serializers.py

Serializers

Expanding the usefulness of the serializers is something that we would like to address. However, it's not a trivial problem, and it will take some serious design work.

> — *Russell Keith-Magee, Django users group*

Serializers allow complex data such as querysets and model instances to be converted to native Python datatypes that can then be easily rendered into `JSON`, `XML` or other content types. Serializers also provide deserialization, allowing parsed data to be converted back into complex types, after first validating the incoming data.

The serializers in REST framework work very similarly to Django's `Form` and `ModelForm` classes. We provide a `Serializer` class which gives you a powerful, generic way to control the output of your responses, as well as a `ModelSerializer` class which provides a useful shortcut for creating serializers that deal with model instances and querysets.

Declaring Serializers

Let's start by creating a simple object we can use for example purposes:

```
from datetime import datetime

class Comment(object):
    def __init__(self, email, content, created=None):
        self.email = email
        self.content = content
        self.created = created or datetime.now()

comment = Comment(email='leila@example.com', content='foo bar')
```

We'll declare a serializer that we can use to serialize and deserialize data that corresponds to `Comment` objects.

Declaring a serializer looks very similar to declaring a form:

```
from rest_framework import serializers

class CommentSerializer(serializers.Serializer):
    email = serializers.EmailField()
    content = serializers.CharField(max_length=200)
    created = serializers.DateTimeField()
```

Serializing objects

We can now use `CommentSerializer` to serialize a comment, or list of comments. Again, using the `Serializer` class looks a lot like using a `Form` class.

```
serializer = CommentSerializer(comment)
serializer.data
# {'email': 'leila@example.com', 'content': 'foo bar', 'created': '2016-01-27T15:17:10.3758
```

At this point we've translated the model instance into Python native datatypes. To finalise the serialization process we render the data into `json`.

```
from rest_framework.renderers import JSONRenderer

json = JSONRenderer().render(serializer.data)
json
# b'{"email":"leila@example.com","content":"foo bar","created":"2016-01-27T15:17:10.375877"
```

Deserializing objects

Deserialization is similar. First we parse a stream into Python native datatypes...

```
from django.utils.six import BytesIO
from rest_framework.parsers import JSONParser

stream = BytesIO(json)
data = JSONParser().parse(stream)
```

...then we restore those native datatypes into a dictionary of validated data.

```
serializer = CommentSerializer(data=data)
serializer.is_valid()
# True
serializer.validated_data
# {'content': 'foo bar', 'email': 'leila@example.com', 'created': datetime.datetime(2012, 0
```

Saving instances

If we want to be able to return complete object instances based on the validated data we need to implement one or both of the `.create()` and `update()` methods. For example:

```
class CommentSerializer(serializers.Serializer):
    email = serializers.EmailField()
    content = serializers.CharField(max_length=200)
    created = serializers.DateTimeField()

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

    def update(self, instance, validated_data):
        instance.email = validated_data.get('email', instance.email)
        instance.content = validated_data.get('content', instance.content)
        instance.created = validated_data.get('created', instance.created)
        return instance
```

```
        return instance
```

If your object instances correspond to Django models you'll also want to ensure that these methods save the object to the database. For example, if `Comment` was a Django model, the methods might look like this:

```
    def create(self, validated_data):
        return Comment.objects.create(**validated_data)

    def update(self, instance, validated_data):
        instance.email = validated_data.get('email', instance.email)
        instance.content = validated_data.get('content', instance.content)
        instance.created = validated_data.get('created', instance.created)
        instance.save()
        return instance
```

Now when deserializing data, we can call `.save()` to return an object instance, based on the validated data.

```
comment = serializer.save()
```

Calling `.save()` will either create a new instance, or update an existing instance, depending on if an existing instance was passed when instantiating the serializer class:

```
# .save() will create a new instance.
serializer = CommentSerializer(data=data)

# .save() will update the existing `comment` instance.
serializer = CommentSerializer(comment, data=data)
```

Both the `.create()` and `.update()` methods are optional. You can implement either neither, one, or both of them, depending on the use-case for your serializer class.

Passing additional attributes to `.save()`

Sometimes you'll want your view code to be able to inject additional data at the point of saving the instance. This additional data might include information like the current user, the current time, or anything else that is not part of the request data.

You can do so by including additional keyword arguments when calling `.save()`. For example:

```
serializer.save(owner=request.user)
```

Any additional keyword arguments will be included in the `validated_data` argument when `.create()` or `.update()` are called.

Overriding `.save()` directly.

In some cases the `.create()` and `.update()` method names may not be meaningful. For example, in a contact form we may not be creating new instances, but instead sending an email or other message.

In these cases you might instead choose to override `.save()` directly, as being more readable and meaningful.

For example:

```
class ContactForm(serializers.Serializer):
    email = serializers.EmailField()
    message = serializers.CharField()

    def save(self):
        email = self.validated_data['email']
        message = self.validated_data['message']
        send_email(from=email, message=message)
```

Note that in the case above we're now having to access the serializer `.validated_data` property directly.

