Learning Objectives

- **Explain the benefits of Django Rest Framework**
- **Define the role of a serializer**
- **Create a serializer for the `User` model**
- **Hide passwords when creating serializers**
- **Deserialize JSON data**
- **Raise an exception when serializing invalid data**
- **Create objects from validated data**
- **Serialize and deserialize multiple objects at once**
- **Provide custom validation with field-level validation and validator functions**
- **Differentiate a `ModelSerializer` from a `Serializer`**
- **Add a `ModelSerializer` for the `Post` model**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

Django Rest Framework

Django Rest Framework

Introduction

In Module 1 we coded up a basic REST API using just Django's built-in classes. But, it was brittle and insecure. For example:

- There was no authentication.
- We didn't validate the input.
- Exceptions returned as HTML, not converted to JSON.
- Data had to be in JSON format.
- There was no support for relationships.
- No pagination or filtering.
- No way to explore the API.
- We had to use an external application (e.g. Postman) to perform anything other than `GET`.

Yes, we could spend time coding solutions to all these problems, but luckily there's a reliable third-party library that's already done that all for us: Django Rest Framework. It's easy to install and configure, and makes building APIs easy. We'll jump right into it on the next page, starting with *Serializers*.

Serializers

Introduction

Probably the most fundamental part of an API is converting data from the native variables/objects into a serialized form that's suitable for transmission. For example, converting Python objects or Django model instances into a JSON string, and back again.

Django Rest Framework (or DRF for short) uses *Serializer* classes to perform this back and forth conversion. You can kind-of think of them like Django forms, in a very loose way. That is, a Django forms can use Python data to populate the initial values of a form for display in HTML on a web page. The same form class will accept data posted by the browser, validate it, then convert it to Python objects.

In DRF, a serializer converts Python objects into a form suitable for transmission over HTTP. It accepts data in an HTTP request body, validates it, and then converts it back to Python.

Another similarity is that like how Django provides `Form` and `ModelForm` base classes, DRF provides `Serializer` and `ModelSerializer` classes; the latter makes it easier to automatically create a serializer from existing models. But, let's start by looking at the `Serializer` class, then come back to `ModelSerializer`.

Serializers

Setup

When using serializers, you don't ever work with the base `Serializer` (`rest_framework.serializers.Serializer`) class directly. Instead, like with Django forms, you subclass it and add fields, then interact with that subclass.

Serializer fields are also kind of like Django form or model fields. They have a type, can have validation rules (like if they're required, or have a maximum length), and perform the conversions to and from Python native objects.

Manually configuring a basic serializer is straightforward. A serializer can be used to serialize or deserialize any type of object, provided that the fields of each match. Start by installing DRF.

```
pip3 install djangorestframework
```

Let's look at a short example. First we'll define a class that represents a user. This looks a lot like a Django `User` class but it's just a plain Python object. These examples will use the Python shell, so start it before continuing.

```
python3 manage.py shell
```

▼ **Copying and pasting in the shell**
You can copy and paste blocks of code in the Python shell. Press **Enter** two times after pasting to execute the newly copied code.

```
from django.utils import timezone


class User:
    def __init__(self, username, email=None, first_name=None,
        last_name=None, password=None, join_date=None):
        self.username = username
        self.email = email
        self.first_name = first_name
        self.last_name = last_name
        self.password = password
        self.join_date = join_date or timezone.now()
```

Now let's look at how to write a serializer for a user. This `UserSerializer` class will do:

```
from rest_framework import serializers


class UserSerializer(serializers.Serializer):
    username = serializers.CharField()
    email = serializers.EmailField(required=False)
    first_name = serializers.CharField(max_length=20,
        required=False)
    last_name = serializers.CharField(max_length=20,
        required=False)
    password = serializers.CharField(write_only=True,
        required=False)
    join_date = serializers.DateTimeField(read_only=True)
```

You can see how the serializer fields are configured and how they are analogous to other Django fields. We'll go through what we're using.

- `username` is a `CharField` which is for strings.

- `email`: is an `EmailField` for working with email addresses. We don't require it to create a `User`, so `required` is set to `False`.
- `first_name` and `last_name` are both `CharFields`. Neither are required, and both have a `max_length` of `20` characters.
- `password` has the `write_only` argument which means it won't be included in the output when the `User` object is serialized. It's also not required, which means a user can update other fields without updating their password.
- `join_date` is a `DateTimeField` and stores a date and time. It's read-only, so user's can't set this. Because of this it's not required either.

