django-nap Documentation

Release 0.40.0

Curtis Maloney

Nov 18, 2019

Contents

Web APIs you can do in your sleep...

In the spirit of the Unix philosophy, Nap provides a few tools which each do one thing, and do it well. They are:

1. Data Mapper

   Wrapper classes for providing JSON-friendly interfaces to your objects.

2. RESTful Class-Based Views

   A collection of mixins and views for building class-based API views.

3. RPC View

   A mixin for Django's class-based views which allows a single url to provide multiple RPC methods.

Nap does not provide the wide range of features you see in tools like Django REST Framework, such as rate limiting, token authentication, automatic UI, etc. Instead, it provides a flexible framework that makes it easy to combine with other specialised apps.

Contents:

CHAPTER 1

Quick Start

Nap REST views work by combining Mappers with composible Class-Based Views.

Let's see how you might got about providng an API for the Poll example from the Django tutorial.

1.1 Mapper/Views Quick Start

1. Create a Mapper for your Model in mappers.py

This is very much like defining a ModelForm.

```
from nap import mapper

from . import models


class QuestionMapper(mapper.ModelMapper):
    class Meta:
        model = models.Question
        fields = '__all__'
```

2. Create some views in rest_views.py

```
from nap.rest import views

from . import mappers, models

class QuestionMixin:
    model = models.Question
    mapper_class = mappers.QuestionMapper


class QuestionListView(QuestionMixin,
                       views.ListGetMixin,
```

(continues on next page)

(continued from previous page)

```
                        views.ListPostMixin,
                        views.ListBaseView):
    pass

class QuestionObjectView(QuestionMixin,
                         views.ObjectGetMixin,
                         views.ObjectPutMixin,
                         views.ObjectBaseView):
    pass
```

The *ListBaseView* provides the core of any object list view, deriving from Django's *MultipleObjectMixin*. Then we mix in the default handlers for `GET` and `POST` actions.

Similarly, the *ObjectBaseView* supports single object access, deriving from Django's *SingleObjectMixin*.

Where the list view has `POST` to create a new record, the object view has `PUT` to update an existing record.

3. Add your APIs to your URLs:

```
urlpatterns = [
    url(r'^question/$',
        QuestionListView.as_view(),
        name='question-list'),

    url(r'^question/(?P<pk>\d+)/$',
        QuestionObjectView.as_view(),
        name='question-detail'),
]
```

And we're done. You can now access your Question model!

CHAPTER 2

Tutorials

The next pages will guide you through some stages of adding an API to an app.

We'll start assuming you have the *polls* app from the Django Tutorial, and we'll add a JSON API to it.

Be sure, however, that you used Python 3 when creating your project, as *django-nap* no longer supports Python 2.

2.1 Installation

Installing *django-nap* is as simple as using *pip*:

```
pip install django-nap
```

It does not require being added to *settings.INSTALLED_APPS*, not does it require any settings to be added.

2.2 Mappers

Mappers help us to convert our Python objects, like model instances, into simpler type that are supported by JSON, and back again.

They help us map between how we want our data viewed in the API, and how it's viewed by the rest of the system.

Mappers use a declarative style, just like Django's Models and Forms.

Also, just like Django Forms, there are ModelMappers to simplify building Mappers for models.

2.2.1 Question Mapper

So let's start with the *QuestionMapper*. Create a new file in your poll/ app, and call it "mappers.py"

Listing 1: polls/mappers.py

```
from nap import mapper

from . import models


class QuestionMapper(mapper.ModelMapper):
    class Meta:
        model = models.Question
        fields = '__all__'
```

For anyone familiar with ModelForms, this should look very familiar.

So what does this get us? Well, let's drop into a shell and try it out.

```
>>> from polls.mappers import QuestionMapper
>>> from polls.models import Question
>>> q = Question.objects.first()
>>> m = QuestionMapper(q)
```

So we can create a new instance of our mapper and "bind" it to our model instance.

From now on, accessing attributes on the mapper instance will extract values from that model instance.

```
>>> m.question_text
"What's new?"
>>> m.pub_date
'2017-06-17 05:30:58+00:00'
```

Notice that the *pub_date* field came out as a string, in ISO-8601 format.

This works both ways. We can set values on our model via the mapper:

```
>>> m.question_text = "So, what is new?"
>>> q.question_text
'So, what is new?'
>>> m.pub_date = '1975-11-05 23:30:00'
>>> q.pub_date
datetime.datetime(1975, 11, 5, 23, 30)
```

See that the ISO-8601 string was converted back to a datetime instance.

There's also a helpful function to grab all the defined fields and return them as a dict:

```
>>> m._reduce()
{'pub_date': '2017-06-17 05:30:58+00:00', 'question_text': "What's new?", 'id': 1}
```

The built in RESTful views in *django-nap* use this method to create JSON serialisable data from your models.

Calculated Fields

What if, as well as the publication date, we want to provide the age?

We can define mapper fields that do "work" as simply as we would add `property` to a class:

Listing 2: polls/mappers.py

```
from django.utils.timesince import timesince

class QuestionMapper(mapper.ModelMapper):
    class Meta:
        model = models.Question
        fields = '__all__'

    @mapper.field
    def age(self):
        return timesince(self.pub_date)
```

Of interest here is that the *self* passed to the getter function is not the *QuestionMapper* class, but the object it is bound to - that is, our model instance.

2.2.2 Choice Mapper

The *ChoiceMapper* is just as simple:

Listing 3: polls/mappers.py

```
class ChoiceMapper(maper.ModelMapper):
    class Meta:
        model = models.Choice
        fields = '__all__'
```

2.2.3 Updates

Besides setting each field individually, *Mapper* provides two approaches to updating your instance: `_apply` and `_patch`. They update the instance from a dict, as well as validate the data passed.

`_apply` is used to update all the fields defined on the Mapper from a dict. If a field on the mapper is marked as *required*, but is not provided in the dict, this will be treated as an error.

Alternatively, `_patch` is used to update only the fields provided.

Any validation errors raised by fields will be gathered and raised in a single ValidationError exception at the end of processing. The errors will also be stored on the Mapper instance as `_errors`.

Readonly fields

But wait! We don't want to let people alter the Question a Choice is assignd to!

We need to mark that field as read only.

For fields discovered from models, we can override their readonly nature in the Meta:

Listing 4: polls/mappers.py

```
class ChoiceMapper(maper.ModelMapper):
    class Meta:
        model = models.Choice
        fields = '__all__'
```

(continues on next page)

(continued from previous page)

```
        readonly = {
            'question': True,
        }
```

And for a *field*, we can pass an argument when declaring it:

Listing 5: polls/mappers.py

```
@mapper.field(readonly=True)
def age(self):
    return timesince(self.pub_date)
```

This will mean *_apply* and *_patch* will ignore values for this field.

2.3 Views

Now it's time to make our data visible to the outside world.

django-nap builds on Django's Class-Based Generic Views.

2.3.1 Question List

Now it's time to add our question list endpoint.

First, we'll define a common `QuestionMixin` class to help hold common definitions for list and object views:

Listing 6: polls/views.py

```
from nap.rest import views

from . import mappers, models

class QuestionMixin:
    model = models.Question
    mapper_class = mappers.QuestionMapper
```

Next we'll define our `QuestionListView`, based on this and the `ListBaseView` fron nap:

Listing 7: polls/views.py

```
class QuestionListView(QuestionMixin,
                       views.ListBaseView):
    pass
```

As it is, this view won't do anything, as it has no `get`, `post` or other methods. What it does provide is Django's `MultipleObjectMixin`, along with nap's `MapperMixin` and `NapView` classes.

To add the default GET behavior for a list, we mix in the `ListGetMixin`:

Listing 8: polls/views.py

```
class QuestionListView(QuestionMixin,
                       views.ListGetMixin,
```

(continues on next page)

(continued from previous page)

```
                        views.ListBaseView):
    pass
```

The *ListGetMixin* adds a simple *get* method, which will return a list of mapped instances of our model.

Let's add our new view to the existing urls, but with a 'api/' prefix:

Listing 9: polls/urls.py

```
from django.conf.urls import url

from . import views

urlpatterns = [
    # ex: /polls/
    url(r'^$', views.index, name='index'),
    # ex: /polls/5/
    url(r'^(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
    # ex: /polls/5/results/
    url(r'^(?P<question_id>[0-9]+)/results/$', views.results, name='results'),
    # ex: /polls/5/vote/
    url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),

    url(r'^api/', include([
      url(r'question/$', views.QuestionListView.as_view()),
    ]))
]
```

So we can now access our list of Questions at http://localhot:8000/api/question/ and should see something like this:

```
[
   {
      "id": 1,
      "question_text": "What's new?",
      "pub_date": "2017-06-17 05:30:58+00:00",
      "age": "20\u00a0hours, 15\u00a0minutes"
   }
]
```

Nested Records

That's great, but a Question with no Choices isn't much use, is it?

We can ask our mapper to render a list of related records using a *ToMany* field:

Listing 10: polls/mappers.py

```
class QuestionMapper(mapper.ModelMapper):
   class Meta:
        model = models.Question
        fields = '__all__'

   @mapper.field
   def age(self):
        return timesince(self.pub_date)
```

(continues on next page)

(continued from previous page)

```
    choices = mapper.ToManyField('choice_set')
```

The *ToManyField* will check if its value is a `django.db.models.Manager`, and call *.all()* on it if it is.