Validation

When deserializing data, you always need to call `is_valid()` before attempting to access the validated data, or save an object instance. If any validation errors occur, the `.errors` property will contain a dictionary representing the resulting error messages. For example:

```
serializer = CommentSerializer(data={'email': 'foobar', 'content': 'baz'})
serializer.is_valid()
# False
serializer.errors
# {'email': [u'Enter a valid e-mail address.'], 'created': [u'This field is required.']}
```

Each key in the dictionary will be the field name, and the values will be lists of strings of any error messages corresponding to that field. The `non_field_errors` key may also be present, and will list any general validation errors. The name of the `non_field_errors` key may be customized using the `NON_FIELD_ERRORS_KEY` REST framework setting.

When deserializing a list of items, errors will be returned as a list of dictionaries representing each of the deserialized items.

Raising an exception on invalid data

The `.is_valid()` method takes an optional `raise_exception` flag that will cause it to raise a `serializers.ValidationError` exception if there are validation errors.

These exceptions are automatically dealt with by the default exception handler that REST framework provides, and will return `HTTP 400 Bad Request` responses by default.

```
# Return a 400 response if the data was invalid.
serializer.is_valid(raise_exception=True)
```

Field-level validation

You can specify custom field-level validation by adding `.validate_<field_name>` methods to your `Serializer` subclass. These are similar to the `.clean_<field_name>` methods on Django forms.

These methods take a single argument, which is the field value that requires validation.

Your `validate_<field_name>` methods should return the validated value or raise a `serializers.ValidationError`. For example:

```
from rest_framework import serializers

class BlogPostSerializer(serializers.Serializer):
    title = serializers.CharField(max_length=100)
    content = serializers.CharField()

    def validate_title(self, value):
        """
        Check that the blog post is about Django.
        """
        if 'django' not in value.lower():
            raise serializers.ValidationError("Blog post is not about Django")
        return value
```

Note: If your `<field_name>` is declared on your serializer with the parameter `required=False` then this validation step will not take place if the field is not included.

Object-level validation

To do any other validation that requires access to multiple fields, add a method called `.validate()` to your `Serializer` subclass. This method takes a single argument, which is a dictionary of field values. It should raise a `ValidationError` if necessary, or just return the validated values. For example:

```
from rest_framework import serializers

class EventSerializer(serializers.Serializer):
    description = serializers.CharField(max_length=100)
    start = serializers.DateTimeField()
    finish = serializers.DateTimeField()

    def validate(self, data):
        """
        Check that the start is before the stop.
        """
        if data['start'] > data['finish']:
            raise serializers.ValidationError("finish must occur after start")
        return data
```

Validators

Individual fields on a serializer can include validators, by declaring them on the field instance, for example:

```
def multiple_of_ten(value):
    if value % 10 != 0:
        raise serializers.ValidationError('Not a multiple of ten')

class GameRecord(serializers.Serializer):
    score = IntegerField(validators=[multiple_of_ten])
    ...
```

Serializer classes can also include reusable validators that are applied to the complete set of field data. These validators are included by declaring them on an inner `Meta` class, like so:

```
class EventSerializer(serializers.Serializer):
    name = serializers.CharField()
    room_number = serializers.IntegerField(choices=[101, 102, 103, 201])
    date = serializers.DateField()

    class Meta:
        # Each room only has one event per day.
        validators = UniqueTogetherValidator(
            queryset=Event.objects.all(),
            fields=['room_number', 'date']
        )
```

For more information see the validators documentation.

Accessing the initial data and instance

When passing an initial object or queryset to a serializer instance, the object will be made available as `.instance`. If no initial object is passed then the `.instance` attribute will be `None`.

When passing data to a serializer instance, the unmodified data will be made available as `.initial_data`. If the data keyword argument is not passed then the `.initial_data` attribute will not exist.

Partial updates

By default, serializers must be passed values for all required fields or they will raise validation errors. You can use the `partial` argument in order to allow partial updates.

```
# Update `comment` with partial data
serializer = CommentSerializer(comment, data={'content': u'foo bar'}, partial=True)
```

Dealing with nested objects

The previous examples are fine for dealing with objects that only have simple datatypes, but sometimes we also need to be able to represent more complex objects, where some of the attributes of an object might not be simple datatypes such as strings, dates or integers.

The `Serializer` class is itself a type of `Field`, and can be used to represent relationships where one object type is nested inside another.

```
class UserSerializer(serializers.Serializer):
    email = serializers.EmailField()
    username = serializers.CharField(max_length=100)

class CommentSerializer(serializers.Serializer):
    user = UserSerializer()
    content = serializers.CharField(max_length=200)
    created = serializers.DateTimeField()
```

If a nested representation may optionally accept the `None` value you should pass the `required=False` flag to the nested serializer.

```
class CommentSerializer(serializers.Serializer):
    user = UserSerializer(required=False)  # May be an anonymous user.
    content = serializers.CharField(max_length=200)
    created = serializers.DateTimeField()
```

Similarly if a nested representation should be a list of items, you should pass the `many=True` flag to the nested serialized.

```
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()
```

Writable nested representations

When dealing with nested representations that support deserializing the data, any errors with nested objects will be nested under the field name of the nested object.

```
serializer = CommentSerializer(data={'user': {'email': 'foobar', 'username': 'doe'}, 'conte
serializer.is_valid()
# False
serializer.errors
# {'user': {'email': [u'Enter a valid e-mail address.']}, 'created': [u'This field is requi
```

Similarly, the `.validated_data` property will include nested data structures.