▼ **Documentation**
The full documentation for DRF Serializer Fields can be found in the official documentation.

Serializing

Now let's see it in action, note that these are running in a Django Python shell. First we'll create a user:

```
u = User("cwilson", "cwilson@example.com", "Callum",
         password="p4ssw0rd")
```

Then, we instantiate the `UserSerializer`, passing in the `User` object.

```
s = UserSerializer(u)
```

And we can get the serialized representation of the user by accessing the `data` attribute on the serializer.

```
s.data
```

You should see output similar to this:

```
ReturnDict([('username', 'cwilson'),
            ('email', 'cwilson@example.com'),
            ('first_name', 'Callum'),
            ('last_name', None),
            ('join_date', '2021-09-02T19:24:02.520184Z')])
```

There should be no surprises here, the `User` object has been transformed into a dictionary. You don't see a `password` entry in the output, since it's write-only, and the `last_name` is `None` (just as it was on the object itself).

The serialized data is kind of like the output of the `post_to_dict` function in Blango, or `fruit_to_dict` in the last coding exercise. However as we'll see soon, it's much better as it can go in both directions.

▼ **Serialized data**

It's important to point out here that while the syntax of Python dictionaries and JSON objects look similar, the serialized data is **not** JSON. We might want to send the data over HTTP using some format other than JSON, and so the dictionary-to-string process is separate. We could use Python's built in `json` library, for example:

```
import json
json.dumps(s.data)
```

You should see the following output:

```
  '{"username": "cwilson", "email": "cwilson@example.com",
"first_name": "Callum", "last_name": null, "join_date": "2021-
09-02T19:32:00.994050Z"}'
```

There are other ways of working with the serialized data too. DRF provides its own `JSONRenderer` and `JSONParser` classes, or we could use the dictionary in a Django `JsonResponse` class. However, we won't need to worry about any of that as we'll just be using DRF's views to perform the final serialization for us. We'll cover that in the next section, but for now, let's return to deserializing and validation.

Deserializing

Deserializing

Enter the `User` and `UserSerializer` classes before continuing.

```
from django.utils import timezone
from rest_framework import serializers


class User:
  def __init__(self, username, email=None, first_name=None,
        last_name=None, password=None, join_date=None):
    self.username = username
    self.email = email
    self.first_name = first_name
    self.last_name = last_name
    self.password = password
    self.join_date = join_date or timezone.now()

class UserSerializer(serializers.Serializer):
  username = serializers.CharField()
  email = serializers.EmailField(required=False)
  first_name = serializers.CharField(max_length=20,
        required=False)
  last_name = serializers.CharField(max_length=20,
        required=False)
  password = serializers.CharField(write_only=True,
        required=False)
  join_date = serializers.DateTimeField(read_only=True)
```

To validate the data dictionary, we also use the same serializer. However we pass in the dictionary into the `data` argument of the constructor.

```
u2 = {"username": "tharrison", "join_date": "2021-08-
09T22:15:27.934034Z"}
s2 = UserSerializer(data=u2)
```

There are three ways we can access the data, the first is with the `initial_data` attribute. This is the `data` we passed in:

```
s2.initial_data
```

Your output should look something like this:

```
{'join_date': '2021-08-09T22:15:27.934034Z', 'username':
        'tharrison'}
```

The next way is with the `data` attribute, however this can only be accessed once you've called the `is_valid()` method on the serializer.

```
s2.is_valid()
```

Your output should be:

```
True
```

Now call the `data` attribute.

```
s2.data
```

Your output should be:

```
ReturnDict([('username', 'tharrison')])
```

This contains the initial data, without any fields that aren't writable in the serializer. For example, since `join_date` is not writable, it's not in the `data`.

And finally, you can read the `validated_data`, which is only populated if the data provided is valid.

```
s2.validated_data
```

Python should return the following:

```
OrderedDict([('username', 'tharrison')])
```

On our case, we provided valid data so `data` and `validated_data` are the same. But what happens if we provide some invalid data? Let's see how all these dictionaries differ. We'll pass in some data that has an extra field and a last name that's too long.

```
u3 = {"username": "blonglastname", "first_name": "Brandon",
        "last_name": "This Is 26 Characters Long",
        "some_other_key": "extra"}
s3 = UserSerializer(data=u3)
s3.initial_data
```

`initial_data` contains all the data we passed in when creating `u3`.

```
{'first_name': 'Brandon',
 'last_name': 'This Is 26 Characters Long',
 'some_other_key': 'extra',
 'username': 'blonglastname'}
```

Now let's see if the data is valid.

```
s3.is_valid()
```

Because the last name is too long and because of the extra field, the data is not valid.

```
False
```

Now look at the `data` dictionary.

```
s3.data
```

```
{'username': 'blonglastname', 'first_name': 'Brandon',
        'last_name': 'This Is 26 Characters Long'}
```

It still contains all the information we passed in for any fields on the serializer. Note that only the `some_other_key` has been removed, the other items are still there verbatim.

