.**Boolean**(**args*, ***field_opts*)

Boolean represents a value that must be valid boolean type.

Usage:

```
from kim import Mapper
from kim import field

class UserMapper(Mapper):
    __type__ = User

    active = field.Boolean(
        required=True,
        true_boolean_values=[True, 'true', 1],
        false_boolean_values=[False, 'false', 0])
```

Constructs a new instance of Field. Each Field accepts a set of kwargs that will be passed directly to the fields defined FieldOpts.

Parameters

- **args** – list of arguments passed to the field
- **kwargs** – keyword arguments typically passed to the FieldOpts class attached to this Field.

Raises FieldOptsError

Returns None

See also:

FieldOpts

marshal_pipeline
alias of BooleanMarshalPipeline

opts_class
alias of BooleanFieldOpts

serialize_pipeline
alias of BooleanSerializePipeline

class kim.field.**BooleanFieldOpts**(***kwargs*)
Custom FieldOpts class that provides additional config options for Boolean.

Construct a new instance of BooleanFieldOpts and set config options

Parameters

- **true_boolean_values** – Specify an array of values that will validate as being 'true' when the field is marshaled.
- **false_boolean_values** – Specify an array of values that will validate as being 'false' when the field is marshaled.

Raises FieldOptsError

Returns None

class kim.field.**Nested**(**args*, ***kwargs*)
Nested represents an object that is represented by another mapper.

Usage:

```
from kim import Mapper
from kim import field

class PostMapper(Mapper):
    __type__ = User

    id = field.String()
    name= field.String()
    content = field.String()
    user = field.Nested(
        'UserMapper',
        role='public',
        getter=user_getter,
        allow_upadtes=False,
        allow_partial_updates=False,
        allow_updates_in_place=False,
        allow_create=False,
        required=True)
```

See also:

NestedFieldOpts

get_mapper(*as_class=False*, ***mapper_params*)
Retrieve the specified mapper from the Mapper registry.

Parameters

- **as_class** – Return the Mapper class object without calling the constructor. This is typically used when nested is mapping many objects.
- **mapper_params** – A dict of kwarg's to pass to the specified mappers constructor

Return type Mapper

Returns a new instance of the specified mapper

marshal_pipeline
alias of NestedMarshalPipeline

opts_class
alias of NestedFieldOpts

serialize_pipeline
alias of NestedSerializePipeline

class kim.field.**NestedFieldOpts**(*mapper_or_mapper_name*, ***kwargs*)
Custom FieldOpts class that provides additional config options for Nested.

Construct a new instance of NestedFieldOpts

Parameters

- **mapper_or_mapper_name** – a required instance of a Mapper or a valid mapper name
- **role** – specify the name of a role to use on the Nested mapper
- **collection_class** – provide a custom type to be used when mapping many nested objects
- **getter** – provide a function taking a pipeline session which returns the object to be set on this field, or None if it can't find one. This is useful where your API accepts simply *{'id': 2}* but you want a full object to be set
- **allow_updates** – Allow existing objects returned by the getter function to be updated.
- **allow_updates_in_place** – Whereas allow_updates requires the getter to return an existing object which it will then update, allow_updates_in_place will make updates to any existing object it finds at the specified key.
- **allow_create** – If the getter returns None, allow the Nested field to create a new instance.
- **allow_partial_updates** – Allow existing object to be updated using a subset of the fields defined on the Nested field.

class kim.field.**Collection**(**args*, ***field_opts*)
Collection represents collection of other field types, typically stored in a list.

Usage:

```
from kim import Mapper
from kim import field

class UserMapper(Mapper):
    __type__ = User

    id = field.String()
    friends = field.Collection(field.Nested('UserMapper', required=True))
    user_ids = field.Collection(field.String())
```

See also:

`CollectionFieldOpts`

Constructs a new instance of Field. Each Field accepts a set of kwargs that will be passed directly to the fields defined `FieldOpts`.

Parameters

- **args** – list of arguments passed to the field
- **kwargs** – keyword arguments typically passed to the FieldOpts class attached to this Field.

Raises `FieldOptsError`

Returns None

See also:

`FieldOpts`

marshal_pipeline
alias of `CollectionMarshalPipeline`

opts_class
alias of `CollectionFieldOpts`

serialize_pipeline
alias of `CollectionSerializePipeline`

class `kim.field.`**CollectionFieldOpts**(*field*, ***kwargs*)
Custom FieldOpts class that provides additional config options for `Collection`.