Writing `.create()` methods for nested representations

If you're supporting writable nested representations you'll need to write `.create()` or `.update()` methods that handle saving multiple objects.

The following example demonstrates how you might handle creating a user with a nested profile object.

```
class UserSerializer(serializers.ModelSerializer):
    profile = ProfileSerializer()

    class Meta:
        model = User
        fields = ('username', 'email', 'profile')

    def create(self, validated_data):
        profile_data = validated_data.pop('profile')
        user = User.objects.create(**validated_data)
        Profile.objects.create(user=user, **profile_data)
        return user
```

Writing `.update()` methods for nested representations

For updates you'll want to think carefully about how to handle updates to relationships. For example if the data for the relationship is `None`, or not provided, which of the following should occur?

- Set the relationship to `NULL` in the database.
- Delete the associated instance.
- Ignore the data and leave the instance as it is.
- Raise a validation error.

Here's an example for an `update()` method on our previous `UserSerializer` class.

```
    def update(self, instance, validated_data):
        profile_data = validated_data.pop('profile')
        # Unless the application properly enforces that this field is
        # always set, the follow could raise a `DoesNotExist`, which
        # would need to be handled.
        profile = instance.profile

        instance.username = validated_data.get('username', instance.username)
        instance.email = validated_data.get('email', instance.email)
        instance.save()

        profile.is_premium_member = profile_data.get(
            'is_premium_member',
            profile.is_premium_member
        )
        profile.has_support_contract = profile_data.get(
            'has_support_contract',
            profile.has_support_contract
         )
        profile.save()
```

```
        return instance
```

Because the behavior of nested creates and updates can be ambiguous, and may require complex dependencies between related models, REST framework 3 requires you to always write these methods explicitly. The default `ModelSerializer` `.create()` and `.update()` methods do not include support for writable nested representations.

It is possible that a third party package, providing automatic support some kinds of automatic writable nested representations may be released alongside the 3.1 release.

Handling saving related instances in model manager classes

An alternative to saving multiple related instances in the serializer is to write custom model manager classes that handle creating the correct instances.

For example, suppose we wanted to ensure that `User` instances and `Profile` instances are always created together as a pair. We might write a custom manager class that looks something like this:

```
class UserManager(models.Manager):
    ...

    def create(self, username, email, is_premium_member=False, has_support_contract=False):
        user = User(username=username, email=email)
        user.save()
        profile = Profile(
            user=user,
            is_premium_member=is_premium_member,
            has_support_contract=has_support_contract
        )
        profile.save()
        return user
```

This manager class now more nicely encapsulates that user instances and profile instances are always created at the same time. Our `.create()` method on the serializer class can now be re-written to use the new manager method.

```
def create(self, validated_data):
    return User.objects.create(
        username=validated_data['username'],
        email=validated_data['email']
        is_premium_member=validated_data['profile']['is_premium_member']
        has_support_contract=validated_data['profile']['has_support_contract']
    )
```

For more details on this approach see the Django documentation on model managers, and this blogpost on using model and manager classes.

Dealing with multiple objects

The `Serializer` class can also handle serializing or deserializing lists of objects.

Serializing multiple objects

To serialize a queryset or list of objects instead of a single object instance, you should pass the `many=True` flag when instantiating the serializer. You can then pass a queryset or list of objects to be serialized.

```
queryset = Book.objects.all()
serializer = BookSerializer(queryset, many=True)
serializer.data
# [
#     {'id': 0, 'title': 'The electric kool-aid acid test', 'author': 'Tom Wolfe'},
#     {'id': 1, 'title': 'If this is a man', 'author': 'Primo Levi'},
#     {'id': 2, 'title': 'The wind-up bird chronicle', 'author': 'Haruki Murakami'}
# ]
```

Deserializing multiple objects

The default behavior for deserializing multiple objects is to support multiple object creation, but not support multiple object updates. For more information on how to support or customize either of these cases, see the ListSerializer documentation below.

Including extra context

There are some cases where you need to provide extra context to the serializer in addition to the object being serialized. One common case is if you're using a serializer that includes hyperlinked relations, which requires the serializer to have access to the current request so that it can properly generate fully qualified URLs.

You can provide arbitrary additional context by passing a `context` argument when instantiating the serializer. For example:

```
serializer = AccountSerializer(account, context={'request': request})
serializer.data
# {'id': 6, 'owner': u'denvercoder9', 'created': datetime.datetime(2013, 2, 12, 09, 44, 56,
```

The context dictionary can be used within any serializer field logic, such as a custom `.to_representation()` method, by accessing the `self.context` attribute.

ModelSerializer

Often you'll want serializer classes that map closely to Django model definitions.

The `ModelSerializer` class provides a shortcut that lets you automatically create a `Serializer` class with fields that correspond to the Model fields.

The `ModelSerializer` class is the same as a regular `Serializer` class, except that:

- It will automatically generate a set of fields for you, based on the model.
- It will automatically generate validators for the serializer, such as unique_together validators.
- It includes simple default implementations of `.create()` and `.update()`.

