Django REST Framework

Friday Presentation

Shrijan Subedi, November 3rd, 2020

Web API

Introduction

- A web API is a interface consisting of one or more publicly exposed endpoints to a serve some data or receive data from a client program.
- We use a standardized data format such as XML or JSON for communication between the client and server.
- A web API is exposed via the web—most commonly by means of an HTTP-based web server.
- Some architectural styles for creating an API are REST, SOAP

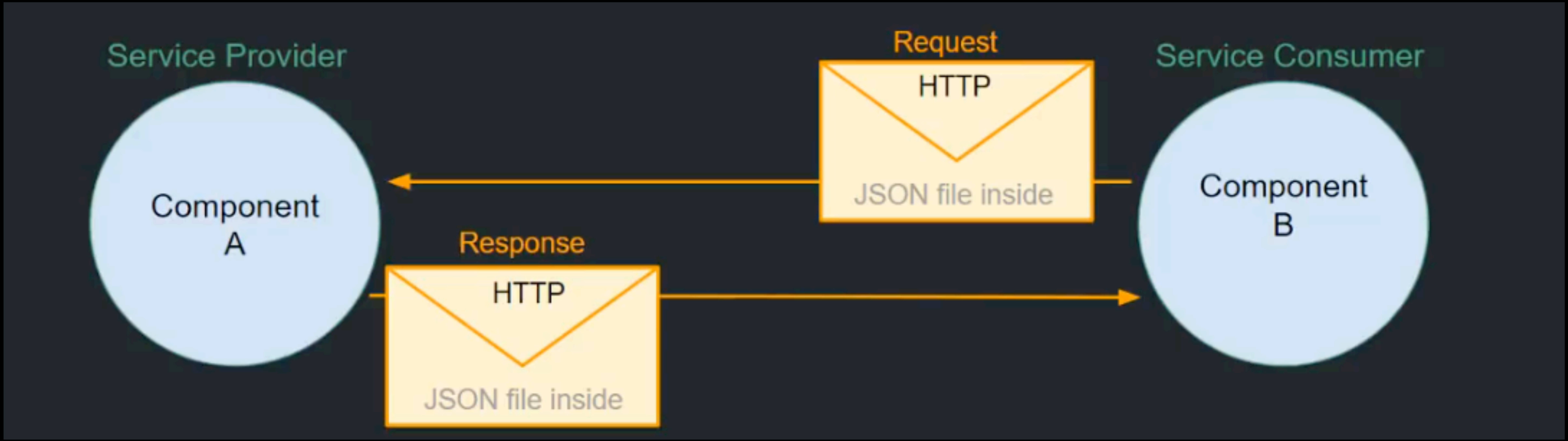
REST API

Introduction

- REST is acronym for REpresentational State Transfer. It is architectural style for distributed hypermedia systems and was first presented by Roy Fielding in 2000 in his famous dissertation.
- REST defines a set of constraints to be used for creating Web services.
- Any API that follows the REST guidelines is referred to as a RESTful API
- RESTful Web services allow the requesting systems to access and manipulate textual representations of Web resources by using a uniform and predefined set of stateless operations.

REST API

Example



REST API

Guiding Principles of REST

- REST has 6 guiding constraints which must be satisfied if an interface needs to be referred as RESTful:

1. Client–server: Separate UI concerns from data storage concerns to improve portability and scalability.

2. Stateless: Each request from client to server must contain all of the information necessary to understand the request and cannot take advantage of any stored context on the server.

3. Cacheable: Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable.

REST API

Guiding Principles of REST

4. Uniform interface: REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.

5. Layered system: The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot “see” beyond the immediate layer with which they are interacting.

6. Code on demand (optional): REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts

REST API

Resources

- The key abstraction of information in REST is a resource.
- Any information that can be named can be a resource: a document or image, a temporal service, a collection of other resources etc.
- REST uses a resource identifier to identify the particular resource involved in an interaction between components.
- The state of the resource at any particular timestamp is known as resource representation.
- A representation consists of data, metadata describing the data and hypermedia links which can help the clients in transition to the next desired state.