Finally let's look at `validated_data`.

```
s3.validated_data
```

Python should return an empty dictionary because the data is not valid.

```
{}
```

We saw that `is_valid()` returned `False` for invalid data. If we want to find out more about what caused the data to not be valid, there are a couple of ways. First, we can investigate the `errors` attribute on a serializer, which is populated when `is_valid` returns `False`.

```
s3.errors
```

Python returns a dictionary with the error.

```
{'last_name': [ErrorDetail(string='Ensure this field has no more
        than 20 characters.', code='max_length')]}
```

Or, if you prefer, you can have the `is_valid()` method raise an exception for invalid fields, by passing the argument `raise_exception=True`.

```
s3.is_valid(raise_exception=True)
```

Python returns the exception.

```
---------------------------------------------------------------
        -----------
ValidationError                           Traceback (most recent
        call last)
<ipython-input-3-539bd46e65bd> in <module>()
----> 1 s3.is_valid(raise_exception=True)

/home/codio/.local/lib/python3.6/site-
        packages/rest_framework/serializers.py in is_valid(self,
        raise_exception)
    226
    227         if self._errors and raise_exception:
--> 228             raise ValidationError(self.errors)
    229
    230         return not bool(self._errors)

ValidationError: {'last_name': [ErrorDetail(string='Ensure this
        field has no more than 20 characters.',
        code='max_length')]}}
```

You might prefer to use the exception-raising method when working with serializers in a Django view as it is safer. The exception will automatically bubble up and prevent the request from completing, and you won't be working with bad data. Without the `raise_exception` argument, there's a chance you forget to check `is_valid()` returns `False`, and then attempt to work with incomplete data.

On the other hand, if you want to show the errors to the user, you should check that `is_valid()` returns `False` and then return a response containing `serializer.errors`.

Saving and Multiple Objects

Saving Data

Now that we have validated data, how can we work with it? It's just a normal dictionary so you could use it to "manually" update or create an object. For example:

```
validated_data = serializer.validated_data
user.first_name = validated_data["first_name"]
user.last_name = validated_data["last_name"]
```

or

```
new_user = User(**validated_data)
```

But DRF serializers have built-in ways to take care of this, with the `create()` and `update()` methods.

`create()` takes a single argument, the `validated_data` dictionary. It should return a new instance built with this data.

For example:

```
class UserSerializer(serializers.Serializer):
    # field definitions truncated

    def create(self, validated_data):
        return User(**validated_data)
```

We know that `validated_data` will only have keys that are valid to pass to the `__init__()` method of `User` (because we defined the serializer fields accordingly), so it's this simple to create a new `User`.

The `update()` method takes two arguments, the `instance` to update, and the `validated_data` dictionary. We could update an existing `User` like this (copy/paste this code into the Python shell):

```
from django.utils import timezone
from rest_framework import serializers


class User:
    def __init__(self, username, email=None, first_name=None,
        last_name=None, password=None, join_date=None):
        self.username = username
        self.email = email
        self.first_name = first_name
        self.last_name = last_name
        self.password = password
        self.join_date = join_date or timezone.now()


class UserSerializer(serializers.Serializer):
    username = serializers.CharField()
    email = serializers.EmailField(required=False)
    first_name = serializers.CharField(max_length=20,
        required=False)
    last_name = serializers.CharField(max_length=20,
        required=False)
    password = serializers.CharField(write_only=True,
        required=False)
    join_date = serializers.DateTimeField(read_only=True)


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


    def update(self, instance, validated_data):
        for key, value in validated_data.items():
            setattr(instance, key, value)
        return instance

user = User("cwilson", "cwilson@example.com", "Callum",
        password="p4ssw0rd")
```

Since we know `validated_data` only contains values for keys that were posted, we know we can update the `instance` without overwriting any other data that the user might not want to update.

How should we call the `create()` and `update()` methods? They are not designed to be called by us, instead we should call the `save()` method on the serializer, which will call the correct method based on how it was instantiated (with an initial instance or not).

