Using Django Rest Framework

To Simplify Your App

Ed Henderson

March 10, 2015

Core Features

Everything you need to build a REST API

- Serializers
- Renderers and Parsers
- Views, Viewsets
- Routers
- Authentication and Permissions
- Throttling, Filtering, Pagination, Versioning

Serializers

- Convert your data into Python native datatypes.
- Used to convert a DB Model (or Models) into native types
- Works with Foreign Keys, and arrays of objects.
- Deserialize data also.

Example: Serialize an Object

- Basic Python Object, not a model

```python
class SomeObject(object):

    def __init__(self, charfield, floatfield, intfield, emailfield):
        self.charfield = charfield
        self.floatfield = floatfield
        self.intfield = intfield
        self.emailfield = emailfield

    def __str__(self):
        return "SomeObject:{} - {} - {} -{}".format(
            self.charfield, self.floatfield, self.intfield, self.emailfield
        )

```

Example: Simple Object Serializer

- Same format as Model and Form definitions
- Must provide explicit create and update methods

```python

class ObjectSerializer(serializers.Serializer):
    charfield = serializers.CharField(max_length=100)
    floatfield = serializers.FloatField()
    intfield = serializers.IntegerField()
    emailfield = serializers.EmailField()

    def create(self, validated_data):
        return SomeObject(**validated_data)

    def update(self, instance, validated_data):
        """
        Check each param, and update as needed on the instance..
        """
        # Update the instance
        return instance

```

Example: Simple Object Serializer - save

- create function used during save
- update function used during update

```
# Save
        serializer = ObjectSerializer(data=data)
        serializer()
        obj = serializer.save()

# Update
        serializer = ObjectSerializer(instance, data=data)
        serializer()
        obj = serializer.save()
```

Serializer Fields and Validation

- Standard fields have basic validation
- Field types define basic validation
- Define read or write only fields
- Allow null fields, provide defaults, specify a field validator
- def validate() for object level validation
- Can create custom fields also

Serializers can be nested

- Serializer class is a 'Field' type
- Handle complex hierarchies of objects
- Nested serializer could be a list of items
- create() and update() methods more complex

```
class CommentSerializer(serializers.Serializer):
    user = UserSerializer(required=False)
    edits = EditItemSerializer(many=True)  # A nested list of 'edit' items.
    content = serializers.CharField(max_length=200)
    created = serializers.DateTimeField()
```

Model Serializers

- Uses introspection to determine fields
- Creates model validators, i.e. "unique together"
- Creates default save and update methods
- Default behavior handles most situations

```python
class SomeModelSerializer(serializers.ModelSerializer):
    class Meta:
        model = SomeModel
        fields = (<list of fields>) # defaults to all fields
```

Model Serializers: Extending

- Add additional fields to a serializer
- Fields can be based on a value, property or function
- Specify a different field type than the default

```
class SomeModelSerializer(serializers.ModelSerializer):
    extra_field = serializers.CharField(source='get_extra_data', read_only=True)

    class Meta:
        model = SomeModel
        fields = (<list of fields>) # defaults to all fields
```

Model Serializers: Relations

- Handling of Foreign Key, OneToOne, ManyToMany Fields
- Default is to list the ID as an integer
- Other choices
    - StringRelatedField

```
tracks = serializers.StringRelatedField(many=True)
tracks = serializers.PrimaryKeyRelatedField(many=True, read_only=True)
tracks = serializers.HyperlinkedRelatedField(...)
tracks = serializers.SlugRelatedField(...)
tracks = TrackSerializer(many=True, read_only=True)
```

Model Serializers: Relations

- Handling of Foreign Key, OneToOne, ManyToMany Fields
- Default is to list the ID as an integer
- Other choices
  - StringRelatedField
  - PrimaryKeyRelatedField (default)

```
tracks = serializers.StringRelatedField(many=True)
tracks = serializers.PrimaryKeyRelatedField(many=True, read_only=True)
tracks = serializers.HyperlinkedRelatedField(...)
tracks = serializers.SlugRelatedField(...)
tracks = TrackSerializer(many=True, read_only=True)
```

Model Serializers: Relations

- Handling of Foreign Key, OneToOne, ManyToMany Fields
- Default is to list the ID as an integer
- Other choices
    - StringRelatedField
    - PrimaryKeyRelatedField (default)
    - HyperlinkedRelatedField

```
tracks = serializers.StringRelatedField(many=True)
tracks = serializers.PrimaryKeyRelatedField(many=True, read_only=True)
tracks = serializers.HyperlinkedRelatedField(...)
tracks = serializers.SlugRelatedField(...)
tracks = TrackSerializer(many=True, read_only=True)
```