REST API
Resource Methods

- REST resource methods are used to perform the desired transition to change a resource state.
- While using HTTP, we may use HTTP methods to retrieve data or invoke operations. The methods may be:
  - GET : retrieve a resource
  - POST: create a new resource
  - PUT: update an existing resource
  - DELETE: delete an resource
- The GET method is safe. The GET, PUT and DELETE methods are idempotent

REST API

Advantages of RESTFul Systems:

1. REST APIs assist in the separation of web application modules most notably in the form of Frontend and Backend. Frontend and Backend applications can communicate using RESTFul APIs.

2. By using a stateless protocol and standard operations, RESTful systems aim for fast performance, reliability, and the ability to grow by reusing components that can be managed and updated without affecting the system as a whole, even while it is running.

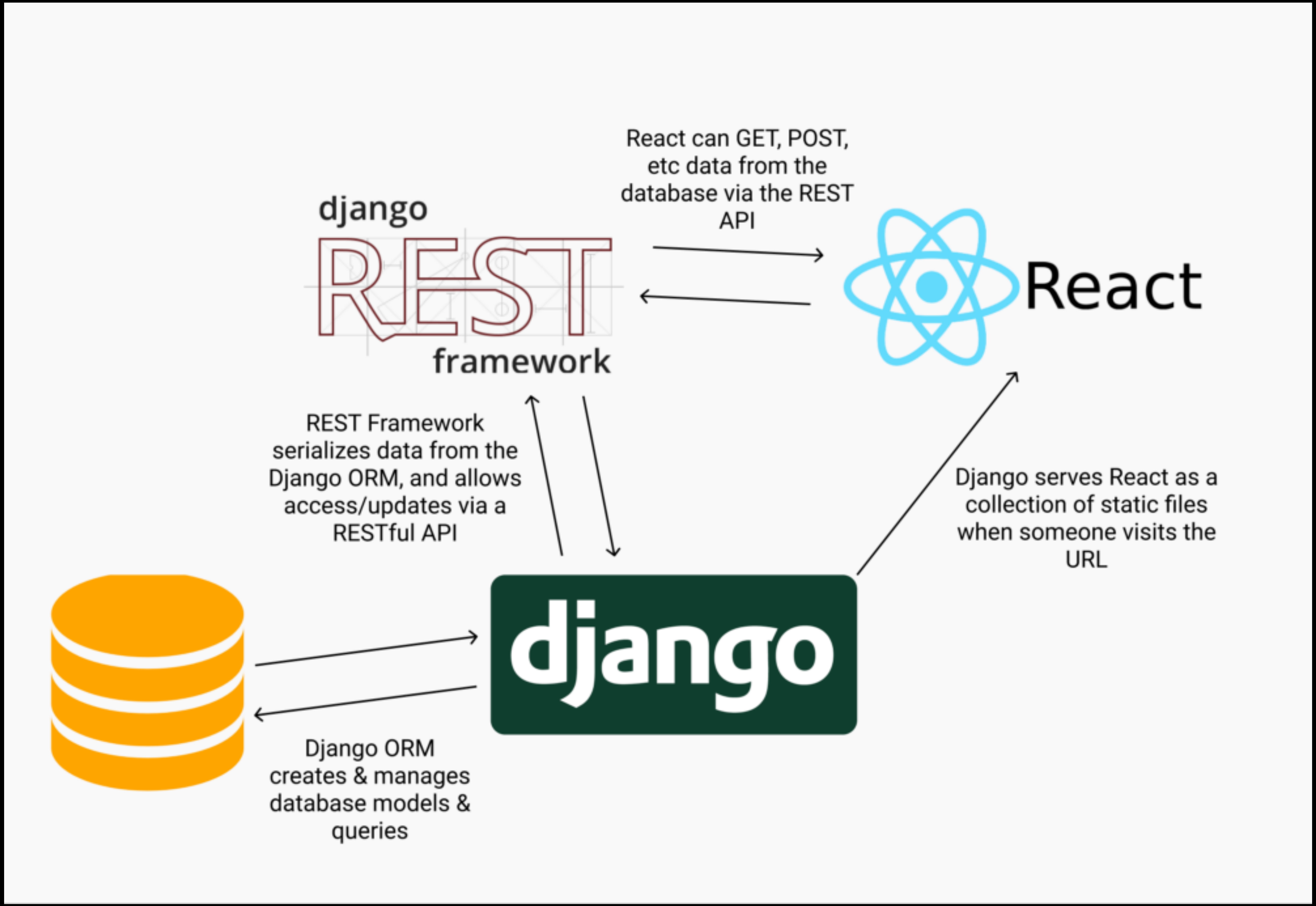
Django REST framework

Introduction

- Django REST framework is a powerful and flexible toolkit for building Web APIs.
- Features:
  - Web browsable API
  - Authentication Policies including OAuth1 and OAuth2
  - Serialization support for both ORM and non-ORM sources
  - Customizable for variety of use cases
  - Extensive documentation and support