For example, if we use the serializer like this:

```
user_data = {"username": "tharrison"}
serializer = UserSerializer(data=user_data)
serializer.is_valid()  # must be called before save
```

This returns `True` as the data is valid.

```
True
```

Now we can call the `save` method.

```
serializer.save()
```

Then this will call the `create()` method, as a `User` instance wasn't passed to the `UserSerializer` on instantiation.

```
<__main__.User at 0x7f6cac20ffd0>
```

On the contrary, we could provide a `User` instance, like this:

```
user_data = {"username": "tharrison", "first_name": "Tony",
        "last_name": "Harrison"}
serializer = UserSerializer(user, data=user_data)  # pass User
        instance as first argument
serializer.is_valid()
```

You should see `True` as the output.

```
True
```

```
user2 = serializer.save()
```

`save()` then calls the `update()` method, passing in the `User` object that was used to instantiate the `UserSerializer` and the `validated_data` dictionary. We can validate this by examining the object that was returned.

Return the value for `first_name`.

```
user2.first_name
```

This should be `Tony`.

```
'Tony'
```

Return the value for `last_name`.

```
user2.last_name
```

This should be `Harrison`.

```
'Harrison'
```

Now check if `user2` and `user` point to the the same object.

```
user2 is user
```

This should be `True`.

```
True
```

Multiple Objects

We can serialize or deserialize multiple objects at once, by passing `many=True` to the serializer constructor. Here's serializing to a list of dictionaries:

```
users = [
User("lduffy"),
User("blongname"),
User("tharrison")
]
multi_serializer = UserSerializer(users, many=True)
multi_serializer.data
```

```
[OrderedDict([('username', 'lduffy'), ('email', None),
        ('first_name', None), ('last_name', None), ('join_date',
        '2021-08-10T00:37:14.703118Z')]),
        OrderedDict([('username', 'blongname'), ('email', None),
        ('first_name', None), ('last_name', None), ('join_date',
        '2021-08-10T00:37:14.703144Z')]),
        OrderedDict([('username', 'tharrison'), ('email', None),
        ('first_name', None), ('last_name', None), ('join_date',
        '2021-08-10T00:37:14.703152Z')])]
```

And here's deserializing a list of dictionaries, then saving:

```
user_dicts = [{"username": "tford"}, {"username": "brabbit"},
        {"username": "cbob"}]
multi_serializer = UserSerializer(data=user_dicts, many=True)
multi_serializer.is_valid()
```

You should see `True` as the output.

```
True
```

We can take the list of deserialized dictionaries and turn them into `User` objects.

```
new_users = multi_serializer.save()
new_users
```

You should see the memory address for each object.

```
[<User object at 0x10be0b1f0>, <User object at 0x10be0b370>,
        <User object at 0x10be48e80>]
```

Validate this by extracting the `username` for each object.

```
print(", ".join(map(lambda u: u.username, new_users)))
```

Python returns the value for `username` of each object.

```
tford, brabbit, cbob
```

When deserializing multiple objects, the `save()` method only supports creation, due to uncertainties of mapping together the passed-in instances and the data from the dictionaries.

Validation

Custom Validation

Another similarity with Django forms, is that serializers and fields can have custom validation rules applied to them. There are three methods of performing custom validation: field-level validation, object-level validation, and validator functions.

Field-Level Validation

These are methods added to your `Serializer`, named in the format `validate_<field_name>()`. For example, to validate the `email field` name it `validate_email()`. This is similar to the `clean_<field_name>()` method on a Django form. The method
is passed the deserialized value as an argument, and should return the value if it's valid. This means as well as validating you can transform the value inside the validate method.

Validate methods are only called if the field classes' own validators pass. For example, our `clean_email()` method won't be called if the data doesn't contain a valid email address (`EmailField` performs this check for us). Similarly, a `validate_first_name()` method wouldn't be called if the name were too long.

If the value is invalid, a `serializers.ValidationError` should be raised, with an error message describing why it's invalid.