Construct a new instance of *CollectionFieldOpts*

Parameters

- **field** – Specify the field type mpapped inside of this collection. This may be any `Field` type.
- **unique_on** – Specify a key that is used to check the collection for duplicates.

get_name()
Proxy access to the `FieldOpts` defined for this collections field.

Return type str

Returns The value of get_name from the collections Field.

set_name(**args*, ***kwargs*)
proxy access to the `FieldOpts` defined for this collections field.

Returns None

validate()
Exra validation for Collection Field.

Raises FieldOptsError

class `kim.field.`**Static**(**args*, ***field_opts*)
`Static` represents a field that outputs a constant value.

This field is implicitly read_only and therefore is typically only used during serialization flows.

Usage:

```
from kim import Mapper
from kim import field

class UserMapper(Mapper):
    __type__ = User

    id = field.String()
    object_type = field.Static(value='user')
```

Constructs a new instance of Field. Each Field accepts a set of kwargs that will be passed directly to the fields defined `FieldOpts`.

Parameters

- **`args`** – list of arguments passed to the field
- **`kwargs`** – keyword arguments typically passed to the FieldOpts class attached to this Field.

Raises `FieldOptsError`

Returns None

See also:

`FieldOpts`

`opts_class`
alias of `StaticFieldOpts`

`serialize_pipeline`
alias of `StaticSerializePipeline`

class `kim.field.`**`StaticFieldOpts`**(*value*, ***kwargs*)
Custom FieldOpts class that provides additional config options for `Static`.

Construct a new instance of `StaticFieldOpts`

Parameters **`value`** – specify the static value to return when this field is serialized.

class `kim.field.`**`DateTime`**(**args*, ***field_opts*)
`DateTime` represents an iso8601 encoded date time

```
from kim import Mapper
from kim import field

class UserMapper(Mapper):
    __type__ = User

    created_at = field.DateTime(required=True)
```

Constructs a new instance of Field. Each Field accepts a set of kwargs that will be passed directly to the fields defined `FieldOpts`.

Parameters

- **`args`** – list of arguments passed to the field
- **`kwargs`** – keyword arguments typically passed to the FieldOpts class attached to this Field.

Raises `FieldOptsError`

Returns None

See also:

FieldOpts

marshal_pipeline
alias of DateTimeMarshalPipeline

opts_class
alias of DateTimeFieldOpts

serialize_pipeline
alias of DateTimeSerializePipeline

class kim.field.**Date**(**args*, ***field_opts*)
Date represents a date object

```
from kim import Mapper
from kim import field

class UserMapper(Mapper):
    __type__ = User

    signup_date = field.Date(required=True)
```

Constructs a new instance of Field. Each Field accepts a set of kwargs that will be passed directly to the fields defined FieldOpts.

Parameters

- **args** – list of arguments passed to the field
- **kwargs** – keyword arguments typically passed to the FieldOpts class attached to this Field.

Raises FieldOptsError

Returns None

See also:

FieldOpts

marshal_pipeline
alias of DateMarshalPipeline

opts_class
alias of DateFieldOpts

Roles

class kim.role.**Role**(**args*, ***kwargs*)
Roles are a fundamental feature of Kim. It's very common to need to provide a different view of your data or to only require a selection of fields when marshaling data. Roles in Kim allow users to shape their data at runtime in a simple yet flexible manner.

Roles are added to your *Mapper* declarations using the __roles__ attribute.

Usage:

```
from kim import Mapper, whitelist, field

class UserMapper(Mapper):
    __type__ = User

    id = field.Integer(read_only=True)
```

```
        name = field.String(required=True)
        company = field.Nested('myapp.mappers.CompanyMapper')

        __roles__ = {
            'id_only': whitelist('id')
        }
```

initialise a new `Role`.

Parameters **whitelist** – pass a boolean indicating whether this role is a whitelist

__contains__(*field_name*)

overloaded membership test that inverts the check depending on wether the role is a whitelist or blacklist.

If the role is defined as whitelist=True the normal membership test is applied ie:

```
        >>> 'name' in whitelist('name')
        True
```

For blacklist the test is flipped as we are aiming to ensure the field name is not present in the role:

```
        >>> 'other_name' in blacklist('name')
        True
        >>> 'name' in blacklist('name')
        False
```

Parameters **field_name** – name of a field to test for membership

Return type boolean

Returns boolean indicating wether field_name is found in the role

__or__(*other*)

Override handling of producing the union of two Roles to provide native support for merging whitelist and blacklist roles correctly.