And now out output will look something like this:

```
[
    {
        "id": 1,
        "age": "20\u00a0hours, 19\u00a0minutes",
        "question_text": "What's new?",
        "pub_date": "2017-06-17 05:30:58+00:00",
        "choices": [1, 2]
    }
]
```

By default, a *ToManyField* will only render the primary keys of the related objects. If you want to control how it's serialised, just specify a mapper on the field.

Listing 11: polls/mappers.py

```
choices = mapper.ToManyField('choice_set', mapper=ChoiceMapper)
```

Which will give us this output:

```
[
    {
        "pub_date": "2017-06-17 05:30:58+00:00",
        "age": "20\u00a0hours, 22\u00a0minutes",
        "question_text": "What's new?",
        "id": 1,
        "choices": [
            {
                "question": 1,
                "choice_text": "First Choice",
                "id": 1,
                "votes": 0
            },
            {
                "question": 1,
                "choice_text": "Another Choice",
                "id": 2,
                "votes": 0
            }
        ]
    }
]
```

We really don't need the question ID embedded there, so let's define a new choice mapper which will exclude that.

Listing 12: polls/mappers.py

```
class InlineChoiceMapper(mapper.ModelMapper):
    class Meta:
        model = models.Choice
```

(continues on next page)

(continued from previous page)

```
        fields = '__all__'
        exclude = ('question',)
```

And finally we see:


```
[
   {
      "choices": [
         {
            "votes": 0,
            "id": 1,
            "choice_text": "First Choice"
         },
         {
            "votes": 0,
            "id": 2,
            "choice_text": "Another Choice"
         }
      ],
      "question_text": "What's new?",
      "age": "20\u00a0hours, 27\u00a0minutes",
      "pub_date": "2017-06-17 05:30:58+00:00",
      "id": 1
   }
]
```


2.4 Authorisation

Because `django-nap` uses Django compatible Class-Based Views, you can simply use the same mixins provided by `django.contrib.auth`.

2.4.1 Login Required

Here is an example of a view which only permits logged in users to get/post Choices:

```
from django.contrib.auth.mixins import LoginRequiredMixin

class ChoiceListView(ChoiceMixin,
                     LoginRequiredMixin,
                     views.ListGetMixin,
                     views.ListPostMixin,
                     views.ListBaseView):
    pass
```

2.5 Old Mapper Tutorial

In this tutorial we will write a small django-nap powered RESTful service for a to-do list application. The tutorial has been tested against Django (1.8.3) and django-nap (0.30.4).

Instead of using a more 'traditional' *Serialiser* based approach to building the service, we will use nap's powerful *Mappers* and Django CBV mixins.

2.5.1 1. Setup

First things first, as with any Python programming application, we want to create a virtual environment sandbox for us to manage our applications dependencies. Let's get started by creating a virtual environment and activating it:

```
virtualenv -p python3 nap-todo
source nap-todo/bin/activate
```

If you see *(nap-todo)* prefixed to all of your terminal commands you'll know that you correctly created and activated the virtual environment.

Next we're going to need to install Django and django-nap in our virtual environment. Go ahead and execute the following commands to do that:

```
pip install django
pip install django-nap
```

Great! We've now installed Django and django-nap and are ready to start building our API service. Let's create a new Django project.

```
django-admin.py startproject todoproject
```

Change directory into the newly created todoproject directory. We'll now create a new Django app inside the todoproject.

```
cd todoproject
python manage.py startapp todoapp
```

Don't forget to add 'todoapp' to settings.INSTALLED_APPS!

That's great, our project directory is all set up and ready for us to start creating the models that we will use in our application.

2.5.2 2. Models

Our application is going to allow a *User* to create *Lists* of *Items*. *Items* represent task that are to be done. A *List* represents collections of *Items*. Each *Item* is associated with a *User* (from `django.contrib.auth`). Let's begin by adding the models we want to the todoapp/models.py file.

```
class List(models.Model):
    name = models.CharField(max_length=64)

    def __str__(self):
        return self.name


class Item(models.Model):
    title = models.CharField(max_length=64)
    list = models.ForeignKey('todoapp.List')
    completed = models.BooleanField(default=False)
    owner = models.ForeignKey('auth.User')

    def __str__(self):
        return self.title
```

Next we need to create a migration and migrate the database. In your terminal window execute the following commands:

```
python manage.py makemigrations
python manage.py migrate
```

Awesome let's move on to the next step.

2.5.3 3. Mappers

We need Mappers to reduce Python objects into simple data types supported by JSON and back again. nap's *Mappers* are an alternative approach to traditional *Serialisers*. They serve the same function, but do it in slightly different ways. A *Mapper* will map properties on itself to your object. This allows you to easily convert from JSON to Python objects and vice-versa.

Mapper for User

Let's start by creating a *Mapper* for the *User* model so that you can get a better feel for how it works. A *ModelMapper* is a shortcut that creates a *Mapper* and automatically generates a set of fields for you based on the model. Similarly to how *ModelForms* and *Forms* relate.

Let's create a new file in the todoapp directory called mappers.py and add the following code to your todoapp/mappers.py file:

```
from django.contrib.auth.models import User

from nap import mapper


class UserMapper(mapper.ModelMapper):
    class Meta:
        model = User
        fields = '__all__'
```

The *ModelMapper* will create a Mapper for us and all we need to tell it is which model we want to map, and which fields to use. As you can see we have told the *ModelMapper* to use __all__ of the User fields.

Mapper for List

Next let's add a *ModelMapper* for the *List* model. This should be very similar to the *ModelMapper* we created for the User model. Your todoapp/mappers.py file should now look like this:

```
from django.contrib.auth.models import User

from nap import mapper

from . import models # Don't forget this


class UserMapper(mapper.ModelMapper):
    class Meta:
        model = User
        fields = '__all__'


class ListMapper(mapper.ModelMapper):
```

(continues on next page)

(continued from previous page)

```
    class Meta:
        model = models.List
        fields = '__all__'
```

Mapper for Item

Next let's add a *ModelMapper* for the Item model. This ones a little different though because there are some more complicated fields in the *Item* model than there are in our *User* and *List* models. Let's start by implementing the parts of the *ItemMapper* we know. We're going to add a *ModelMapper* for *Item* to our code in the todoapp/mappers.py file so that it looks like this:

```
from django.contrib.auth.models import User

from nap import mapper

from . import models


class UserMapper(mapper.ModelMapper):
    class Meta:
        model = User
        fields = '__all__'


class ListMapper(mapper.ModelMapper):
    class Meta:
        model = models.List
        fields = '__all__'


class ItemMapper(mapper.ModelMapper):
    class Meta:
        model = models.Item
        fields = '__all__'
        exclude = ['owner', 'list']
```

As you can see we've defined the model and fields we want, but this time we're also telling the *ModelMapper* which fields to exclude. We're going to exclude the more complicated Foreign Key fields, owner and list, and deal with them later.

Now that we've got our *Mappers* implemented for all of our models, we can go on to create the URLs and views for our RESTful service.

2.5.4 4. Class-Based Views and URLs

Let's being by add a pattern for /api/ to our root url configuration (todoproject/urls.py). Your root url configuration should look something like this now:

```
from django.conf.urls import include, url
from django.contrib import admin


urlpatterns = [
```

(continues on next page)

(continued from previous page)

```
    url(r'^admin/', include(admin.site.urls)),
    url(r'^api/', include('todoapp.urls')),
]
```

You'll notice that we've used `include` to point all requests to /api/ on to `'todoapp.urls'` but if you've been following closely you'll realise we don't actually have a module called todoapp.urls! Let's fix that up quickly... create a urls.py file in the todoapp directory. Now we can edit the todoapp/urls.py file and start to think about what endpoints we want to create. I like to write mine in the urls.py file as comments, and uncomment them as I write the view code.

List of endpoints in words

1. Get a list of all of the `List` resources
2. Add a new List resource to the list of `List` resources
3. Get a single instance of a `List` resource
4. Get a list of all of the `Item` resources
5. Add a new Item resource to the list of `Item` resources
6. Get a single instance of an `Item` resource
7. Authenticate a users username and password combination

Let's add some endpoints (as comments) to the todoapp/urls.py file that will achieve this. I've added a comment next to each endpoint that explains which of the "List of endpoints in words" section the url will handle.

```
from django.conf.urls import include, url

from . import views


urlpatterns = [
    # /api/list/ # GET will deal with (1) and POST will deal with (2)
    # /api/list/<id>/ # GET will deal with (3)
    # /api/item/ # GET will deal with (4) and POST will deal with (5)
    # /api/item/<id>/ # GET will deal with (6)
    # /api/login/ # POST will deal with 7
]
```

Writing the view: list of List

Now that we know what endpoints we are planning to build, and what each will need to do we can create the views that will process the requests. We're going to start by implementing (1) which requires us to: "get a list of all of the `List` resources".