We can add a validate method to ensure that a user's email is on the `example.com` domain. The validate method will also convert the email address to lowercase.

```
from rest_framework import serializers

class UserSerializer(serializers.Serializer):
    username = serializers.CharField()
    email = serializers.EmailField(required=False)
    first_name = serializers.CharField(max_length=20,
        required=False)
    last_name = serializers.CharField(max_length=20,
        required=False)
    password = serializers.CharField(write_only=True,
        required=False)
    join_date = serializers.DateTimeField(read_only=True)

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

    def update(self, instance, validated_data):
        for key, value in validated_data.items():
            setattr(instance, key, value)
        return instance

    def validate_email(self, value):
        value = value.lower()
        domain = value.split("@")[1]  # safe to do since we know
        value is valid email address
        if domain != "example.com":
            raise serializers.ValidationError("domain must be
        example.com")
        return value
```

Here's what happens if we call it with bad data:

```
u = UserSerializer(data={"username": "evernon", "email":
        "User@NotExample.com"})
u.is_valid()
```

```
False
```

Take a look at the errors associated with data.

```
u.errors
```

Python returns the validation error specified in the `UserSerializer` class.

```
ReturnDict([('email', [ErrorDetail(string='domain must be
        example.com', code='invalid')])])
```

And with good data, we can see the email gets converted to lowercase.

```
u = UserSerializer(data={"username": "evernon", "email":
        "User@Example.com"})
u.is_valid()
```

Python returns `True` as there are no problems with the data.

```
True
```

Take a look at the validate data.

```
u.validated_data
```

Python returns the data with the email address being lowercase.

```
OrderedDict([('username', 'evernon'), ('email',
        'user@example.com')])
```

Object-Level Validation

If you need to validate two different fields against each other, you should use object-level validation. This is done by adding a `validate()` method to serializer class. It accepts a single argument, `data`, which is a dictionary of deserialized data. This dictionary should also be returned from the function. You can validate data, raising a `serializers.ValidationError` on error, or make changes to the `data` dictionary if you choose. You might notice that this is similar to the behavior of the `clean()` method on a Django form.

As an example, let's make our `UserSerializer` require a user to supply a first name if they provide a last name, and vice versa (i.e. they will need to provide either both names, or neither, but not just one).

```
class UserSerializer(serializers.Serializer):
    username = serializers.CharField()
    email = serializers.EmailField(required=False)
    first_name = serializers.CharField(max_length=20,
        required=False)
    last_name = serializers.CharField(max_length=20,
        required=False)
    password = serializers.CharField(write_only=True,
        required=False)
    join_date = serializers.DateTimeField(read_only=True)

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

    def update(self, instance, validated_data):
        for key, value in validated_data.items():
            setattr(instance, key, value)
        return instance

    def validate_email(self, value):
        value = value.lower()
        domain = value.split("@")[1]  # safe to do since we know
        value is valid email address
        if domain != "example.com":
            raise serializers.ValidationError("domain must be
        example.com")
        return value

    def validate(self, data):
        if (not data.get("first_name")) != (not
        data.get("last_name")):
            raise serializers.ValidationError("first_name and
        last_name must be provided together")

        return data
```

Here it is in action with some bad data:

```
UserSerializer(data={"username": "evernon", "first_name":
        "Eve"}).is_valid(raise_exception=True)
```

Python raises an exception because a last name is missing.

```
---------------------------------------------------------------
        -----------
ValidationError                           Traceback (most recent
        call last)
<ipython-input-8-35e9b9819ea2> in <module>()
----> 1 UserSerializer(data={"username": "evernon",
        "first_name": "Eve"}).is_valid(raise_exception=True)

/home/codio/.local/lib/python3.6/site-
        packages/rest_framework/serializers.py in is_valid(self,
        raise_exception)
    226
    227         if self._errors and raise_exception:
--> 228             raise ValidationError(self.errors)
    229
    230         return not bool(self._errors)

ValidationError: {'non_field_errors':
        [ErrorDetail(string='first_name and last_name must be
        provided together', code='invalid')]}
```

Try the same thing but have a last name but no first name.

```
UserSerializer(data={"username": "evernon", "last_name":
        "Vernon"}).is_valid(raise_exception=True)
```

Python raises the same exception.

```
---------------------------------------------------------------
        -----------
ValidationError                           Traceback (most recent
        call last)
<ipython-input-3-06f43fe0df5f> in <module>()
----> 1 UserSerializer(data={"username": "evernon", "last_name":
        "Vernon"}).is_valid(raise_exception=True)

/home/codio/.local/lib/python3.6/site-
        packages/rest_framework/serializers.py in is_valid(self,
        raise_exception)
    226
    227         if self._errors and raise_exception:
--> 228             raise ValidationError(self.errors)
    229
    230         return not bool(self._errors)

ValidationError: {'non_field_errors':
        [ErrorDetail(string='first_name and last_name must be
        provided together', code='invalid')]}
```

And with some good data, validation passes:

```
UserSerializer(data={"username":
        "evernon"}).is_valid(raise_exception=True)
UserSerializer(data={"username": "evernon", "first_name": "Eve",
        "last_name": "Vernon"}).is_valid(raise_exception=True)
```

Validator Functions

The final way of adding validation is with validator functions. These are functions that have one argument, the value to validate. They should raise `serializers.ValidationError` if an exception occurs, or just do nothing (and not return anything) if the value is valid. They are useful if you want to apply the same validation to multiple fields.