This overloading allows users to produce the union of two roles that may, on one side, want to allow fields and on the other exclude them.

Usage:

```
        >>> from kim.role import whitelist, blacklist
        >>> my_role = whitelist('foo', 'bar') | blacklist('foo', 'baz')
        >>> my_role
        Role('bar')
```

Parameters **other** – another instance of *kim.role.Role*

Raises *kim.exception.RoleError*

Return type *kim.role.Role*

Returns a new *kim.role.Role* containng the set of field names

fields

return an iterable containing all the field names defined in this role.

Return type list

Returns iterable of field names

class kim.role.**whitelist**(**args*, ***kwargs*)

Whitelists are roles that define a list of fields that are permitted for inclusion when marhsaling or serializing. For example, a whitelist role called `id_only` that contains the field name `id` instructs kim that whenever the `id_only` role is used **only** the `id` field should be considered in the input/output data.

Usage:

```
from kim import whitelist

id_only_role = whitelist('id')

class IdMixin(object):

    id = fields.Integer(read_only=True)

    __roles__ = {
        'id_only': id_only
    }
```

class kim.role.**blacklist**(**args*, ***kwargs*)

Blacklists are role that act in the opposite manner to whitelists. They define a list of fields that should not be used when marshaling and serializing data. A blacklist role named `id_less` that contained the field name `id` would instruct kim that every field defined on the mapper should be considered except `id`.

Usage:

```
from kim import whitelist

class UserMapper(Mapper):

    id_less_role = blacklist('id')

    __roles__ = {
        'id_less': blacklist('id')
    }
```

Pipelines

kim.pipelines.base.**pipe**(***pipe_kwargs*)

Pipe decorator is provided as a convenience to avoid duplicating logic like not running pipes when session.data is null.

Parameters **run_if_none** – Specify wether the pipe function should be called if session.data is None.

Usage:

```
from kim.pipelines.base import pipe

@pipe(run_if_none=True)
def my_pipe(session):

    do_stuff(session)
```

class kim.pipelines.base.**Pipeline**

Pipelines provide a simple, extensible way of processing data for a *kim.field.Field*. Each pipeline provides 4 input groups, `input_pipes`, `validation_pipes`, `process_pipes` and `output_pipes`. Each containing *pipe* functions that are called in order passing data from one pipe to another.

Kim pipes are similar to unix pipes, where each pipe in the chain has a single role in handling data before passing it on to the next pipe in the chain.

Pipelines are typically ignorant to whether they are marhsaling data or serializing data, they simply take data in one end, this may be a deserialized dict of JSON or an object that's been populated from the database, and produce an output at the other.

Usage:

```
from kim.pipelines.base import Pipeline

class StringIntPipeline(Pipeline):

    input_pipes = [get_data_from_json]
    validation_pipes = [is_numeric_string]
    process_pipes [cast_to_int]
    output_pipes = [update_output]
```

class kim.pipelines.marshaling.**MarshalPipeline**

See also:

kim.pipelines.base.read_only *kim.pipelines.base.get_data_from_name* *kim.pipelines.base.update_output_to_source*

class kim.pipelines.serialization.**SerializePipeline**

See also:

kim.pipelines.base.get_data_from_name *kim.pipelines.base.update_output_to_name*

class kim.pipelines.string.**StringMarshalPipeline**

See also:

kim.pipelines.base.is_valid_choice *kim.pipelines.string.is_valid_string* *kim.pipelines.marshaling.MarshalPipeline*

class kim.pipelines.string.**StringSerializePipeline**

See also:

kim.pipelines.serialization.SerializePipeline

class kim.pipelines.numeric.**IntegerMarshalPipeline**

See also:

kim.pipelines.numeric.is_valid_integer *kim.pipelines.base.is_valid_choice* *kim.pipelines.numeric.bounds_check* *kim.pipelines.marshaling.MarshalPipeline*

class kim.pipelines.numeric.**IntegerSerializePipeline**

See also:

kim.pipelines.serialization.SerializePipeline

class kim.pipelines.boolean.**BooleanMarshalPipeline**

See also:

kim.pipelines.base.is_valid_choice *kim.pipelines.boolean.coerce_to_boolean* *kim.pipelines.marshaling.MarshalPipeline*

class kim.pipelines.numeric.**DecimalMarshalPipeline**

See also:

kim.pipelines.numeric.is_valid_decimal *kim.pipelines.numeric.coerce_to_decimal* *kim.pipelines.marshaling.MarshalPipeline*

class kim.pipelines.numeric.**DecimalSerializePipeline**

See also:

kim.pipelines.numeric.coerce_to_decimal *kim.pipelines.numeric.to_string* *kim.pipelines.serialization.SerializePipeline*

class kim.pipelines.boolean.**BooleanSerializePipeline**

See also:

kim.pipelines.serialization.SerializePipeline

class kim.pipelines.nested.**NestedMarshalPipeline**

See also:

kim.pipelines.nested.marshal_nested *kim.pipelines.marshaling.MarshalPipeline*

class kim.pipelines.nested.**NestedSerializePipeline**

See also:

kim.pipelines.nested.serialize_nested *kim.pipelines.serialization.SerializePipeline*

class kim.pipelines.collection.**CollectionMarshalPipeline**

See also:

kim.pipelines.collection.check_duplicates kim.pipelines.collection.marshal_collection *kim.pipelines.marshaling.MarshalPipeline*

class kim.pipelines.collection.**CollectionSerializePipeline**

See also:

kim.pipelines.collection.serialize_collection *kim.pipelines.serialization.SerializePipe*

class kim.pipelines.datetime.**DateTimeMarshalPipeline**

See also:

kim.pipelines.datetime.is_valid_datetime *kim.pipelines.base.is_valid_choice* *kim.pipelines.marshaling.MarshalPipeline*

class kim.pipelines.datetime.**DateTimeSerializePipeline**

See also:

kim.pipelines.datetime.format_datetime *kim.pipelines.marshaling.MarshalPipeline*

class kim.pipelines.datetime.**DateMarshalPipeline**

See also:

kim.pipelines.datetime.cast_to_date *kim.pipelines.marshaling.MarshalPipeline*

class kim.pipelines.datetime.**DateSerializePipeline**

See also:

kim.pipelines.serialization.SerializePipeline

class kim.pipelines.static.**StaticSerializePipeline**

See also:

kim.pipelines.static.get_static_value *kim.pipelines.serialization.SerializePipeline*

Pipes

Base

kim.pipelines.base.**get_data_from_name**(*session*, **args*, ***kwargs*)
Extracts a specific key from data using `field.name`. This pipe is typically used as the entry point to a chain of input pipes.

Parameters **session** – Kim pipeline session instance

Return type mixed

Returns the key found in data using field.name

kim.pipelines.base.**get_data_from_source**(*session*, **args*, ***kwargs*)
Extracts a specific key from data using `field.source`. This pipe is typically used as the entry point to a chain of output pipes.

Parameters **session** – Kim pipeline session instance

Return type mixed

Returns the key found in data using field.source

kim.pipelines.base.**get_field_if_required**(*session*, **args*, ***kwargs*)

kim.pipelines.base.**read_only**(*session*, **args*, ***kwargs*)
End processing of a pipeline if a Field is marked as read_only.

Parameters **session** – Kim pipeline session instance

Raises StopPipelineExecution

kim.pipelines.base.**is_valid_choice**(*session*, **args*, ***kwargs*)
End processing of a pipeline if a Field is marked as read_only.

Parameters **session** – Kim pipeline session instance

Raises StopPipelineExecution

kim.pipelines.base.**update_output_to_name**(*session*, **args*, ***kwargs*)
Store `data` at `field[name]` for a `field` inside of `output`

Parameters **session** – Kim pipeline session instance

Returns None

kim.pipelines.base.**update_output_to_source**(*session*, **args*, ***kwargs*)
Store `data` at field.opts.source for a `field` inside of `output`

Parameters **session** – Kim pipeline session instance

Raises FieldError

Returns None

String

kim.pipelines.string.**is_valid_string**(*session*, **args*, ***kwargs*)
Pipe used to determine if a value can be coerced to a string

Parameters **session** – Kim pipeline session instance

kim.pipelines.string.**to_unicode**(*session*, **args*, ***kwargs*)
Convert incoming value to unicode string

Parameters **session** – Kim pipeline session instance

kim.pipelines.string.**bounds_check**(*session*, **args*, ***kwargs*)
Pipe used to determine if a value is within the min and max bounds on the field

Parameters **session** – Kim pipeline session instance

Integer

kim.pipelines.numeric.**is_valid_integer**(*session*, **args*, ***kwargs*)
Pipe used to determine if a value can be coerced to an int