Model Serializers: Relations

- Handling of Foreign Key, OneToOne, ManyToMany Fields
- Default is to list the ID as an integer
- Other choices
  - StringRelatedField
  - PrimaryKeyRelatedField (default)
  - HyperlinkedRelatedField
  - SlugRelatedField

```
tracks = serializers.StringRelatedField(many=True)
tracks = serializers.PrimaryKeyRelatedField(many=True, read_only=True)
tracks = serializers.HyperlinkedRelatedField(...)
tracks = serializers.SlugRelatedField(...)
tracks = TrackSerializer(many=True, read_only=True)
```

Model Serializers: Relations

- Handling of Foreign Key, OneToOne, ManyToMany Fields
- Default is to list the ID as an integer
- Other choices
  - StringRelatedField
  - PrimaryKeyRelatedField (default)
  - HyperlinkedRelatedField
  - SlugRelatedField
  - HyperlinkedIdentityField

```
tracks = serializers.StringRelatedField(many=True)
tracks = serializers.PrimaryKeyRelatedField(many=True, read_only=True)
tracks = serializers.HyperlinkedRelatedField(...)
tracks = serializers.SlugRelatedField(...)
tracks = TrackSerializer(many=True, read_only=True)
```

Model Serializers: Relations

- Handling of Foreign Key, OneToOne, ManyToMany Fields
- Default is to list the ID as an integer
- Other choices
    - StringRelatedField
    - PrimaryKeyRelatedField (default)
    - HyperlinkedRelatedField
    - SlugRelatedField
    - HyperlinkedIdentityField
    - Nested Relationship

```
tracks = serializers.StringRelatedField(many=True)
tracks = serializers.PrimaryKeyRelatedField(many=True, read_only=True)
tracks = serializers.HyperlinkedRelatedField(...)
tracks = serializers.SlugRelatedField(...)
tracks = TrackSerializer(many=True, read_only=True)
```

Serializers: What did I skip?

- Take control of field mapping
- Specify some field type defaults: url, or choice fields
- Customization

Renderers and Parsers

- Serializers convert your object/model to/from Python native datatypes.
  - Lists
  - Dictionaries
  - String, Float, Int
- Renderers/Parsers convert to/from
  - JSON
  - Template HTML
  - Static HTML
  - Browsable API

Which renderer to use?

- Accept header
  - text/html
  - application/json
- .format headers
  - .json
  - .xml
- query parameters
  - format=json
  - format=xml

Generic Views

- Just covering Class based views
- Views assembled from Mixins and a GenericAPIView
- GenericAPIView based on APIView
    - Authentication, Throttling
    - Content negotiation
    - get_queryset, get_object
    - Serialization, filtering
    - Pagination

Mixins

- Used when assembling a working generic
- Can be used in your own views also
- Add the core functions
    - create (CreateModelMixin)
    - list (ListModelMixin)
    - retrieve (RetrieveModelMixin)
    - update, partial_update (UpdateModelMixin)
    - destroy (DestroyModelMixin)

REST method mapping

-
- get
  - list
  - retrieve
- post
  - create
- put
  - update
- patch
  - partial_update
- delete
  - destroy

Mapping Views

- Too many of these view types to list
- take a look in generics file
- Contains most of the combinations you will need
- Or, roll your own

```python
class ListCreateAPIView(mixins.ListModelMixin,
                        mixins.CreateModelMixin,
                        GenericAPIView):
    """
    Concrete view for listing a queryset or creating a model instance.
    """
    def get(self, request, *args, **kwargs):
        return self.list(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        return self.create(request, *args, **kwargs)
```

ViewSets

- A Class based view with action methods
- create, retrieve, update, destroy, ...
- Manually mapped to a REST operation
- Or, dynamically mapped using routers.
- ModelViewSet provide the kitchen sink

```
class ModelViewSet(mixins.CreateModelMixin,
                   mixins.RetrieveModelMixin,
                   mixins.UpdateModelMixin,
                   mixins.DestroyModelMixin,
                   mixins.ListModelMixin,
                   GenericViewSet):
    """
    A viewset that provides default `create()`, `retrieve()`, `update()`,
    `partial_update()`, `destroy()` and `list()` actions.
    """
    pass
```

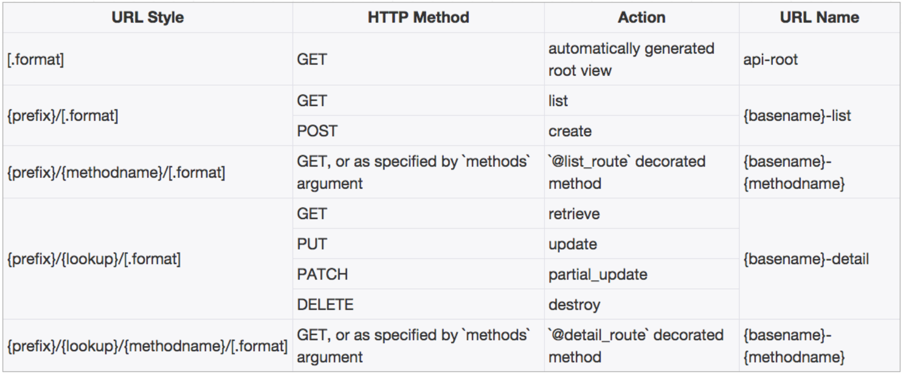
URL Mapping

Now we have some shiny new views, but need to map them to a set of URLs

- list: 'ŵidget$'
- retrieve: 'ŵidget{pk}'
- update: 'ŵidget{pk}'
- destroy: 'ŵidget{pk}'
- We need different operations for some of these URLs....