For our `UserSerializer`, we want to make sure users capitalize their first and last names, so we can write a function to validate this:

```
def is_capitalized(value):
    if value[0].lower() == value[0]:
        raise serializers.ValidationError("Value must be
        capitalized")
```

The validator is passed to a field class using the `validators` argument, which must be a list of validators (it might contain just one value though).

We can pass the validator function to our name fields like this:

```
class UserSerializer(serializers.Serializer):
    username = serializers.CharField()
    email = serializers.EmailField(required=False)
    first_name = serializers.CharField(
        max_length=20,
        required=False,
        validators=[is_capitalized]
    )
    last_name = serializers.CharField(
        max_length=20,
        required=False,
        validators=[is_capitalized]
    )
    password = serializers.CharField(write_only=True,
        required=False)
    join_date = serializers.DateTimeField(read_only=True)

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

    def update(self, instance, validated_data):
        for key, value in validated_data.items():
            setattr(instance, key, value)
        return instance

    def validate_email(self, value):
        value = value.lower()
        domain = value.split("@")[1]  # safe to do since we know
        value is valid email address
        if domain != "example.com":
            raise serializers.ValidationError("domain must be
        example.com")
        return value

    def validate(self, data):
        if (not data.get("first_name")) != (not
        data.get("last_name")):
            raise serializers.ValidationError("first_name and
        last_name must be provided together")


        return data
```

And here it is in action:

```
UserSerializer(data={"username": "evernon", "first_name": "eve",
        "last_name": "vernon"}).is_valid(raise_exception=True)
```

Python returns two error messages; one saying the first name is not capitalized and another saying the last name is not capitalized.

```
------------------------------------------------------------
        -----------
ValidationError                           Traceback (most recent
        call last)
<ipython-input-4-4024482f87b8> in <module>()
----> 1 UserSerializer(data={"username": "evernon",
        "first_name": "eve", "last_name":
        "vernon"}).is_valid(raise_exception=True)

/home/codio/.local/lib/python3.6/site-
        packages/rest_framework/serializers.py in is_valid(self,
        raise_exception)
    226
    227         if self._errors and raise_exception:
--> 228             raise ValidationError(self.errors)
    229
    230         return not bool(self._errors)

ValidationError: {'first_name': [ErrorDetail(string='Value must
        be capitalized', code='invalid')], 'last_name':
        [ErrorDetail(string='Value must be capitalized',
        code='invalid')]}
```

Now, validation in a serializer is good, but it's not a substitute for validating at a lower level (like in a model). So any rules that you write to validate in a serializer should also exist (or perhaps should only exist) at a lower level like a model. We can use validators to prevent bad data coming in through the API, but a user might be able to get around these restrictions by using a form on the website, for example. If a serializer is tied to a model, then instead of setting the `max_length` on the serializer's `CharField`, then we should make sure it's set on the model's `CharField`. But how do you avoid doubling up all this validation code?

That brings us to our next topic, which is taking a look at the `ModelSerializer` class.

ModelSerializer

ModelSerializer

At the start of this topic we discussed the similarity between `Form` and `Serializer` and briefly mentioned `ModelSerializer`. This is a way to create a `Serializer` directly from a Django model, similar to creating a `Form` from a Django model with `ModelForm`.

All that needs to be done to create a `ModelSerializer` is to specify a `model` on its `Meta` attribute.