Declaring a `ModelSerializer` looks like this:

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        fields = ('id', 'account_name', 'users', 'created')
```

By default, all the model fields on the class will be mapped to a corresponding serializer fields.

Any relationships such as foreign keys on the model will be mapped to `PrimaryKeyRelatedField`. Reverse relationships are not included by default unless explicitly included as specified in the serializer relations documentation.

Inspecting a `ModelSerializer`

Serializer classes generate helpful verbose representation strings, that allow you to fully inspect the state of their fields. This is particularly useful when working with `ModelSerializers` where you want to determine what set of fields and validators are being automatically created for you.

To do so, open the Django shell, using `python manage.py shell`, then import the serializer class, instantiate it, and print the object representation…

```
>>> from myapp.serializers import AccountSerializer
>>> serializer = AccountSerializer()
>>> print(repr(serializer))
AccountSerializer():
    id = IntegerField(label='ID', read_only=True)
    name = CharField(allow_blank=True, max_length=100, required=False)
    owner = PrimaryKeyRelatedField(queryset=User.objects.all())
```

Specifying which fields to include

If you only want a subset of the default fields to be used in a model serializer, you can do so using `fields` or `exclude` options, just as you would with a `ModelForm`. It is strongly recommended that you explicitly set all fields that should be serialized using the `fields` attribute. This will make it less likely to result in unintentionally exposing data when your models change.

For example:

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        fields = ('id', 'account_name', 'users', 'created')
```

You can also set the `fields` attribute to the special value `'__all__'` to indicate that all fields in the model should be used.

For example:

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        fields = '__all__'
```

You can set the `exclude` attribute to a list of fields to be excluded from the serializer.

For example:

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        exclude = ('users',)
```

In the example above, if the `Account` model had 3 fields `account_name`, `users`, and `created`, this will result in the fields `account_name` and `created` to be serialized.

The names in the `fields` and `exclude` attributes will normally map to model fields on the model class.

Alternatively names in the `fields` options can map to properties or methods which take no arguments that exist on the model class.

Specifying nested serialization

The default `ModelSerializer` uses primary keys for relationships, but you can also easily generate nested representations using the `depth` option:

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        fields = ('id', 'account_name', 'users', 'created')
        depth = 1
```

The `depth` option should be set to an integer value that indicates the depth of relationships that should be traversed before reverting to a flat representation.

If you want to customize the way the serialization is done you'll need to define the field yourself.

Specifying fields explicitly

You can add extra fields to a `ModelSerializer` or override the default fields by declaring fields on the class, just as you would for a `Serializer` class.

```
class AccountSerializer(serializers.ModelSerializer):
    url = serializers.CharField(source='get_absolute_url', read_only=True)
    groups = serializers.PrimaryKeyRelatedField(many=True)

    class Meta:
        model = Account
```

Extra fields can correspond to any property or callable on the model.

Specifying read only fields

You may wish to specify multiple fields as read-only. Instead of adding each field explicitly with the `read_only=True` attribute, you may use the shortcut Meta option, `read_only_fields`.

This option should be a list or tuple of field names, and is declared as follows:

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        fields = ('id', 'account_name', 'users', 'created')
        read_only_fields = ('account_name',)
```

Model fields which have `editable=False` set, and `AutoField` fields will be set to read-only by default, and do not need to be added to the `read_only_fields` option.

Note: There is a special-case where a read-only field is part of a `unique_together` constraint at the model level. In this case the field is required by the serializer class in order to validate the constraint, but should also not be editable by the user.

The right way to deal with this is to specify the field explicitly on the serializer, providing both the `read_only=True` and `default=…` keyword arguments.

One example of this is a read-only relation to the currently authenticated `User` which is `unique_together` with another identifier. In this case you would declare the user field like so:

```
user = serializers.PrimaryKeyRelatedField(read_only=True, default=serializers.CurrentUserDe
```

Please review the Validators Documentation for details on the UniqueTogetherValidator and CurrentUserDefault classes.

Additional keyword arguments

There is also a shortcut allowing you to specify arbitrary additional keyword arguments on fields, using the `extra_kwargs` option. As in the case of `read_only_fields`, this means you do not need to explicitly

declare the field on the serializer.

This option is a dictionary, mapping field names to a dictionary of keyword arguments. For example:

```
class CreateUserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ('email', 'username', 'password')
        extra_kwargs = {'password': {'write_only': True}}

    def create(self, validated_data):
        user = User(
            email=validated_data['email'],
            username=validated_data['username']
        )
        user.set_password(validated_data['password'])
        user.save()
        return user
```

Relational fields

When serializing model instances, there are a number of different ways you might choose to represent relationships. The default representation for `ModelSerializer` is to use the primary keys of the related instances.

Alternative representations include serializing using hyperlinks, serializing complete nested representations, or serializing with a custom representation.

For full details see the serializer relations documentation.

Customizing field mappings

The ModelSerializer class also exposes an API that you can override in order to alter how serializer fields are automatically determined when instantiating the serializer.

Normally if a `ModelSerializer` does not generate the fields you need by default then you should either add them to the class explicitly, or simply use a regular `Serializer` class instead. However in some cases you may want to create a new base class that defines how the serializer fields are created for any given model.

`.serializer_field_mapping`

A mapping of Django model classes to REST framework serializer classes. You can override this mapping to alter the default serializer classes that should be used for each model class.

`.serializer_related_field`

This property should be the serializer field class, that is used for relational fields by default.