Lets add the following code to the todoapp/views.py file:

```
from nap.rest import views

from . import mappers
from . import models


class ListMixin:
```

(continues on next page)

(continued from previous page)

```
    model = models.List
    mapper_class = mappers.ListMapper


class ListListView(ListMixin,
                   views.ListBaseView):
    pass
```

Given we want to get a list of all the List resources, we will use the `nap.rest.views.ListBaseView` as a starting point. The ListBaseView combines ListMixin (which extends Django's MultipleObjectMixin) with View. From the Django docs: "MultipleObjectMixin can be used to display a list of objects." This sounds like what we need!

Adding GET functionality: list of List

We do however want to use `nap.rest.views.ListGetMixin` which provides the get() method for lists. This means the HTTP verb GET can now be used with our view. We need to update our `ListListView(views.ListBaseView)` class to include the `ListGetMixin` so let's do that.

Update your todoapp/views.py file to look like this:

```
from nap.rest import views

from . import mappers
from . import models


class ListMixin:
    model = models.List
    mapper_class = mappers.ListMapper


class ListListView(ListMixin,
                   views.ListGetMixin,
                   views.ListBaseView):
    pass
```

Adding POST functionality: list of List

We decided when planning our URLs, that to add a List resource to the list of Lists, we'd POST to the same url (/api/list/). That's as simple as including the `ListPostMixin` to the `ListListView`. This will provide the post() method which will allow us to use the POST HTTP verb.

Let's go ahead and do that now. Update your todoapp/views.py file to look like this:

```
from nap.rest import views

from . import mappers
from . import models


class ListMixin:
    model = models.List
    mapper_class = mappers.ListMapper
```

(continues on next page)

(continued from previous page)

```
class ListListView(ListMixin,
                   views.ListPostMixin,
                   views.ListGetMixin,
                   views.ListBaseView):
    model = models.List
    mapper_class = mappers.ListMapper
```

Defining the URL: list of List

One last thing before we take our API for a test drive. We need to uncomment the api endpoint for /api/list/ and actually write the proper URL pattern. We're going to cheat a little here and use the inbuilt Django `@csrf_exempt` decorator to bypass CSRF, but please ALWAYS use CSRF in production code.

Update your todoapp/urls.py to look like this:

```
from django.conf.urls import include, url
from django.views.decorators.csrf import csrf_exempt

from . import views


urlpatterns = [
    url(r'^list/$', csrf_exempt(views.ListListView.as_view())),
    # /api/list/<id>/ # GET will deal with (3)
    # /api/item/ # GET will deal with (4) and POST will deal with (5)
    # /api/item/<id>/ # GET will deal with (6)
    # /api/login/ # POST will deal with 7
]
```

You can see that we've mapped the list/ endpoint to ListListView class that we wrote earlier. Now that we have built the functionality to create Lists and view Lists it's time to see if our API works.

Testing with Python Requests: list of List

We'll use Python Requests (http://www.python-requests.org/) to POST a List object to our database. In a terminal window that you have activated your virtual environment in, run your HTTP server with `python manage.py runserver`. Open up a second terminal window, active your virtual environment as before. Install Requests with `pip install requests`. Open the Python interpreter by typing `python` at the console. This is not a tutorial on using requests so just enter this boilerplate code into your Python interpreter:

```
import requests
payload = {'name': 'my demo list'}
r = requests.post("http://127.0.0.1:8000/api/list/", params=payload)
r.status_code
```

The result of r.status_code should be `HTTP 201 Created`. This will confirm that we've created a list in our database with the name 'my demo list'. You can confirm this by looking at the admin interface at http://127.0.0.1:8000/admin. Remember you may need to create a superuser in order to use the admin interface.

So now that we've got a List instance in our database, we can execute a GET to the /api/list/ endpoint and we should receive a JSON response. We don't need to use Requests for this because our browser provides all the GET functionality that we need. Simply load the url http://127.0.0.1:8000/api/list/ in your browser and you should see a JSON

representation of all of the lists (at this stage only 1) in your database. You should play around with Requests and add some more List instances to the database.

Recap: list of List

So a quick recap of what we've done before we move on. We've created a *List* database model and a *ModelMapper* that maps our Python models to JSON and vice-versa. We've created a ListListView, which handles both GETing all our List instances in the database and POSTing new instances to our database. We've also then mapped our /api/list/ url to that view which allows external clients to use our API.

Not bad huh? We'll repeat the process and write view classes and corresponding url patterns for the other endpoints that we defined earlier.

Writing the views: object of List

We're now going to write the view that will return a single instance of a List object. Similar to how we used the `nap.rest.views.ListBaseView` mixin when writing our list of List view, we're now going to use the ObjectBaseView mixin. The ObjectBaseView combines ObjectMixin (which extends Django's SingleObjectMixin) with View. From the Django docs: "SingleObjectMixin provides a mechanism for looking up an object associated with the current HTTP request." Again, this sounds like what we need!

Lets add the following code to the todoapp/views.py file:

```
class ListObjectView(ListMixin,
                     views.ObjectBaseView):
    pass
```

Adding GET functionality: object of List

You should be getting a lot more comfortable with how nap uses the Django Class-Based View. Lets add GET functionality to our ListObjectView. In a similar fashion to how we have done throughout this tutorial we'll simply include one of the powerful mixins. Namely, the ListObjectView mixin.

The todoapp/views.py file should now look like this:

```
from nap.rest import views

from . import mappers
from . import models


class ListMixin:
    model = models.List
    mapper_class = mappers.ListMapper


class ListListView(ListMixin,
                   views.ListPostMixin,
                   views.ListGetMixin,
                   views.ListBaseView):
    pass


class ListObjectView(ListMixin,
```

(continues on next page)

(continued from previous page)

```
                    views.ObjectGetMixin,
                    views.ObjectBaseView):
    pass
```

Defining the URL: object of List

Lets quickly add a URL to actually call this view and then we can test to actually see if it works.

Add this url to your todoapp/urls.py file:

```
url(r'^list/(?P<pk>\d+)/$', csrf_exempt(views.ListObjectView.as_view())),
```

Again we're using the csrf_exempt() decorator for the sake of this tutorial.

Testing: object of List

We are only allowing the HTTP GET verb to be used with this view. That means we don't need to use Requests (although you certainly could) to test it. All you need to do is access the url we defined above with your web browser. Let's do just that and access the following url: http://127.0.0.1:8000/api/list/1/.

A quick explanation of what's happening here: the /1/ component of your URL corresponds to the (?P<pk>d+) regular expression in the url tuple. You can change the value of the pk component to retrieve an individual object view of any List instance. At this stage there's not much in a detail view - only the List title, but we're going to go on and add a bit more content next.

Quick pass through views for Item

So far we've built the GET and POST functionality for our List resource. You should be able to replicate the process we went through above and build GET and POST functionality for the Item resource yourself. I'm going to paste the code for that below, but I recommend you try do it yourself first! Note, the code below excludes the more complicated foreign key fields which we will build together.

Add the following to todoapp/views.py:

```
class ItemMixin:
    model = models.Item
    mapper_class = mappers.ItemMapper


class ItemListView(ItemMixin,
                   views.ListPostMixin,
                   views.ListGetMixin,
                   views.ListBaseView):
    pass


class ItemObjectView(ItemMixin,
                     views.ObjectGetMixin,
                     views.ObjectBaseView):
    pass
```

Don't forget to update todoapp/urls.py with the URL tuples that will call these views:

```
url(r'^item/$', csrf_exempt(views.ItemListView.as_view())),
url(r'^item/(?P<pk>\d+)/$', csrf_exempt(views.ItemObjectView.as_view())),
```

2.5.5 5. Update Mappers

Lets start modifying our *Mappers* so that we can serialise any extra fields, including related field sets and Foreign Key fields.

ListMapper: List item_set()

If we were writing a client application to consume the /api/list/ API endpoint, we would probably want to include all of the Item's that are in a List. Essentially that means we want to define a proxy field on the model, which means we're going to add another field called `items` to our Mapper.

Your ListMapper class in todoapp/mappers.py should look like this now:

```
class ListMapper(mapper.ModelMapper):
    class Meta:
        model = models.List
        fields = '__all__'

    @mapper.field
    def items(self):
        'Produces a list of dicts with pk and title.'
        return self.item_set.all()
```

You can see that we are using the `field` decorator to provide the get functionality we want. If you try to access the http://127.0.0.1:8000/api/list/1/ URL though, you'll notice Django raises a TypeError: `Item is not JSON serializable`. So we're going to use a handy shortcut and cast our item_set into a Python list.

Change the return line of the item so that your class looks like this:

```
class ListMapper(mapper.ModelMapper):
    class Meta:
        model = models.List
        fields = '__all__'

    @mapper.field
    def items(self):
        'Produces a list of dicts with pk and title.'
        return list(
            self.item_set.values()
        )
```

This will return a list of Item dictionaries - `[{<Item>},{<Item>} ... {<Item>}]`. Lets get rid of all the excess Item data and only return the pk's and and title's, change our queryset definition to this: `self.item_set.values('pk', 'title')`.

ItemMapper: get/set an owner (User)

When we create an Item object (via an HTTP POST) we will pass it an id value which represents the primary key of the User who owns it. That means we need to update our ItemMapper and tell it how to set the owner field (User foreign key). Again we'll use the `field` decorator to provide the get functionality we want.