Here's how we would create one for a Blango `Post`:

```
from rest_framework import serializers
from blog.models import Post

class PostSerializer(serializers.ModelSerializer):
    class Meta:
        model = Post
        fields = "__all__"
```

Then it can be used like the other serializers:

```
p = Post.objects.first()
s = PostSerializer(p)
s.data
```

Python returns a serialized version of the first blog post (**note:** your data will look different).

```
ReturnDict([('id', 1),
            ('created_at', '2021-07-19T13:49:44.224000Z'),
            ('modified_at', '2021-07-19T13:49:44.224000Z'),
            ('published_at', '2021-07-19T13:48:18Z'),
            ('title', 'Super Happy Fun Title'),
            ('slug', 'super-happy-fun-title'),
            ('summary', 'Test post'),
            ('content', 'blah, blah, blah...'),
            ('author', 1),
            ('tags', [1])])
```

Of course we can customize the `fields` attribute to change the fields that are included in the serializer, just like with a `ModelForm`.

```
class PostSerializer(serializers.ModelSerializer):
    class Meta:
        model = Post
        fields = ["pk", "title", "slug", "summary"]
```

Now test it with the serializer from above. You should only see the primary key, title, slug, and summary.

```
s.data
```

Or we can do the inverse, and exclude fields using the `exclude` attribute.

```
class PostSerializer(serializers.ModelSerializer):
    class Meta:
        model = Post
        exclude = ["modified_at", "created_at"]
```

Test it again by passing the first `Post` object to the newly created `PostSerializer`. The output should not change.

```
p = Post.objects.first()
s = PostSerializer(p)
s.data
```

▼ **IDs**

You'll notice that attributes like `author` and `tags` come back with just their IDs. We will look at how to make these display a bit nicer in the next module.

The `ModelSerializer` implements the `create` and `update` methods for us, to write the data back to the database. Here's how we could make changes to a `Post`.

```
updater = PostSerializer(p, data={"title": "Leo: A Deeper
        Look"}, partial=True)
updater.is_valid()
```

Python returns `True` because the data is valid.

```
True
```

Now save the updated post.

```
updater.save()
```

Python returns the `Post` object.

```
<Post: Leo: A Deeper Look>
```

Validate the change by looking at the title of the first `Post` object.

```
Post.objects.first().title
```

You should see the updated title.

```
'Leo: A Deeper Look'
```

Note here that we're also introducing the `partial=True` argument to the constructor. This argument is also valid for normal serializers, and it means the serializer will only validate the values passed into the data and won't trigger an error if required fields are missing. It will instead just use the current value for any required fields; however this means that you can't create new instances. It also means that a custom `validate()` method might fail if you have not provided a field that's used in it.

Another useful thing we can do is override or add fields in the same way we would add fields to a normal serializer: by adding fields as attributes.

We might want to replicate the slug autogeneration that we get from Django admin, so that if a slug isn't provided in the body of the `Post` object, and the user provides the `autogenerate_slug` value of `True`, then we can generate the `slug` from the `title` using the `slugify` function.

The `PostSerializer` is defined like this.

```
from django.utils.text import slugify


class PostSerializer(serializers.ModelSerializer):
    slug = serializers.SlugField(required=False)
    autogenerate_slug = serializers.BooleanField(required=False,
        write_only=True, default=False)

    class Meta:
        model = Post
        exclude = ["modified_at", "created_at"]

    def validate(self, data):
        if not data.get("slug"):
            if data["autogenerate_slug"]:
                data["slug"] = slugify(data["title"])
            else:
                raise serializers.ValidationError("slug is
        required if autogenerate_slug is not set")
        del data["autogenerate_slug"]
        return data
```

Here we override the `slug` field and set it to not be `required`. Then we add a new, write-only field, called `autogenerate_slug`.

Our `validate` method checks if `slug` is not set/blank, and if so, checks if `autogenerate_slug` is `True`. If it is, then we'll generate a slug from the `title`. Otherwise we raise a `ValidationError`. At the end of the method we must delete `autogenerate_slug` from the data dictionary otherwise it will be passed to the `Post` constructor by the `create` method and cause an exception.

Here's how it behaves with invalid data:

```
post_data = {
    "title": "How to API",
    "author": 1,
    "summary": "How do you API? Here's how.",
    "content": "<p>REST APIs are crazy, and fun. Let's use one.
        </p>",
    "tags": [1]
}
s = PostSerializer(data=post_data)
s.is_valid(raise_exception=True)
```

Python raises an exception because the data is missing a slug.

```
---------------------------------------------------------------
        -----------
ValidationError                           Traceback (most recent
        call last)
<ipython-input-10-36a7f0bce20d> in <module>()
      7 }
      8 s = PostSerializer(data=post_data)
----> 9 s.is_valid(raise_exception=True)

/home/codio/.local/lib/python3.6/site-
        packages/rest_framework/serializers.py in is_valid(self,
        raise_exception)
    226
    227         if self._errors and raise_exception:
--> 228             raise ValidationError(self.errors)
    229
    230         return not bool(self._errors)

ValidationError: {'non_field_errors': [ErrorDetail(string='slug
        is required if autogenerate_slug is not set',
        code='invalid')]}
```