For `ModelSerializer` this defaults to `PrimaryKeyRelatedField`.

For `HyperlinkedModelSerializer` this defaults to `serializers.HyperlinkedRelatedField`.

`serializer_url_field`

The serializer field class that should be used for any `url` field on the serializer.

Defaults to `serializers.HyperlinkedIdentityField`

`serializer_choice_field`

The serializer field class that should be used for any choice fields on the serializer.

Defaults to `serializers.ChoiceField`

The field_class and field_kwargs API

The following methods are called to determine the class and keyword arguments for each field that should be automatically included on the serializer. Each of these methods should return a two tuple of `(field_class, field_kwargs)`.

`.build_standard_field(self, field_name, model_field)`

Called to generate a serializer field that maps to a standard model field.

The default implementation returns a serializer class based on the `serializer_field_mapping` attribute.

`.build_relational_field(self, field_name, relation_info)`

Called to generate a serializer field that maps to a relational model field.

The default implementation returns a serializer class based on the `serializer_relational_field` attribute.

The `relation_info` argument is a named tuple, that contains `model_field`, `related_model`, `to_many` and `has_through_model` properties.

`.build_nested_field(self, field_name, relation_info, nested_depth)`

Called to generate a serializer field that maps to a relational model field, when the `depth` option has been set.

The default implementation dynamically creates a nested serializer class based on either `ModelSerializer` or `HyperlinkedModelSerializer`.

The `nested_depth` will be the value of the `depth` option, minus one.

The `relation_info` argument is a named tuple, that contains `model_field`, `related_model`, `to_many` and `has_through_model` properties.

`.build_property_field(self, field_name, model_class)`

Called to generate a serializer field that maps to a property or zero-argument method on the model class.

The default implementation returns a `ReadOnlyField` class.

`.build_url_field(self, field_name, model_class)`

Called to generate a serializer field for the serializer's own `url` field. The default implementation returns a `HyperlinkedIdentityField` class.

`.build_unknown_field(self, field_name, model_class)`

Called when the field name did not map to any model field or model property. The default implementation raises an error, although subclasses may customize this behavior.

HyperlinkedModelSerializer

The `HyperlinkedModelSerializer` class is similar to the `ModelSerializer` class except that it uses hyperlinks to represent relationships, rather than primary keys.

By default the serializer will include a `url` field instead of a primary key field.

The url field will be represented using a `HyperlinkedIdentityField` serializer field, and any relationships on the model will be represented using a `HyperlinkedRelatedField` serializer field.

You can explicitly include the primary key by adding it to the `fields` option, for example:

```
class AccountSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Account
        fields = ('url', 'id', 'account_name', 'users', 'created')
```

Absolute and relative URLs

When instantiating a `HyperlinkedModelSerializer` you must include the current `request` in the serializer context, for example:

```
serializer = AccountSerializer(queryset, context={'request': request})
```

Doing so will ensure that the hyperlinks can include an appropriate hostname, so that the resulting representation uses fully qualified URLs, such as:

```
http://api.example.com/accounts/1/
```

Rather than relative URLs, such as:

```
/accounts/1/
```

If you *do* want to use relative URLs, you should explicitly pass `{'request': None}` in the serializer context.

How hyperlinked views are determined

There needs to be a way of determining which views should be used for hyperlinking to model instances.

By default hyperlinks are expected to correspond to a view name that matches the style `'{model_name}-detail'`, and looks up the instance by a `pk` keyword argument.

You can override a URL field view name and lookup field by using either, or both of, the `view_name` and `lookup_field` options in the `extra_kwargs` setting, like so:

```
class AccountSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Account
        fields = ('account_url', 'account_name', 'users', 'created')
        extra_kwargs = {
            'url': {'view_name': 'accounts', 'lookup_field': 'account_name'},
            'users': {'lookup_field': 'username'}
        }
```

Alternatively you can set the fields on the serializer explicitly. For example:

```
class AccountSerializer(serializers.HyperlinkedModelSerializer):
    url = serializers.HyperlinkedIdentityField(
        view_name='accounts',
        lookup_field='slug'
    )
    users = serializers.HyperlinkedRelatedField(
        view_name='user-detail',
        lookup_field='username',
        many=True,
        read_only=True
    )

    class Meta:
        model = Account
        fields = ('url', 'account_name', 'users', 'created')
```

Tip: Properly matching together hyperlinked representations and your URL conf can sometimes be a bit fiddly. Printing the `repr` of a `HyperlinkedModelSerializer` instance is a particularly useful way to inspect exactly which view names and lookup fields the relationships are expected to map too.

Changing the URL field name

The name of the URL field defaults to 'url'. You can override this globally, by using the `URL_FIELD_NAME` setting.

ListSerializer

The `ListSerializer` class provides the behavior for serializing and validating multiple objects at once. You won't *typically* need to use `ListSerializer` directly, but should instead simply pass `many=True` when instantiating a serializer.

When a serializer is instantiated and `many=True` is passed, a `ListSerializer` instance will be created. The serializer class then becomes a child of the parent `ListSerializer`

The following argument can also be passed to a `ListSerializer` field or a serializer that is passed `many=True`:

`allow_empty`

This is `True` by default, but can be set to `False` if you want to disallow empty lists as valid input.