Update your ItemMapper in todoapp/mappers.py to look like this:

```
class ItemMapper(mapper.ModelMapper):
    class Meta:
        model = models.Item
        fields = '__all__'
        exclude = ['owner', 'list']

    @mapper.field
    def owner_id(self):
        return self.owner_id
```

We're now telling the Mapper to include an owner_id field in the JSON representation of an Item, and to return the owner_id (which is the primary key of the owner field). Lets also now add the set functionality for this field. This will tell the Mapper how to take a JSON payload with an owner_id value and actually set the owner field on the model instance. Again we'll use the built in decorators to perform this, we'll use the `setter` decorator to provide the set functionality.

Update your ItemMapper in todoapp/mappers.py to look like this:

```
class ItemMapper(mapper.ModelMapper):
    class Meta:
        model = models.Item
        fields = '__all__'
        exclude = ['owner', 'list']

    @mapper.field
    def owner_id(self):
        return self.owner_id

    @owner_id.setter
    def owner_id(self, value):
        try:
            self.owner = User.objects.get(pk=value)
        except models.User.DoesNotExist:
            raise ValidationError("Invalid owner_id")
```

Recap

You can see that we have modified our *Mappers* to use the `field` and `setter` decorators to provide the get/set functionality. The `field` decorator extends the builtin `property`, and so supports `@x.setter` and `@x.deleter` for setting the setter and deleter functions.

2.5.6 6. Authorisation

nap does not provide authentication, but it is very easy to combine nap with Django's authentication system, or any other third party authentication applications.

nap does provide authorisation through a `permit` decorator. You can use it to control the permissions of any handler method. We're going to create a login view that will authorise a user using the Django authentication system. This means we'll be able to make use of Django's inbuilt forms too.

In your views.py add the following class:

```
from django.contrib import auth as django_auth # Don't forget this
from django.contrib.auth.forms import AuthenticationForm # Don't forget this
```

(continues on next page)

(continued from previous page)

```
from nap import http # Don't forget this


class LoginView(views.ObjectBaseView):
    mapper_class = mappers.UserMapper

    def get(self, request):
        if request.user.is_authenticated():
            return self.single_response(object=request.user)
        return http.Forbidden()

    def post(self, request):
        if request.user.is_authenticated():
            django_auth.logout(request)
            return self.get(request)
        form = AuthenticationForm(request, self.get_request_data())
        if form.is_valid():
            django_auth.login(request, form.get_user())
            return self.get(request)
        return self.error_response(form.errors)
```

We have defined a ObjectBaseView that will allow get() and post(). If logged in, GET will return a serialised representation of the User, and if not logged in will return an HTTP 403. If not logged in, POST will authenticate the User and either log them in, or return an error dictionary. POSTing to this view when already logged in will log the User out.

2.5.7 7. Permissions

Now that we have created an authorisation endpoint and view, we can now leverage Django's build in authentication mixins to control access.

We've decided we only want to allow logged in users to post new messages, so we mix in the *UserPassesTestMixin* to the ListListView class. All we need is to add a *test_func* to only check if a user is authentencated if it's a POST.

```
from django.contrib.auth.mixins import UserPassesTestMixin

...

class ListListView(UserPassesTestMixin,
                   ListMixin,
                   views.ListPostMixin,
                   views.ListGetMixin,
                   views.ListBaseView):

    def test_func(self):
        if self.request.method == 'POST':
            return self.user.is_authenticated:
        return True
```

Let's update our Item related views to only allow authorised Users to GET and POST. We'll use Dango's provided *LoginRequiredMixin*.

Update the ItemListView class in todoapp/views.py to look like this:

```
from django.contrib.auth.mixins import LoginRequiredMixin

...

class ItemListView(LoginRequiredMixin,
                   ItemMixin,
                   views.ListPostMixin,
                   views.ListGetMixin,
                   views.ListBaseView):
    pass
```

2.5.8 8. Finished!

Well done. We've finished building our API service!

CHAPTER 3

Mappers

3.1 Fields

3.1.1 The *field* decorator

The *field* decorator works exactly like *property*, however it will operate on the "bound" object, not the Mapper.

class field

Parameters

- **required** – Is this field required? Default: True
- **default** – The value to use if the source value is absent. May be a callable that takes no arguments.
- **readonly** – Can the field be updated? Default: True
- **null** – Is None a valid valie? Default: False

The decorator can be used bare, or with arguments:

```
class M(Mapper):

    @mapper.field
    def foo(self):
        return self.bar

    @mapper.field(default=0)
    def baz(self):
        return self.qux

    @baz.setter
    def baz(self, value):
        self.qux = value
```

As you can see, both the getter and setter of a *field* are defined the same way as with *property*.

3.1.2 Basic fields

For simple cases where the descriptor protocol is overkill, there is the *Field* class.

class Field(...)

Parameters

- **attr** – The name of the attribute on the bound object it gets/sets.
- **required** – Is this field required? Default: True
- **default** – The value to use if the source value is absent.
- **readonly** – Can the field be updated? Default: True
- **null** – Is None a valid valie? Default: False

```
class M(Mapper):
    foo = Field('bar')
    baz = Field('qux', default=0)
```

There are also typed fields:

- BooleanField
- IntegerField
- FloatField
- TimeField
- DateField
- DateTimeField

These will ensure the values stored are of the correct type, as well as being presented to JSON in a usable format.

3.1.3 Accessing extra state

Sometimes when serialising an object, you need to provide additional state. This can be done using a `context_field`, which subclasses `field`, but passes any extra `kwargs` that were pased to *Mapper* instance *context* to the getter and setter methods as an extra argument.

class context_field

Parameters

- **required** – Is this field required? Default: True
- **default** – The value to use if the source value is absent.
- **readonly** – Can the field be updated? Default: True
- **null** – Is None a valid valie? Default: False

The following is an example from the test suite:

```
class M(Mapper):
    @fields.context_field
    def scaled(self, context):
        return self.value * context['factor']

    @scaled.setter
```

(continues on next page)

(continued from previous page)

```
    def scaled(self, value, context):
        obj.value = value // self._context['factor']

m = M(o, factor=10)
```

Accessing `m.scaled` will now return the value multiplied by 10.

Models

3.1.4 Relation Fields

To help with relations, the models module includes two extra field types:

- ToOneField
- ToManyField

Both accept the same extra arguments:

class RelatedField

Parameters

- **model** – The model this field relates to
- **mapper** – (Optional) the mapper to use to reduce instances.

When the mapper is omitted, only the Primary Key of the related model will be used.

The `ToManyField` will work on any iterable, however if it's passed a `Manager` it will call `.all()` before iterating it. This makes it ideally suited for `ManyToMany` and reverse `ForeignKey` accessors.

3.2 Mapper API

All properties and methods are prefixed with _ to avoid polluting the namespace for your public fields.

class Mapper(*obj=None*, ***kwargs*)

_fields
A dict of (name: field) for all fields on this mapper.

_field_names
A list of field names on this mapper.

_reduce()
Returns a dict containing all the field values on the currently bound object.

_clean(*data*, *full=True*)
Allows whole-object validation.

Should update self._errors dict with any new validation errors.

The *full* flag indicates if this is an _apply (True) or _patch (False) cycle.

_patch(*data*)
Update all properties on this mapper supplied from the dict `data`.

Any omitted fields will be skipped entirely.

_apply(*data*)
Update all properties on this mapper fron the dict `data`.

If a field is marked as *required* it must have either a value provided, or a default specified.

All ValidationErrors raised by fields and their filters will be collected in a single ValidationError. You can access this dict via the exception's *error_dict* property.

As the name suggests, a *Mapper* will map properties on themselves to your object. They allow you to easily write proxy objects, primarily for converting between serialised (JSON) and live (Python) formats of your resources.

Warning: Since a Mapper instance retains a reference to the object they are bound to, even when using << syntax, instances MUST NOT be shared between threads.

3.3 field decorator: get/set

Mappers work using Python's descriptor protocol, which is most commonly used via the `property` built-in. This gives you full control over a Mapper's properties. When constructing a Mapper you can pass an object for it to "bind" to. All attribute access to the Mapper fields will proxy to this bound object.

Here's an example to illustrate some of these concepts:

```
# An object we want to create a Mapper for
class Person:

    def __init__(self, first_name, last_name, is_alive):
        self.first_name = first_name
        self.last_name = last_name
        self.is_alive = is_alive


from nap import mapper

# A Mapper that we are creating for the Person object
class PersonMapper(mapper.Mapper):
    '''
    The self argument refers to the object we bind the Mapper to when we
    construct it. It DOES NOT refer to the instance of the PersonMapper.
    '''
    @mapper.field
    def name(self):
        return '{}'.format(self.first_name, self.last_name)

    # We can use the Field class for simpler cases
    first_name = mapper.Field('first_name')
    last_name = mapper.Field('last_name')
    is_alive = mapper.Field('is_alive')

# Construct instances of the Person and a Mapper classes
person = Person('Jane', 'Doe', 22, True)
mapper = PersonMapper(person)
```

See **'Fields'_** for more details.

3.4 Mapper functions

A Mapper supports several methods:

`_reduce()` will reduce the instance to its serialisable state, returning a dict representation of the *Mapper*.