Routers

- Use introspection to determine what action methods exist
- Builds the URLs for you, and maps the REST methods
- Define custom methods with list_route and detail_route



URL Style	HTTP Method	Action	URL Name
[.format]	GET	automatically generated root view	api-root
{prefix}/[.format]	GET	list	{basename}-list
	POST	create	
{prefix}/{methodname}/[.format]	GET, or as specified by \`methods\` argument	\`@list_route\` decorated method	{basename}-{methodname}
{prefix}/{lookup}/[.format]	GET	retrieve	{basename}-detail
	PUT	update	
	PATCH	partial_update	
	DELETE	destroy	
{prefix}/{lookup}/{methodname}/[.format]	GET, or as specified by \`methods\` argument	\`@detail_route\` decorated method	{basename}-{methodname}

Authentication

- Easily add Authentication requirements to any API point
  - BasicAuth - use only in testing
  - SessionAuth - the default Django auth
  - TokenAuth - client server setups
  - CustomAuth - OAuth, etc.
- Set authentication requirements locally or globally

```
    # define local to view
    authentication_classes = (SessionAuthentication, BasicAuthentication)

    # Or globally
    REST_FRAMEWORK = {
        'DEFAULT_AUTHENTICATION_CLASSES': (
            'rest_framework.authentication.SessionAuthentication',
        )
    }

    # Basic permissions
    permission_classes = (IsAuthenticated,)

```

Permissions

- Limit access to objects, methods
  - IsAuthenticated
  - IsAdminUser
  - IsAuthenticatedOrReadOnly
  - Django model permissions

```
    # Basic permissions
    permission_classes = (IsAuthenticated,)

```

More Stuff...

Too much to cover it all..

- Throttling
- Filtering - override .get_queryset
- Pagination
- Versioning

```
    'DEFAULT_THROTTLE_RATES': {
        'anon': '100/day',
        'user': '1000/day'
    }

def get_serializer_class(self):
    if self.request.version == 'v1':
        return AccountSerializerVersion1
    return AccountSerializer

```

Simplify your Application

- Assume you have some models you want to manipulate
- You are writing a..
    - Client side app in AngularJS, ExtJS, EmberJS
    - Backend for a desktop app
    - Backend for a mobile app
- Django Rest Framework can build your API fast
- But, let you customize it to your hearts content

Plant Database - Models

- Database of plant taxonomy
- Lots of fields..

```python
class PlantUSDA(models.Model):
    accepted_symbol = models.CharField(max_length=30, blank=True, null=True)
    ....
    genus = models.CharField(max_length=30, blank=True, null=True)
    family = models.CharField(max_length=30, blank=True, null=True)
    family_symbol = models.CharField(max_length=30, blank=True, null=True)
    family_common_name = models.CharField(max_length=64, blank=True, null=True)
    order = models.CharField(max_length=30, blank=True, null=True)
    ...

```

Plant Database - Serializer

- Limit the # of fields that are returned

```
class PlantSerializer(serializers.ModelSerializer):

    class Meta:
        model=PlantUSDA
        fields = (
            'accepted_symbol', 'synonym_symbol', 'common_name',
            'genus', 'family', 'order', 'subclass',
            'classname', 'cultivar_name'
        )

```

Plant Database - View and URLs

- This part is too easy

```
#views.py
...
class PlantViewSet(ModelViewSet):
    queryset = PlantUSDA.objects.all()[1:300]
    serializer_class = PlantSerializer

#urls.py
from rest_framework.routers import DefaultRouter

from .views import PlantViewSet

router = DefaultRouter()
router.register(r'router', PlantViewSet)

urlpatterns = patterns("",
    url(r'^', include(router.urls)),
)

```

Summarize

- Serializers, renderers and parsers

Summarize

- Serializers, renderers and parsers
- Views and view sets

Summarize

- Serializers, renderers and parsers
- Views and view sets
- routers and the url

Summarize

- Serializers, renderers and parsers
- Views and view sets
- routers and the url
- auth, perms, throttles, filtering, pagination, versioning

Summarize

- Serializers, renderers and parsers
- Views and view sets
- routers and the url
- auth, perms, throttles, filtering, pagination, versioning
- Questions