Customizing `ListSerializer` behavior

There *are* a few use cases when you might want to customize the `ListSerializer` behavior. For example:

- You want to provide particular validation of the lists, such as checking that one element does not conflict with another element in a list.
- You want to customize the create or update behavior of multiple objects.

For these cases you can modify the class that is used when `many=True` is passed, by using the `list_serializer_class` option on the serializer `Meta` class.

For example:

```
class CustomListSerializer(serializers.ListSerializer):
    ...

class CustomSerializer(serializers.Serializer):
    ...
    class Meta:
        list_serializer_class = CustomListSerializer
```

Customizing multiple create

The default implementation for multiple object creation is to simply call `.create()` for each item in the list. If you want to customize this behavior, you'll need to customize the `.create()` method on `ListSerializer` class that is used when `many=True` is passed.

For example:

```
class BookListSerializer(serializers.ListSerializer):
    def create(self, validated_data):
        books = [Book(**item) for item in validated_data]
        return Book.objects.bulk_create(books)

class BookSerializer(serializers.Serializer):
    ...
    class Meta:
        list_serializer_class = BookListSerializer
```

Customizing multiple update

By default the `ListSerializer` class does not support multiple updates. This is because the behavior that should be expected for insertions and deletions is ambiguous.

To support multiple updates you'll need to do so explicitly. When writing your multiple update code make sure to keep the following in mind:

- How do you determine which instance should be updated for each item in the list of data?
- How should insertions be handled? Are they invalid, or do they create new objects?
- How should removals be handled? Do they imply object deletion, or removing a relationship? Should they be silently ignored, or are they invalid?
- How should ordering be handled? Does changing the position of two items imply any state change or is it ignored?

You will need to add an explicit `id` field to the instance serializer. The default implicitly-generated `id` field is marked as `read_only`. This causes it to be removed on updates. Once you declare it explicitly, it will be available in the list serializer's `update` method.

Here's an example of how you might choose to implement multiple updates:

```
class BookListSerializer(serializers.ListSerializer):
    def update(self, instance, validated_data):
        # Maps for id->instance and id->data item.
        book_mapping = {book.id: book for book in instance}
        data_mapping = {item['id']: item for item in validated_data}

        # Perform creations and updates.
        ret = []
        for book_id, data in data_mapping.items():
            book = book_mapping.get(book_id, None)
            if book is None:
                ret.append(self.child.create(data))
            else:
                ret.append(self.child.update(book, data))

        # Perform deletions.
        for book_id, book in book_mapping.items():
            if book_id not in data_mapping:
                book.delete()

        return ret
```

```
class BookSerializer(serializers.Serializer):
    # We need to identify elements in the list using their primary key,
    # so use a writable field here, rather than the default which would be read-only.
    id = serializers.IntegerField()

    ...
    id = serializers.IntegerField(required=False)

    class Meta:
        list_serializer_class = BookListSerializer
```

It is possible that a third party package may be included alongside the 3.1 release that provides some automatic support for multiple update operations, similar to the `allow_add_remove` behavior that was present in REST framework 2.

Customizing ListSerializer initialization

When a serializer with `many=True` is instantiated, we need to determine which arguments and keyword arguments should be passed to the `.__init__()` method for both the child `Serializer` class, and for the parent `ListSerializer` class.

The default implementation is to pass all arguments to both classes, except for `validators`, and any custom keyword arguments, both of which are assumed to be intended for the child serializer class.

Occasionally you might need to explicitly specify how the child and parent classes should be instantiated when `many=True` is passed. You can do so by using the `many_init` class method.

```
    @classmethod
    def many_init(cls, *args, **kwargs):
        # Instantiate the child serializer.
        kwargs['child'] = cls()
        # Instantiate the parent list serializer.
        return CustomListSerializer(*args, **kwargs)
```

BaseSerializer

`BaseSerializer` class that can be used to easily support alternative serialization and deserialization styles.

This class implements the same basic API as the `Serializer` class:

- `.data` - Returns the outgoing primitive representation.
- `.is_valid()` - Deserializes and validates incoming data.
- `.validated_data` - Returns the validated incoming data.
- `.errors` - Returns any errors during validation.
- `.save()` - Persists the validated data into an object instance.

There are four methods that can be overridden, depending on what functionality you want the serializer class to support:

- `.to_representation()` - Override this to support serialization, for read operations.
- `.to_internal_value()` - Override this to support deserialization, for write operations.
- `.create()` and `.update()` - Override either or both of these to support saving instances.

Because this class provides the same interface as the `Serializer` class, you can use it with the existing generic class-based views exactly as you would for a regular `Serializer` or `ModelSerializer`.

The only difference you'll notice when doing so is the `BaseSerializer` classes will not generate HTML forms in the browsable API. This is because the data they return does not include all the field information that would allow each field to be rendered into a suitable HTML input.