`_patch(data)` will partially update (patch) a *Mapper*'s fields with the values you pass in the data dict. If validation fails it will raise a *ValidationError*.

`_apply(data)` will fully update (put) a *Mapper*'s fields with the values you pass in the data dict. If you don't pass a field in the data dict it will try to set the field to the default value. If there is no default and the field is required it will raise a *ValidationError*.

`_clean(data, full=True)` is a hook for final pass validation. It allows you to define your own custom cleaning code. You should update the `self._errors` dict. The `full` boolean indicates if the calling method was `_apply` (True) or `_patch` (False).

Here is some code to explain how these concepts work. We will continue to use the Person class and PersonMapper class defined above.

Note that these methods only update the fields of the model instance. You will need to call save() yourself to commit changes to the database.

Using _reduce:

```
p = Person('Jane', 'Doe', True)
m = PersonMapper(p)
reduced_p = m._reduce()
print(reduced_p)

# Output: {'first_name': 'Jane', 'last_name': 'Doe', 'is_alive': True}
```

Using _apply:

```
m = PersonMapper()
m._apply({
    "first_name": "Jane",
    "last_name": "Doe",
    "is_alive": False
})
reduced = m.reduce()
print(reduced)

# Output: {'first_name': 'Jane', 'last_name': 'Doe', 'is_alive': False}
```

Using _patch:

```
p = Person('Jane', 'Doe', True)
m = PersonMapper(p)
m._patch({"last_name": "Notdoe"}) # This should patch last_name
reduced = m.reduce()
print(reduced)

# Output: {'first_name': 'Jane', 'last_name': 'Notdoe', 'is_alive': True}
```

Using _clean:

```
class DeadPersonMapper(PersonMapper):
    def _clean(self):
```

(continues on next page)

(continued from previous page)

```
        if self.is_alive:
            raise ValidationError("Only dead people accepted to the morgue.")

m = DeadPersonMapper()
m._apply({'last_name': 'Doe', 'first_name': 'John', 'is_alive': True})

# ValidationError
```

3.4.1 Shortcuts

As a convenience, Mappers support two shorthand syntaxes:

```
>>> data = mapper << obj
```

This will bind the mapper to the obj, and then call `_reduce`.

```
>>> obj = data >> mapper
```

This will call `_patch` on the mapper, passing data, and returning the updated object.

3.5 ModelMappers

A `ModelMapper` will automatically create a `Mapper` for a Django model. A `ModelMapper` behaves very similar to a Django `ModelForm`, you control it by setting some fields in an inner `Meta` class.

The fields that can be set are:

class Meta

model
Default: None

The model this `Mapper` is for

fields
Default: []

The list of fields to use. You can set it to '__all__' to map all fields.

exclude
Default: []

The list of fields to exclude from the Model

required
Default: {}

A map to override required values for fields auto-created from the Model.

readonly
Default: []

The list of fields which are read only.

Must not conflict with *required*.

You can rewrite the Mapper so that it subclasses ModelMapper. Here's a new Person object that subclasses Django's models.Model:

```
from django.db import models


# An Django models.Model we want to create a Mapper for
class Person(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    is_alive = models.BooleanField(default=True)
```

Here is the PersonMapper rewritten to use a ModelMapper:

```
from nap import mapper

# This should reference the model package where we define Person
from . import models


class PersonMapper(mapper.ModelMapper):
    class Meta:
        model = models.Person
        fields = '__all__'
```

You can still use *field* to get/set properties and fields on a ModelMapper. This is useful when the model contains some properties that the ModelMapper cannot understand, or when you want to customise how certain fields are represented.

To illustrate this we will add a new Django field (models.UUIDField) to our model. UUIDField does not have a filter built in to nap, so you will need to define your own get and set functionality using the *field* decorator.

Here is a Person model object with a UUIDField:

```
from django.db import models


# An Django models.Model we want to create a Mapper for
class Person(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    is_alive = models.BooleanField(default=True)
    uuid = models.UUIDField(default=uuid.uuid4, editable=False)
```

And here is a complete ModelMapper that will correctly handle this new type of field:

```
from nap import mapper

from . import models


class PersonMapper(mapper.ModelMapper):
    class Meta:
        model = models.Person
        fields = '__all__'

    @mapper.field(readonly=True)
    def uuid(self):
        return str(self.uuid) # Remember: self refers to the bound object.
```

CHAPTER 4

Class-Based Views

Also included are some mixins for working with Django's Class-Based Generic Views. As they follow the existing CBV interfaces, they are compatible with existing decorators and other utilities.

At their core is the *MapperMixin*, which extends the *:class:JsonMixin <nap.utils.JsonMixin>*. This provides ways to get the mapper to use for the request, and utility functions for returning empty, single object, and multiple object responses.

Additionally it provides wrappers for these to use specific response codes, which can be configured on the class also.

4.1 Base Classes

class MapperMixin

All of the following classes are based on this.

response_class

The class to construct responses from.

Default: nap.http.JsonResponse

content_type

The default content type for responses.

Default: 'application/json'

mapper_class

You must set this to the `Mapper` to use when processing requests and responses.

ok_status

Default: nap.http.STATUS.ACCEPTED

accepted_status

Default: nap.http.STATUS.CREATED

created_status

Default: nap.http.STATUS.NO_CONTENT

error_status
Default: nap.http.STATUS.BAD_REQUEST

HTTP Status codes to use for different response types.

get_mapper(*obj=None*)
Returns an instance of *mapper_class*

empty_response(***kwargs*)
Returns an instance of *response_class* with no content.

single_response(***kwargs*)
Return a response with a single object.

Will use *self.object* if *object* is not passed, or call *self.get_object* if *self.object* does not exist.

Will use *self.mapper* if *mapper* is not passed, or call *self.get_mapper* if *self.mapper* does not exist.

multiple_response(***kwargs*)
Return a response with a list of objects.

Will use *self.object_list* if *object_list* is not passed, or call *self.get_queryset()* if *self.object_list* does not exist.

Will use *self.mapper* if *mapper* is not passed, or call *self.get_mapper()* if *self.mapper* does not exist.

Will apply pagination if *self.paginate_by* is set or *self.include_meta* is True.

get_meta(*page*)
Returns pagination metadata for paginated lists.

accepted_response(***kwargs*)
Returns an empty response with `self.accepted_status`

created_response(***kwargs*)
Returns a single response with `self.created_status`.

deleted_response(***kwargs*)
Returns an empty response with `self.deleted_status`.

error_response(*error*)
Passes the supplied error dict through nap.utils.flatten_errors, and returns it with status=self.error_status

4.2 List Classes

class ListMixin(*MapperMixin*, *MultipleObjectMixin*)
Base list mixin, extends Django's MultipleObjectMixin.

ok_response(***kwargs*)

Calls `self.list_response(status=self.ok_response)`

class ListGetMixin
Provides `get()` for lists.

class ListPostMixin
Provides `post()` for lists.

post_invalid(*errors*)

post_valid(***kwargs*)

class ListBaseView(*ListMixin*, *View*)

4.3 Single Object Classes

class ObjectMixin(*MapperMixin*, *SingleObjectMixin*)
Base single object mixin, extends Django's SingleObjectMixin.

ok_response(***kwargs*)
Calls self.single_response(status=self.ok_status)

class ObjectGetMixin
Provides `get()` for single objects.

class ObjectPutMixin
Provides `put()` for single objects.

put_valid(***kwargs*)

put_invalid(*errors*)

class ObjectPatchMixin
Provides `patch()` for single objects.

patch_valid(***kwargs*)

patch_invalid(*errors*)

class ObjectDeleteMixin
Provides `delete()` for single objects.

delete_valid(***kwargs*)

class ObjectBaseView(*ObjectMixin*, *View*)

4.3.1 Example

Sample `views.py` that provides `GET`, `PUT`, `PATCH`, and `DELETE` methods for the Poll model:

```
from nap.mapper import ModelMapper
from nap.rest.views import (
    ObjectGetMixin, ObjectPutMixin, ObjectPatchMixin, ObjectDeleteMixin,
    ObjectBaseView,
)

from .models import Poll


class PollMapper(ModelMapper):
    class Meta:
        model = Poll
        fields = ['question', 'pub_date']


class PollDetailView(ObjectGetMixin,
                     ObjectPutMixin,
                     ObjectPatchMixin,
                     ObjectDeleteMixin,
                     ObjectBaseView):
    model = Poll
    mapper_class = PollMapper
```

4.3.2 Example: Updating two objects

Here's an example of updating two related objects in a single PATCH call.

```
class UserDetailView(ObjectGetMixin, ObjectBaseView):
    model = User
    mapper_class = UserMapper

    def patch(self, request, *args, **kwargs):
        data = self.get_request_data({})

        self.object = user = self.get_object()

        errors = {}

        mapper = self.get_mapper(user)
        try:
            data >> mapper  # This is shorthand for _patch
        except ValidationError as e:
            errors.update(dict(e))

        profile_mapper = ProfileMapper(user.profile)
        try:
            data >> profile_mapper  # This is shorthand for _patch
        except ValidationError as e:
            errors.update(dict(e))

        if errors:
            return self.patch_invalid(errors)

        user.save()
        user.profile.save()

        return self.ok_response(object=user, mapper=mapper)
```

4.3.3 Example: Customising GET

Here's an example of customising a GET call based on a querystring:

```
class QuestionListView(ListGetMixin, ListBaseView):
    model = Question
    mapper_class = QuestionMapper

    def get(self, request, *args, **kwargs):
        qset = self.get_queryset()

        # Apply filtering to get only questions for a particular poll
        poll_id = request.GET.get('poll_id')
        if poll_id:
            qset = qset.filter(poll_id=poll_id)

        self.object_list = qset
        return self.ok_response(object_list=qset)
```

CHAPTER 5

RPC

The RPC View allows your application to provide APIs that don't mate up with REST patterns.