Django REST framework

Introduction


React can GET, POST, etc data from the database via the REST API
django REST framework
React
REST Framework serializes data from the Django ORM, and allows access/updates via a RESTful API
Django serves React as a collection of static files when someone visits the URL
django
Django ORM creates & manages database models & queries

Django REST framework

Getting Started

Installation

```
$ pip install djangorestframework
```

Add 'rest_framework' to your INSTALLED_APPS setting.

```
INSTALLED_APPS = [
    ...
    'rest_framework',
]
```

Django REST framework

Getting Started

To be able to use browsable API we need to add REST framework’s login and logout view in our urls.py

```
urlpatterns = [
    ...
    path('api-auth/', include('rest_framework.urls'))
]
```

Django REST framework

Configuration

Any global settings for a REST framework API are kept in a single configuration dictionary named REST_FRAMEWORK:

```
REST_FRAMEWORK = {

    'DEFAULT_PERMISSION_CLASSES': [

        'rest_framework.permissions.DjangoModelPermissionsOrAnonReadOnly'

    ],

    …..

}
```

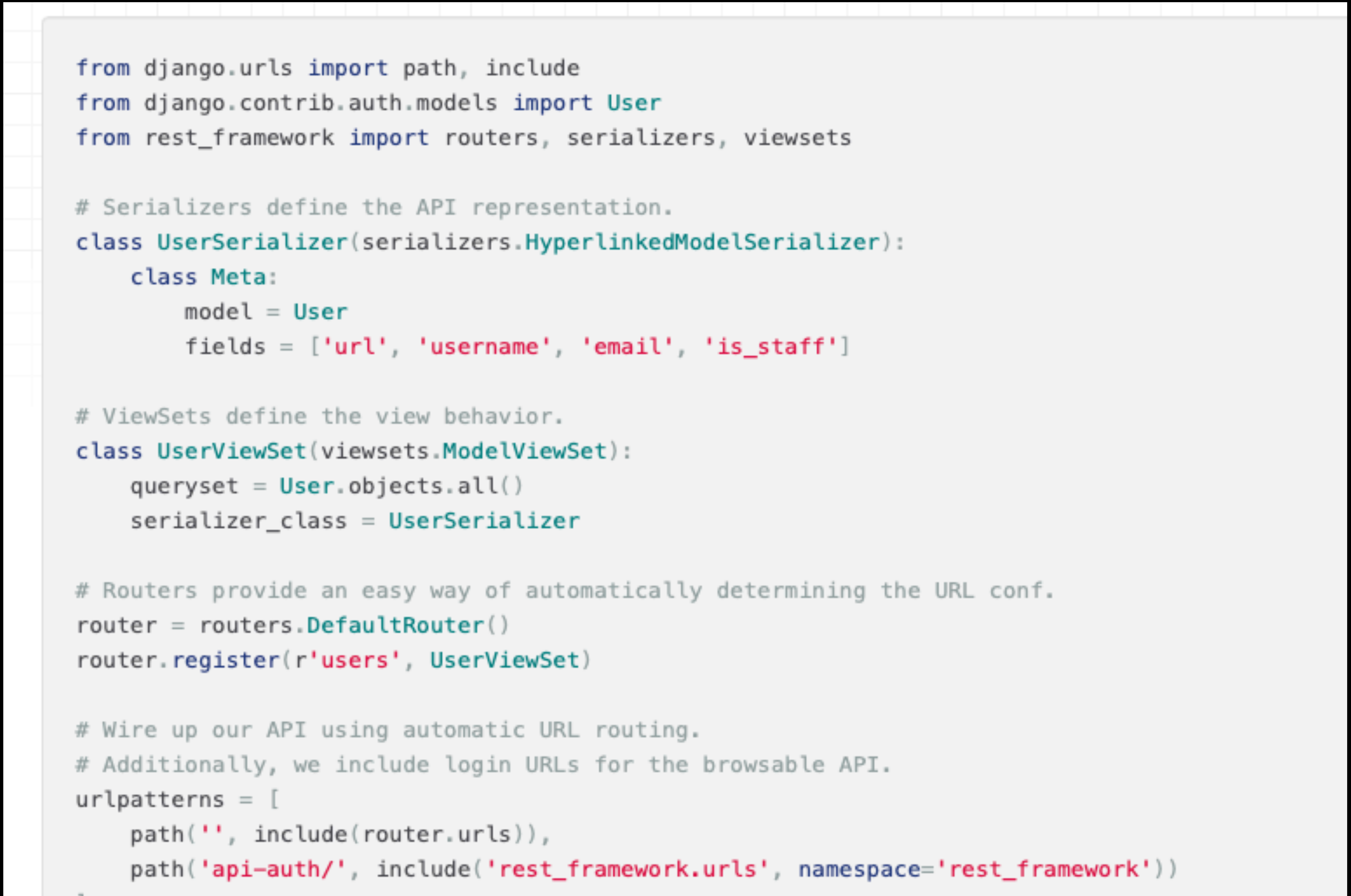
Django REST framework

Quick Example

- Basic steps to create an API endpoint:
    - Create a model for the resource
    - Create a serializer for the model and specify fields to serialize
    - Define Views/ViewSets that define the endpoint behavior
    - Map the URL to the View or instead use Routers for automatically determining the URL conf.

Django REST framework

Quick Example



```
from django.urls import path, include
from django.contrib.auth.models import User
from rest_framework import routers, serializers, viewsets

# Serializers define the API representation.
class UserSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = User
        fields = ['url', 'username', 'email', 'is_staff']

# ViewSets define the view behavior.
class UserViewSet(viewsets.ModelViewSet):
    queryset = User.objects.all()
    serializer_class = UserSerializer

# Routers provide an easy way of automatically determining the URL conf.
router = routers.DefaultRouter()
router.register(r'users', UserViewSet)

# Wire up our API using automatic URL routing.
# Additionally, we include login URLs for the browsable API.
urlpatterns = [
    path('', include(router.urls)),
    path('api-auth/', include('rest_framework.urls', namespace='rest_framework'))
```

Django REST Framework

Testing the API