Parameters **session** – Kim pipeline session instance

kim.pipelines.numeric.**bounds_check**(*session*, **args*, ***kwargs*)
Pipe used to determine if a value is within the min and max bounds on the field

Parameters **session** – Kim pipeline session instance

String

kim.pipelines.numeric.**is_valid_decimal**(*session*, **args*, ***kwargs*)
Pipe used to determine if a value can be coerced to a Decimal

Parameters **session** – Kim pipeline session instance

kim.pipelines.numeric.**coerce_to_decimal**(*session*, **args*, ***kwargs*)
Coerce str representation of a decimal into a valid Decimal object.

kim.pipelines.numeric.**to_string**(*session*, **args*, ***kwargs*)
coerce decimal value into str so it's valid for json

Boolean

kim.pipelines.boolean.**coerce_to_boolean**(*session*, **args*, ***kwargs*)
Given a valid boolean value, ie True, 'true', 'false', False, 0, 1 set the data to the python boolean type True or False

Parameters **session** – Kim pipeline session instance

Nested

kim.pipelines.nested.**marshal_nested**(*session*, **args*, ***kwargs*)
Marshal data using the nested mapper defined on this field.

There are 6 possible scenarios, depending on the security setters and presence of a getter function

- Getter function returns an object and no updates are allowed - Return the object immediately
- Getter function returns an object and updates are allowed - Call the nested mapper with the object to update it
- Object already exists, getter function returns None/does not exist and in place updates are allowed - Call the nested mapper with the existing object to update it
- Getter function returns None/does not exist and creation of new objects is allowed - Call the nested mapper to create a new object
- Getter function returns None/does not exist and creation of new objects is not allowed, nor are in place updates - Raise an exception.
- Object already exists, getter function returns None/does not exist and partial updates are allowed - Call the nested mapper with the existing object to update it

Parameters **session** – Kim pipeline session instance

kim.pipelines.nested.**serialize_nested**(*session*, **args*, ***kwargs*)
Serialize data using the nested mapper defined on this field.

Parameters **session** – Kim pipeline session instance

Collection

kim.pipelines.collection.**marshall_collection**(*session*, **args*, ***kwargs*)
iterate over each item in `data` and marshal the item through the wrapped field defined for this collection

Parameters **session** – Kim pipeline session instance

TODO(mike) this should be called marshal_collection

kim.pipelines.collection.**serialize_collection**(*session*, **args*, ***kwargs*)
iterate over each item in `data` and serialize the item through the wrapped field defined for this collection

Parameters **session** – Kim pipeline session instance

kim.pipelines.collection.**check_duplicates**(*session*, **args*, ***kwargs*)
iterate over collection and check for duplicates if th unique_on FieldOpt has been set of this Collection field

TODO(mike) This should only run if the wrapped field is a nested collection

Datetime

kim.pipelines.datetime.**is_valid_datetime**(*session*, **args*, ***kwargs*)
Pipe used to determine if a value can be coerced to a datetime

Parameters **session** – Kim pipeline session instance

kim.pipelines.datetime.**format_datetime**(*session*, **args*, ***kwargs*)
Convert date or datetime object into formatted string representation.

Date

kim.pipelines.datetime.**cast_to_date**(*session*, **args*, ***kwargs*)
cast session.data datetime object to a date() instance

Static

kim.pipelines.static.**get_static_value**(*session*, **args*, ***kwargs*)
return the static value specified in FieldOpts

Exceptions

exception kim.exception.**KimException**(*message*, **args*, ***kwargs*)
Base Exception for all Kim exception types.

exception kim.exception.**MapperError**(*message*, **args*, ***kwargs*)
MapperError is raised from a mapper that was unable to instantiate correctly.

exception kim.exception.**MappingInvalid**(*errors*, **args*, ***kwargs*)

exception kim.exception.**RoleError**(*message*, **args*, ***kwargs*)

exception kim.exception.**FieldOptsError**(*message*, **args*, ***kwargs*)

exception kim.exception.**FieldError**(*message*, **args*, ***kwargs*)

exception kim.exception.**FieldInvalid**(**args*, ***kwargs*)

exception kim.exception.**StopPipelineExecution**(*message*, **args*, ***kwargs*)

CHAPTER 4

About Kim

Benchmarks

Below is the output of a benchmark written and maintained by @voidfiles. You can find the results here https://voidfiles.github.io/python-serialization-benchmark/

k

Symbols

B

C

D

F

G

I

K

M

N

O

P

R

S

T

U

V

W