5.1 Overview

Any POST request with a `X-RPC-Action` header will be intercepted and treated as a RPC request. If there is a method on the view class which matches the name in the header, and it's been decorated as `@method` accessible, the request data will be parsed, passed as keyword arguments to the method, and the result JSON encoded and returned.

5.2 Usage

Define a View using the Mixin:

```
from nap import rpc

class MathView(rpc.RPCView):

    @rpc.method
    def add(self, a, b):
        return a + b
```

Add it to your URL patterns:

```
url(r'^rpc/$', MathView.as_view(), name'rpc-view'),
```

Invoke it from Javascript:

```
fetch('/rpc/', {
    method: 'POST',
    body: JSON.stringify({a: 5, b: 10}),
    headers: {
```

(continues on next page)

(continued from previous page)

```
        'X-RPC-Action': 'add',
        'Content-Type': 'application/json'
    },
})
.then(resp => resp.json())
.then(data => alert(`Result is: ${data}`);  // "Result is: 15"
```

CHAPTER 6

Extras

There are some extra tools provided to ease your development of APIs.

6.1 HTTP Utilities

In nap.http is a set of tools to go one step further than Django's existing HttpResponse.

6.1.1 Status

Firstly, there is STATUS_CODES, which is a list of two-tuples of HTTP Status codes and their descriptions.

Also, and more usefully, there is the STATUS object. Accessing it as a dict, you can look up HTTP status code descriptions by code:

```
>>> STATUS[401]
'Unauthorized'
```

However, you can also look up attributes to find out the status code:

```
>>> STATUS.UNAUTHORIZED
401
```

This lets it act as a two-way constant.

6.1.2 BaseHttpResponse

This class blends Django's HttpResponse with Python's Exception. Why? Because then, when you're nested who-knows how deep in your code, it can raise a response, instead of having to return one and hope everyone bubbles it all the way up.

- BaseHttpResponse

- HttpResponseSuccess
  - OK
  - Created
  - Accepted
  - NoContent
  - ResetContent
  - PartialContent
- HttpResponseRedirect
  - MultipleChoices
  - MovedPermanently*
  - Found*
  - SeeOther*
  - NotModified
  - UseProxy*
  - TemporaryRedirect
  - PermanentRedirect

  Items marked with a * require a location passed as their first argument. It will be set as the `Location` header in the response.
- HttpResponseError

  A common base class for all Error responses (4xx and 5xx)
- HttpResponseClientError(HttpResponseError)
  - BadRequest
  - Unauthorized
  - PaymentRequired
  - Forbidden
  - NotFound
  - MethodNotAllowed
  - NotAcceptable
  - ProxyAuthenticationRequired
  - RequestTimeout
  - Conflict
  - Gone
  - LengthRequired
  - PreconditionFailed
  - RequestEntityTooLarge
  - RequestURITooLong

        * UnsupportedMediaType
        * RequestedRangeNotSatisfiable
        * ExpectationFailed
    - HttpResponseServerError(HttpResponseError)
        * InternalServerError
        * NotImplemented
        * BadGateway
        * ServiceUnavailable
        * GatewayTimeout
        * HttpVersiontNotSupported

It will be clear that, unlike Django, these mostly do not start with HttpResponse. This is a personal preference, in that typically you'd use:

```
from nap import http

...
    return http.Accept(...)
```

except_response

In case you want to use these raiseable responses in your own views, Nap provides a *except_response* decorator.

```
from nap.http.decorators import except_response

@except_response
def myview(request):
    try:
        obj = Thing.objects.get(user=request.user)
    except:
        raise http.BadRequest()
    return render(...)
```

The decorator will catch any *http.BaseHttpResponse* exceptions and return them as the views response.

6.1.3 Http404 versus http.NotFound

Generally in your API, you'll want to prefer http.NotFound for returning a 404 response. This avoids being caught by the normal 404 handling, so it won't invoke your handler404.

6.2 Simple CSV

A generator friendly, unicode aware CSV encoder class built for speed.

```
>>> csv = Writer(fields=['a', 'b', 'c'])
>>> csv.write_headers()
u'a,b,c\n'
```

(continues on next page)

(continued from previous page)

```
>>> csv.write([1, '2,', 'c'])
u'1,"2,",c\n'
```

Options:

Seprator:

SEP = u’,’

Quote Character:

QUOTE = u’”’

What to replace a QUOTE in a field with

ESCQUOTE = QUOTE + QUOTE

What to put between records

LINEBREAK = u’n’

ENCODING = ‘utf-8’

6.3 Actions

CHAPTER 7

Examples

Sometimes, an example is much easier to understand than abstract API docs, so here's some sample use cases.

7.1 Case 1: Simple Blog API

7.1.1 models.py

```
from django.db import models
from taggit.managers import TaggableManager

class Post(models.Model):
    title = models.CharField(max_length=255)
    author = models.ForeignKey('auth.User')
    published = models.BooleanField(default=False)
    content = models.TextField(blank=True)
    tags = TaggableManager(blank=True)
```

7.1.2 mappers.py

```
from nap import mapper

class PostMapper(mapper.ModelMapper):
    class Meta:
        model = models.Post

    @mapper.field(readonly=True)
    def tags(self):
        return list(obj.tags.values_list('name', flat=True))
```

7.1.3 views.py

```
from nap.rest import views

from . import mappers, models


class PostMixin:
    model = models.Post
    mapper_class = mappers.PostMapper


class PostList(PostMixin,
               views.ListGetMixin,
               views.BaseListMixin):
    paginate_by = 12


class PostDetail(PostMixin,
                 views.ObjectGetMixin,
                 views.BaseObjectMixin):
    pass
```

7.1.4 urls.py

```
from django.conf.urls import include, url

from . import views

urlpatterns = [
    (r'^api/', include([
        url(r'^post/$',
            views.PostList.as_view(),
            name='post-list'),
        url(r'^post/(?P<pk>\d+)/$',
            views.PostDetail.as_view(),
            name='post-detail'),
    ])),
]
```

7.2 Case 2: Login View

Once you've defined a `Mapper` for your *User* model, you can provide this Login endpoint:

```
from django.contrib import auth
from django.contrib.auth.forms import AuthenticationForm
from django.utils.decorators import classonlymethod
from django.views.decorators.csrf import ensure_csrf_cookie

from nap import http
from nap.rest import views

from . import mappers
```

(continues on next page)

(continued from previous page)

```
class LoginView(views.ObjectBaseView):
    mapper_class = mappers.UserMapper

    @classonlymethod
    def as_view(cls, *args, **kwargs):
        view = super().as_view(*args, **kwargs)
        return ensure_csrf_cookie(view)

    def get(self, request):
        '''Returns the current user's details'''
        if request.user.is_authenticated():
            return self.single_response(object=request.user)
        return http.Forbidden()

    def post(self, request):
        form = AuthenticationForm(request, self.get_request_data({}))
        if form.is_valid():
            auth.login(request, form.get_user())
            return self.get(request)
        return self.error_response(form.errors)
```

Note that it decorates *as_view* with *ensure_csrf_cookie*. This ensures the CSRF token is set if your site is a SPA.

You could even use the `DELETE` HTTP method for logout.

```
def delete(self, request):
    auth.logout(request)
    return self.deleted_response()
```

CHAPTER 8

Changelog

8.1 Current

8.1.1 v0.40.0 (2018-??-??)

Python and Django version support change

As of this release, Django 2.0+ is required.

As a result, Python 3.4+ is also required.

Incompatible Changes:

- `nap.rest.views.BaseListView` and `name.rest.views.BaseObjectView` have been renamed to `nap.rest.views.ListBaseView` and `nap.rest.views.ObjectBaseView` respectively.
- `nap.views.rest.MapperMixin.error_response` now calls `get_json_data` on the `errors` parameter.

Enhancements:

- Mappers now use `django.forms.utils.ErrorDict` and `django.forms.utils.ErrorList` for errors.
- `nap.utils.JsonMixin.get_request_data` now uses `cgi.FieldStorage` to parse request data, allowing it to support files in multi-part bodies for PUT and PATCH requests.
- `nap.mapper.fields.field` will now default to *readonly* = *True* unless a setter is specified.

Removed:

- `nap.utils.flatten_errors` has been removed.

Bug Fixes:

- Make *except_response* use *functools.update_wrapper* to not disguise the view function.

8.2 History

8.2.1 v0.30.11 (2017-03-31)

Enhancements:

- Simplify `nam.mapper.base.MetaMapper` discovering fields.
- Simplify `nap.mapper.models.MetaMapper` accordingly.