Read-only `BaseSerializer` classes

To implement a read-only serializer using the `BaseSerializer` class, we just need to override the `.to_representation()` method. Let's take a look at an example using a simple Django model:

```
class HighScore(models.Model):
    created = models.DateTimeField(auto_now_add=True)
    player_name = models.CharField(max_length=10)
    score = models.IntegerField()
```

It's simple to create a read-only serializer for converting `HighScore` instances into primitive data types.

```
class HighScoreSerializer(serializers.BaseSerializer):
    def to_representation(self, obj):
        return {
            'score': obj.score,
            'player_name': obj.player_name
        }
```

We can now use this class to serialize single `HighScore` instances:

```
@api_view(['GET'])
def high_score(request, pk):
    instance = HighScore.objects.get(pk=pk)
    serializer = HighScoreSerializer(instance)
    return Response(serializer.data)
```

Or use it to serialize multiple instances:

```
@api_view(['GET'])
def all_high_scores(request):
    queryset = HighScore.objects.order_by('-score')
    serializer = HighScoreSerializer(queryset, many=True)
    return Response(serializer.data)
```

Read-write `BaseSerializer` classes

To create a read-write serializer we first need to implement a `.to_internal_value()` method. This method returns the validated values that will be used to construct the object instance, and may raise a `ValidationError` if the supplied data is in an incorrect format.

Once you've implemented `.to_internal_value()`, the basic validation API will be available on the serializer, and you will be able to use `.is_valid()`, `.validated_data` and `.errors`.

If you want to also support `.save()` you'll need to also implement either or both of the `.create()` and `.update()` methods.

Here's a complete example of our previous `HighScoreSerializer`, that's been updated to support both read and write operations.

```
class HighScoreSerializer(serializers.BaseSerializer):
    def to_internal_value(self, data):
        score = data.get('score')
        player_name = data.get('player_name')

        # Perform the data validation.
        if not score:
            raise ValidationError({
                'score': 'This field is required.'
            })
        if not player_name:
            raise ValidationError({
                'player_name': 'This field is required.'
            })
        if len(player_name) > 10:
            raise ValidationError({
                'player_name': 'May not be more than 10 characters.'
            })

        # Return the validated values. This will be available as
        # the `.validated_data` property.
        return {
            'score': int(score),
            'player_name': player_name
        }

    def to_representation(self, obj):
        return {
            'score': obj.score,
            'player_name': obj.player_name
        }

    def create(self, validated_data):
        return HighScore.objects.create(**validated_data)
```

Creating new base classes

The `BaseSerializer` class is also useful if you want to implement new generic serializer classes for dealing with particular serialization styles, or for integrating with alternative storage backends.

The following class is an example of a generic serializer that can handle coercing arbitrary objects into primitive representations.

```
class ObjectSerializer(serializers.BaseSerializer):
    """
    A read-only serializer that coerces arbitrary complex objects
    into primitive representations.
    """
    def to_representation(self, obj):
        for attribute_name in dir(obj):
            attribute = getattr(obj, attribute_name)
            if attribute_name('_'):
                # Ignore private attributes.
                pass
            elif hasattr(attribute, '__call__'):
                # Ignore methods and other callables.
                pass
            elif isinstance(attribute, (str, int, bool, float, type(None))):
                # Primitive types can be passed through unmodified.
                output[attribute_name] = attribute
            elif isinstance(attribute, list):
                # Recursively deal with items in lists.
                output[attribute_name] = [
                    self.to_representation(item) for item in attribute
                ]
            elif isinstance(attribute, dict):
                # Recursively deal with items in dictionaries.
                output[attribute_name] = {
                    str(key): self.to_representation(value)
                    for key, value in attribute.items()
                }
            else:
                # Force anything else to its string representation.
                output[attribute_name] = str(attribute)
```

Advanced serializer usage

Overriding serialization and deserialization behavior

If you need to alter the serialization, deserialization or validation of a serializer class you can do so by overriding the `.to_representation()` or `.to_internal_value()` methods.

Some reasons this might be useful include...

- Adding new behavior for new serializer base classes.
- Modifying the behavior slightly for an existing class.
- Improving serialization performance for a frequently accessed API endpoint that returns lots of data.

The signatures for these methods are as follows:

`.to_representation(self, obj)`

Takes the object instance that requires serialization, and should return a primitive representation. Typically this means returning a structure of built-in Python datatypes. The exact types that can be handled will depend on the render classes you have configured for your API.

`.to_internal_value(self, data)`

Takes the unvalidated incoming data as input and should return the validated data that will be made available as `serializer.validated_data`. The return value will also be passed to the `.create()` or `.update()` methods if `.save()` is called on the serializer class.

If any of the validation fails, then the method should raise a `serializers.ValidationError(errors)`. Typically the `errors` argument here will be a dictionary mapping field names to error messages.

The `data` argument passed to this method will normally be the value of `request.data`, so the datatype it provides will depend on the parser classes you have configured for your API.

Serializer Inheritance

Similar to Django forms, you can extend and reuse serializers through inheritance. This allows you to declare a common set of fields or methods on a parent class that can then be used in a number of serializers. For example,

```
class MyBaseSerializer(Serializer):
    my_field = serializers.CharField()

    def validate_my_field(self):
        ...

class MySerializer(MyBaseSerializer):
    ...
```

Like Django's `Model` and `ModelForm` classes, the inner `Meta` class on serializers does not implicitly inherit from it's parents' inner `Meta` classes. If you want the `Meta` class to inherit from a parent class you must do so explicitly. For example:

```
class AccountSerializer(MyBaseSerializer):
    class Meta(MyBaseSerializer.Meta):
        model = Account
```

Typically we would recommend *not* using inheritance on inner Meta classes, but instead declaring all options explicitly.