And if we set the `autogenerate_slug` to `True` then try again, it works:

```
post_data["autogenerate_slug"] = True
s = PostSerializer(data=post_data)
s.is_valid(raise_exception=True)
```

Python returns `True` as the data is valid.

```
True
```

Save the data and take a look at the slug.

```
p = s.save()
p.slug
```

Python returns the auto-generated slug based on the title.

```
'how-to-api'
```

The last configuration we'll look at in regards to `ModelSerializer` is how to mark fields as read-only. This is done with the `readonly` argument set to a list of field names:

```
class PostSerializer(serializers.ModelSerializer):
    class Meta:
        model = Post
        readonly = ["modified_at", "created_at"]
```

These fields will be included in the fetched data, but won't be able to be set.

You might have noticed that `author` and `tags` just show the ID/primary key of the related objects. In the next module 3 we'll examine nested relationships in more detail, and look at how to show some more useful data.

▼ **More information**
For more information on serializers, you can check out the DRF Serializer Documentation.

Try It Out

Try It Out

Let's implement a serializer in Blango, to use in the API views that we created. It will be `PostSerializer`, a `ModelSerializer` subclass.

Our code changes will start by creating a new directory in the `blog` app, called `api` that will store all of our API-related code going forward. Create this directory and the blank `__init__.py` file inside it.

Next, create a file in the `api` directory called `serializers.py`. This will hold our serializer classes.

Open serializers.py

Inside this file, import the `serializers` module, and `Post` model:

```
from rest_framework import serializers
from blog.models import Post
```

Then define your `PostSerializer` class, a subclass of `serializers.ModelSerializer`. We'll keep it simple and just make `modified_at` and `created_at` readonly. Something like this:

```
class PostSerializer(serializers.ModelSerializer):
    class Meta:
        model = Post
        fields = "__all__"
        readonly = ["modified_at", "created_at"]
```

That's all for `serializers.py`, now switch to `api_views.py` in `blog`. In the future we'll put API views inside the `api` directory, but since the API views will be going away soon we'll leave them for now.

Open api_views.py

The first thing we can do is import our serializer class at the top of the file:

```
from blog.api.serializers import PostSerializer
```

Then delete the `post_to_dict()` function, as we'll be able to use our serializer to do the same thing.

Then, we need to make four replacements in our views. The `GET` and `POST` branches in the `post_list` view, and the `GET` and `PUT` branches in the `post_detail` view. Let's go through them.

In the list `GET`, change the code to this:

```
posts = Post.objects.all()
return JsonResponse({"data": PostSerializer(posts,
        many=True).data})
```

We just pass in the `Post` queryset and provide the `many=True` argument, and our serializer does the rest.

In the list `POST`, load the `post_data` in the same way, and don't change the return value. But change how the `Post` is created to this:

```
serializer = PostSerializer(data=post_data)
serializer.is_valid(raise_exception=True)
post = serializer.save()
```

Now we will change the `post_detail` function. If the method is `GET`, return the `JsonResponse` like this:

```
return JsonResponse(PostSerializer(post).data)
```

Finally, if the method is `PUT`, we load the JSON data the same way, and return the same response, but the `Post` saving updating and saving code should be updated to this:

```
serializer = PostSerializer(post, data=post_data)
serializer.is_valid(raise_exception=True)
serializer.save()
```

The final file we need to change is `settings.py`, to add `rest_framework` to our list of `INSTALLED_APPS`. This is not strictly necessary for us to do right now, but we will need to make sure this is set for future functionality.

Open settings.py

Now, you can test the API, using Postman, or a similar tool. The dev server must be running to use Postman. In addition, your Codio host has changed. You need to update the `baseurl` in Postman with your new Codio host.

▼ **Finding your Codio host**
In the Codio menu bar, click "Project" => "Box Info". This will open a tab with your Codio host name in it.

The only differences you should notice are:

- `author_id` is now `author` in the `Post` body.
- `tags` must be a non-empty list on the `Post` body.
- For a `PUT`, you must provide the entire `Post` as we don't support partial updates any more. This brings us more in line with what REST prescribes for a `PUT` request.

We have simplified our API views and made them a bit more robust and with better validation on failure. But serializers are just the first part of Django Rest Framework that we'll look at. In the next section we'll check out DRF views and how they can reduce the amount of code we have to write even more.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish Serializers"
```

- Push to GitHub:

```
git push
```