Bug Fixes:

- Added a dummy *post* method to `nap.rpc.views.RpcMixin` so `View` believes *POST* is an acceptable method.
- Fix fallback when looking for object/object_list/mapper in REST views.

8.2.2 v0.30.10 (2017-08-24)

Enhancements:

- JsonMixin will use request.content_type and request.content_params in Django 1.10+, instead of parsing them itself.
- NapView now decorates the as_view response with except_response, instead of overriding dispatch.
- Simplified code that builds ModelMapper
- RPCView now uses NapView to handle exception responses.
- Micro-optimsations for `nap.extras.simplecsv.Writer`

Bug Fixes:

- A `TypeError` or `ValueError` raised in `nap.mapper.Field.set` will now be caught and raised as a `ValidationError`.

8.2.3 v0.30.9 (2017-06-26)

Enhancements:

- Add *MapperMixin.include_meta* as an override to including meta in responses.
- Moved *StreamingJSONResponse* into *nap.http.response*
- Moved *except_response* into *nap.http.decorators*

8.2.4 v0.30.8 (2017-06-09)

Enhancements:

- *MapperMixin.get_meta(page)* was added to allow customising of meta-data in response.

Deprecations:

- *MapperMixin* no longer provides default values of None for *mapper*, *object* and *object_list*.

8.2.5 v0.30.7 (2017-06-07)

The prototype for *context_field* has been changed.

See documentation for details.

Enhancements:

- Allow *ModelMapper* to inherit its *Meta* from a parent.
- Add `nap.http.StreamingJSONResponse`.
- Add tools to help support generators in JSON encoding
- Changed *context_field* to pass the Mapper._context as the last argument to the getter and setter methods.

8.2.6 v0.30.6 (2017-05-29)

Enhancements:

- Added custom __set__ method for *ToManyField* so it can call *set* on the manager.

Bug Fixes:

- Don't replace inherited fields with auto-added model fields.
- Return a list of PKs if no Mapper provided to ToManyField

8.2.7 v0.30.5 (2017-05-27)

Enhancements:

- Added pagination support to *MapperMixin.multiple_response*
- Import all fields into mapper namespace
- Allow passing **kwargs to all CBV *valid_FOO* methods.

Bug Fixes:

- Call *all()* on *Manager* instances in *ToManyfield*

Deprecations:

- Removed *newrelic* module, as it was only to support Publishers.

8.2.8 v0.30.4 (2017-05-25)

Enhancements:

- Added *nap.utils.NapJSONEncoder* to support *__json__* protocol.

Bug Fixes:

- Fixed __new__ on field so subclasses work.
- Reworked context_field to work properly, and match docs.

8.2.9 v0.30.3 (2017-05-24)

Enhancements:

Bug Fixes:

- Using @foo.setter on a field will now retain other keyword arguments.
- RPCClient now sets Content-Type on request.

8.2.10 v0.30.2 (2017-05-23)

Enhancements:

- Began documenting the *extras* module.

Bug Fixes:

- Fixed ToOneField to reference *self.related_model* not *self.model*

8.2.11 v0.30.1 (2017-05-18)

Enhancements:

- fields will now raise and error when trying to set a value with no setter.

Bug Fixes:

- Include null in field constructor
- Set null correctly in fields on ModelMapper
- Handle null ForeignKey/OneToOneFields properly

Deprecations:

- Dropped *nap.utils.digattr* and *nap.mapper.fields.DigField*
- Typed fields no longer special case None

8.2.12 v0.30.0 (2017-05-16)

Python and Django version support change

Support for Python2 has been dropped.

Support for Django 1.7 is no longer tested.

Warning: API Breakage

Another large code reorganisation was undertaken. *DataMapper* has been renamed to *Mapper*, and large amounts of its cod have been rewritten.

Filters are no longer supported.

Enhancements:

- Added *readonly* attribute to *Mapper.field*
- Added *readonly* list to *ModelMapper.Meta*
- All *MapperMixin.*_response* methods now accept kwargs, and try to *setdefault* their default behaviour in it.
- In PUT/POST REST views, the *_valid* methods now accept kwargs and pass them to their response class.
- Added RPC Client example code.
- Dropped deprecated test class.

Bug Fixes:

- Corrected *Mapper* to work as documented for `obj = data << dm`

8.2.13 v0.20.3 (2017-05-09)

Bug Fixes:

- Handle None values properly in ModelFilter

8.2.14 v0.20.2 (2017-05-06)

Enhancements:

- Added *nap.http.except_response* decorator to handle exceptional responses in view functions.
- Finished updating *nap.extras.actions.ExportCsv* to work with DataMappers.

8.2.15 v0.20.1 (2017-05-04) . . . be with you!

Bug Fixes:

- Remove *default_app_config* [Thanks nkuttler]

8.2.16 v0.20.0 (2017-04-24)

WARNING: Major Refactor

All code related to Publishers and Serialisers have been removed.

Many

Enhancements:

- Add a common base class *HttpResponseError* for Status 4xx and 5xx responses.
- JsonMixin imports settings late to avoid problems

Deprecation:

- Removed backward compatibility shim for JsonResponse, now that we require Django 1.7
- Removed Publishers
- Removed Serialisers
- Removed auth - use Django's built in mixins.

- Removed SerialisedResponseMixin

8.2.17 v0.14.9 (2015-12-08)

Enhancements:

- Dropped support for testing in older Django
- Add ModelFilter to ForeignKeys in ModelDataMapper
- Allow passing kwargs to *JsonMixin.loads* and *JsonMixin.dumps*
- Added ability to change the response class used in auth decorators.
- Added >>= to *ModelDataMapper* to allow applying to new model instance.

Bug Fixes:

- Add any fields not in a supplied *Meta.fields* for a *ModelDataMapper* to the excludes list to ensure model validation also excludes them.
- Fixed *utils.JsonClient* to actually work.
- Propery handle encoding in *JsonMixin.get_request_data* for PUT and PATCH.

8.2.18 v0.14.8 (2015-10-12)

Enhancements:

- Added *Ripper* class to utils.
- Use *six.moves.urllib* instead of our own try/except on import
- Micro-optimisation: Calculate fields and field names for DataMappers at declatation
- Added *NapView* to *nap.rest.views* to handle when custom *http* responses are raised.
- Change default DELETE response to be empty
- Added *nap.rest.views.NapView* to catch and return *nap.http.BaseHttpResponse* exceptions as responses.

Bug Fixes:

- Set safe=False in MapperMixin.empty_response

8.2.19 v0.14.7.1 (2015-09-29)

Enhancements:

- Simplified *auth.permit_groups*

Bug Fixes:

- On a DataMapper, if a Field's default is callable, call it.
- Make _CastFiler and Date/Time filters skip to_python if value is of right type already.

8.2.20 v0.14.7 (2015-09-29)

Enhancements:

- Allow passing extra arguments to MapperMinix.ok_response
- Add *required* and *default* options for datamapper.field
- Add *LoginRequiredMixin* and *StaffRequiredMixin* to *nap.rest.auth*
- Allow use of custom JSONEncoder/JSONDecoder with JsonMixin

8.2.21 v0.14.6 (2015-09-14)

Enhancements:

- Make MapperMixin.single_response and MapperMixin.multiple_response get mapper, object, and queryset if none is provided.
- Dropped testing support for older versions of Django
- Added DataMapper tutorial to docs (Thanks limbera!)
- Added ModelFilter to DataMapper
- Reworked Publisher URLs to be easier to customise, and more consistent
- Added test module
- ModelDataMapper now creates a new Model instance if not passed one at construction.
- Pass list of excluded fields to Model.full_clean()

8.2.22 v0.14.5.1 (2015-08-06)

Bug Fixes:

- Use six.string_types not str in flatten_errors
- Properly update error dict in ModelDataMapper._clean

8.2.23 v0.14.5 (2015-08-06)

Enhancements:

- Add _clean method to DataMapper for whole-object cleaning.
- Make ModelDataMapper._clean call instance.full_clean.

Bug Fixes:

- Fix ModelDataMapper to not get confused by `six.with_metaclass` hacks.
- Fix ListMixin.ok_response to call self.multiple_response not self.list_response

8.2.24 v0.14.4 (2015-05-19)

Enhancements:

- Fix travis config
- Simplify AppConfig usage
- Switched from using Django's HTTP reason phrases to Python's.
- Tidied the abstractions of response helpers in django.rest.views.
- Added BaseListView and BaseObjectView to django.rest.views.

Bug Fixes:

- Use our own get_object_or_404 shortcut in ModelPublisher.
- Fixed rest.views to closer match RFC [Thanks Ian Wilson]

8.2.25 v0.14.3 (2015-02-17)

Ehancements

- JsonMixin.get_request_data will now handle form encoded data for PUT
- Change API for datamapper to separate _apply and _patch.

8.2.26 v0.14.2 (2015-01-23)

WARNING: Removed module

The module nap.exceptions has been completely removed.