Additionally, the following caveats apply to serializer inheritance:

- Normal Python name resolution rules apply. If you have multiple base classes that declare a `Meta` inner class, only the first one will be used. This means the child's `Meta`, if it exists, otherwise the `Meta` of the first parent, etc.
- It's possible to declaratively remove a `Field` inherited from a parent class by setting the name to be `None` on the subclass.

```
class MyBaseSerializer(ModelSerializer):
    my_field = serializers.CharField()

class MySerializer(MyBaseSerializer):
    my_field = None
```

However, you can only use this technique to opt out from a field defined declaratively by a parent class; it won't prevent the `ModelSerializer` from generating a default field. To opt-out from default fields, see Specifying which fields to include.

Dynamically modifying fields

Once a serializer has been initialized, the dictionary of fields that are set on the serializer may be accessed using the `.fields` attribute. Accessing and modifying this attribute allows you to dynamically modify the serializer.

Modifying the `fields` argument directly allows you to do interesting things such as changing the arguments on serializer fields at runtime, rather than at the point of declaring the serializer.

Example

For example, if you wanted to be able to set which fields should be used by a serializer at the point of initializing it, you could create a serializer class like so:

```
class DynamicFieldsModelSerializer(serializers.ModelSerializer):
    """
    A ModelSerializer that takes an additional `fields` argument that
    controls which fields should be displayed.
    """

    def __init__(self, *args, **kwargs):
        # Don't pass the 'fields' arg up to the superclass
        fields = kwargs.pop('fields', None)

        # Instantiate the superclass normally
        super(DynamicFieldsModelSerializer, self).__init__(*args, **kwargs)

        if fields is not None:
            # Drop any fields that are not specified in the `fields` argument.
            allowed = set(fields)
            existing = set(self.fields.keys())
            for field_name in existing - allowed:
                self.fields.pop(field_name)
```

This would then allow you to do the following:

```
>>> class UserSerializer(DynamicFieldsModelSerializer):
>>>     class Meta:
>>>         model = User
>>>         fields = ('id', 'username', 'email')
>>>
>>> print UserSerializer(user)
{'id': 2, 'username': 'jonwatts', 'email': 'jon@example.com'}
>>>
>>> print UserSerializer(user, fields=('id', 'email'))
{'id': 2, 'email': 'jon@example.com'}
```

Customizing the default fields

REST framework 2 provided an API to allow developers to override how a `ModelSerializer` class would automatically generate the default set of fields.

This API included the `.get_field()`, `.get_pk_field()` and other methods.

Because the serializers have been fundamentally redesigned with 3.0 this API no longer exists. You can still modify the fields that get created but you'll need to refer to the source code, and be aware that if the changes you make are against private bits of API then they may be subject to change.

Third party packages

The following third party packages are also available.

Django REST marshmallow

The django-rest-marshmallow package provides an alternative implementation for serializers, using the python marshmallow library. It exposes the same API as the REST framework serializers, and can be used as a drop-in replacement in some use-cases.

Serpy

The serpy package is an alternative implementation for serializers that is built for speed. Serpy serializes complex datatypes to simple native types. The native types can be easily converted to JSON or any other format needed.

MongoengineModelSerializer

The django-rest-framework-mongoengine package provides a `MongoEngineModelSerializer` serializer class that supports using MongoDB as the storage layer for Django REST framework.

GeoFeatureModelSerializer

The django-rest-framework-gis package provides a `GeoFeatureModelSerializer` serializer class that supports GeoJSON both for read and write operations.

HStoreSerializer

The django-rest-framework-hstore package provides an `HStoreSerializer` to support django-hstore `DictionaryField` model field and its `schema-mode` feature.

Dynamic REST

The dynamic-rest package extends the ModelSerializer and ModelViewSet interfaces, adding API query parameters for filtering, sorting, and including / excluding all fields and relationships defined by your serializers.

Dynamic Fields Mixin

The drf-dynamic-fields package provides a mixin to dynamically limit the fields per serializer to a subset specified by an URL parameter.

DRF FlexFields

The drf-flex-fields package extends the ModelSerializer and ModelViewSet to provide commonly used functionality for dynamically setting fields and expanding primitive fields to nested models, both from URL parameters and your serializer class definitions.

Serializer Extensions

The django-rest-framework-serializer-extensions package provides a collection of tools to DRY up your serializers, by allowing fields to be defined on a per-view/request basis. Fields can be whitelisted, blacklisted and child serializers can be optionally expanded.

HTML JSON Forms

The html-json-forms package provides an algorithm and serializer for processing `<form>` submissions per the (inactive) HTML JSON Form specification. The serializer facilitates processing of arbitrarily nested JSON structures within HTML. For example, `<input name="items[0][id]" value="5">` will be interpreted as `{"items": [{"id": "5"}]}`.

DRF-Base64

DRF-Base64 provides a set of field and model serializers that handles the upload of base64-encoded files.

QueryFields

djangorestframework-queryfields allows API clients to specify which fields will be sent in the response via inclusion/exclusion query parameters.

DRF Writable Nested

The drf-writable-nested package provides writable nested model serializer which allows to create/update models with nested related data.