Enhancements:

- Switched custom ValidationError / ValidationErrors to django's ValidationError
- Added DataMapper library
- Added CBV mixins for composing API Views that use DataMappers

8.2.27 v0.14.1.1

Bug Fixes:

- Add required *name* attribute to AppConfig [thanks bobobo1618]

8.2.28 v0.14.1

Enhancements:

- Import REASON_CODES from Django
- Use Django's JsonResponse if available, or our own copy of it.
- Unify all json handling functions into utils.JsonMixin

- Add RPCView introspection
- Use Django's vendored copy of 'six'
- Add new runtests script

Bug Fixes:

- Cope with blank content encoding values in RPC Views
- Raise a 404 on invalid page_size value
- Validate the data we got in RPC View is passable as **kwargs
- ISO_8859_1 isn't defined in older Django versions
- Emulate django template lookups in digattr by ignoring callables flagged 'do_not_call_in_templates'

8.2.29 v0.14.0

WARNING: API breakage

A large reorganisation of the code was undertaken.

Now there are 3 major top-level modules: - serialiser - rest - rpc

---

Enhancements:

- Added functional RPC system [merged from django-marionette]
- Made most things accessible in top-level module

8.2.30 v0.13.9

Enhancements:

- Added Django 1.7 AppConfig, which will auto-discover on ready
- Added a default implementation of ModelPublsiher.list_post_default
- Tidied code with flake8

Bug Fixes:

- Fixed use of wrong argument in auth.permit_groups

8.2.31 v0.13.8

Enhancements:

- Added prefetch_related and select_related support to ExportCsv action
- Added Field.virtual to smooth changes to Field now raising AttributeError, and support optional fields

8.2.32 v0.13.7

Enhancements:

- Added ReadTheDocs, and prettied up the docs
- Use Pythons content-type parsing
- Added RPC publisher [WIP]
- Allow api.register to be used as a decorator
- Make Meta classes more proscriptive
- Allow ModelSerialiser to override Field type used for fields.
- Added ModelReadSerialiser and ModelCreateUpdateSerialiser to support more complex inflate scenarios [WIP]

Bug Fixes:

- Fixed ExportCsv and simplecsv extras
- Raise AttributeError if a deflating a field with no default set would result in using its default. [Fixes #28]
- Fixed auto-generated api_names.
- Purged under-developed ModelFormMixin class

8.2.33 v0.13.6

Enhancements:

- Overhauled testing
- Added 'total_pages' to page meta.
- Added Serialiser.obj_class

8.2.34 v0.13.5.1

Bug Fixes:

- Fix fix for b" from last release, to work in py2

8.2.35 v0.13.5

Bug Fixes:

- Fix use of b" for Py3.3 [thanks zzing]

Enhancements:

- Add options to control patterns

8.2.36 v0.13.4

Bug Fixes:

- Return http.NotFound instead of raising it

Enhancements:

- Added views publisher
- Updated docs
- Re-added support for ujson, if installed
- Tidied up with pyflakes/pylint
- Added Publisher.response_class property

8.2.37 v0.13.3

Bugs Fixed:

- Make API return NotFound, instead of Raising it
- Remove bogus CSV Reader class

8.2.38 v0.13.2.1

Bugs Fixed:

- Fixed typo
- Fixed resolving cache in mixin

8.2.39 v0.13.2

Enhancements:

- Separate Publisher.build_view from Publisher.patterns to ease providing custom patterns
- Added SimplePatternsMixin for Publisher
- Added Publisher.sort_object_list and Publisher.filter_object_list hooks

8.2.40 v0.13.1

Bugs Fixed:

- Fixed silly bug in inflate

8.2.41 v0.13.0

WARNING: API breakage

Changed auto-discover to look for 'publishers' instead of 'seraliser'.

Enhancements:

- Added Field.null support
- Now use the Field.default value
- ValidationError handled in all field and custom inflator methods

8.2.42 v0.12.5.1

Bugs Fixed:

- Fix mistake introduced in 0.12.3 which broke NewRelic support

8.2.43 v0.12.5

Bugs Fixed:

- Restored Django 1.4 compatibility

Enhancements:

- Allow disabling of API introspection index

8.2.44 v0.12.4

Bugs Fixed:

- Fixed filename generation in csv export action
- Fixed unicode/str issues with type() calls

Enhancements:

- Split simplecsv and csv export into extras module
- Merged engine class directly into Publisher
- Added fields.StringField

8.2.45 v0.12.3

Bugs Fixed:

- Fix argument handling in Model*SerialiserFields
- Tidied up with pyflakes

Enhancements:

- Added support for Py3.3 [thanks ioneyed]
- Overhauled the MetaSerialiser class
- Overhauled "sandbox" app
- Added csv export action

8.2.46 v0.12.2

Enhancements:

- Support read_only in modelserialiser_factory

8.2.47 v0.12.1

Bugs Fixed:

- Flatten url patterns so object_default can match without trailing /
- Fix class returned in permit decorator [Thanks emilkjer]

Enhancements:

- Allow passing an alternative default instead of None for Publisher.get_request_data
- Added "read_only_fields" to ModelSerialiser [thanks jayant]

8.2.48 v0.12

Enhancements:

- Tune Serialisers to pre-build their deflater/inflater method lists, removing work from the inner loop
- Remove *args where it's no helpful

8.2.49 v0.11.6.1

Bugs Fixed:

- Renamed HttpResponseRedirect to HttpResponseRedirection to avoid clashing with Django http class

8.2.50 v0.11.6

Bugs Fixed:

- Raise a 404 on paginator raising EmptyPage, instead of failing

8.2.51 v0.11.5.1

Bugs Fixed:

- Fix arguments passed to execute method

8.2.52 v0.11.5

Enhancements:

- Add Publisher.execute to make wrapping handler calls easier [also, makes NewRelic simpler to hook in]
- Allow empty first pages in pagination
- Added support module for NewRelic

8.2.53 v0.11.4

Enhancements:

- Make content-type detection more forgiving

8.2.54 v0.11.3

Enhancements:

- Make get_page honor limit parameter, but bound it to max_page_size, which defaults to page_size
- Allow changing the GET param names for page, offset and limit
- Allow passing page+limit or offset+limit

8.2.55 v0.11.2

Enhancements:

- Added BooleanField
- Extended tests
- Force CSRF protection

8.2.56 v0.11.1

Enhancements:

- Changed SerialiserField/ManySerialiserField to replace reduce/restore instead of overriding inflate/deflate methods
- Fixed broken url pattern for object action
- Updated fields documentation

8.2.57 v0.11

API breakage

Serialiser.deflate_object and Serialiser.deflate_list have been renamed.

Enhancements:

- Changed deflate_object and deflate_list to object_deflate and list_deflate to avoid potential field deflater name conflict
- Moved all model related code to models.py
- Added modelserialiser_factory
- Updated ModelSerialiserField/ModelManySerialiserField to optionally auto-create a serialiser for the supplied model

8.2.58 v0.10.3

Enhancements:

- Added python2.6 support back [thanks nkuttler]
- Added more documentation

- Added Publisher.get_serialiser_kwargs hook
- Publisher.get_data was renamed to Publisher.get_request_data for clarity

8.2.59 v0.10.2

Bugs Fixed:

- Removed leftover debug print

8.2.60 v0.10.1

Enhancements:

- Added Publisher introspection
- Added LocationHeaderMixin to HTTP classes

8.2.61 v0.10

Bugs Fixed:

- Removed useless cruft form utils

Enhancements:

- Replaced http subclasses with Exceptional ones
- Wrap call to handlers to catch Exceptional http responses

8.2.62 v0.9.1

Enhancements:

- Started documentation
- Added permit_groups decorator
- Minor speedup in MetaSerialiser

8.2.63 v0.9

Bugs Fixed:

- Fixed var name bug in ModelSerialiser.restore_object
- Removed old 'may' auth API

Enhancements:

- Added permit decorators
- use string formatting not join - it's slightly faster

8.2.64 v0.8

Enhancements:

- Added create/delete methods to ModelPublisher
- Renamed HttpResponse subclasses
- Split out BasePublisher class
- Added http.STATUS dict/list utility class

Note: Because this uses OrderedDict nap is no longer python2.6 compatible

---

8.2.65 v0.7.1

Enhancements:

- Use first engine.CONTENT_TYPES as default content type for responses

8.2.66 v0.7

Bugs Fixed:

- Removed custom JSON class

Enhancements:

- Added Engine mixin classes
- Added MsgPack support
- Added type-casting fields

8.2.67 v0.6

Bugs Fixed:

- Fixed JSON serialising of date/datetime objects

Enhancements:

- Added index view to API
- Make render_single_object use create_response
- Allow create_response to use a supplied response class

8.2.68 v0.5

Enhancements:

- Added names to URL patterns
- Added “argument” URL patterns

8.2.69 v0.4

Enhancements:

- Added next/prev flags to list meta-data
- Added tests

8.2.70 v0.3

Enhancements:

- Changed to more generic extra arguments in Serialiser

8.2.71 v0.2

Bugs Fixed:

- Fixed bug in serialiser meta-class that broke inheritance
- Fixed variable names

Enhancements:

- Pass the Publisher down into the Serialiser for more flexibility
- Allow object IDs to be slugs
- Handle case of empty request body with JSON content type
- Added SerialiserField and ManySerialiserField
- Added Api machinery
- Changed Serialiser to use internal Meta class
- Added ModelSerialiser class

8.2.72 v0.1

Enhancements:

- Initial release, fraught with bugs :)

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

C

D

E

F

G

L

M

O

P

R

S