- We can use the browsable API to access the resources:

```
python manage.py runserver
```

  http://127.0.0.1:8000

- Or we may use curl or httpie to send requests to the endpoint

Django REST framework

Request Objects

- REST framework's Request class extends the standard HttpRequest,
- It adds support for REST framework's flexible request parsing and request authentication.
- Request Parsing: REST framework's Request objects provide flexible request parsing that allows to treat requests with JSON data or other media types in the same way that you would normally deal with form data.
  - request.data: returns the parsed content of the request body.
  - request..query_params: returns the query parameters of a request

Django REST framework

Request Objects

- Authentication: REST framework provides both user and token information associated with the incoming request.
    - request.user: returns instance of Django.contrib.auth.models.User/Anonymous according to the authentication policy
    - request.auth: returns any additional authentication context. None if no context is present.
- Browser enhancements:
    - request.method: return request's HTTP method
    - request.content_type: return content type of request body

Django REST framework

Response Objects

- REST framework supports HTTP content negotiation by providing a Response class which allows to return content that can be rendered into multiple content types, depending on the client request.
- We may use HttpResponse but Response class simply provides a nicer interface for returning content-negotiated Web API responses.
- Creating Responses:
  - Response( ): Response(data, status=None, template_name=None, headers=None, content_type=None)
- Unlike regular HttpResponse objects, we do not instantiate Response objects with rendered content. Instead we pass in unrendered data, which may consist of any Python primitives.

Django REST framework

Response Objects

- The renderers used by the Response class cannot natively handle complex datatypes such as Django model instances.
- So we need to serialize the data into primitive datatypes before creating the Response object.
- Arguments:
    - data: The serialized data for the response.
    - status: A status code for the response. Defaults to 200.
    - template_name: A template name to use if HTMLRenderer is selected.
    - headers: A dictionary of HTTP headers to use in the response.
    - content_type: The content type of the response.

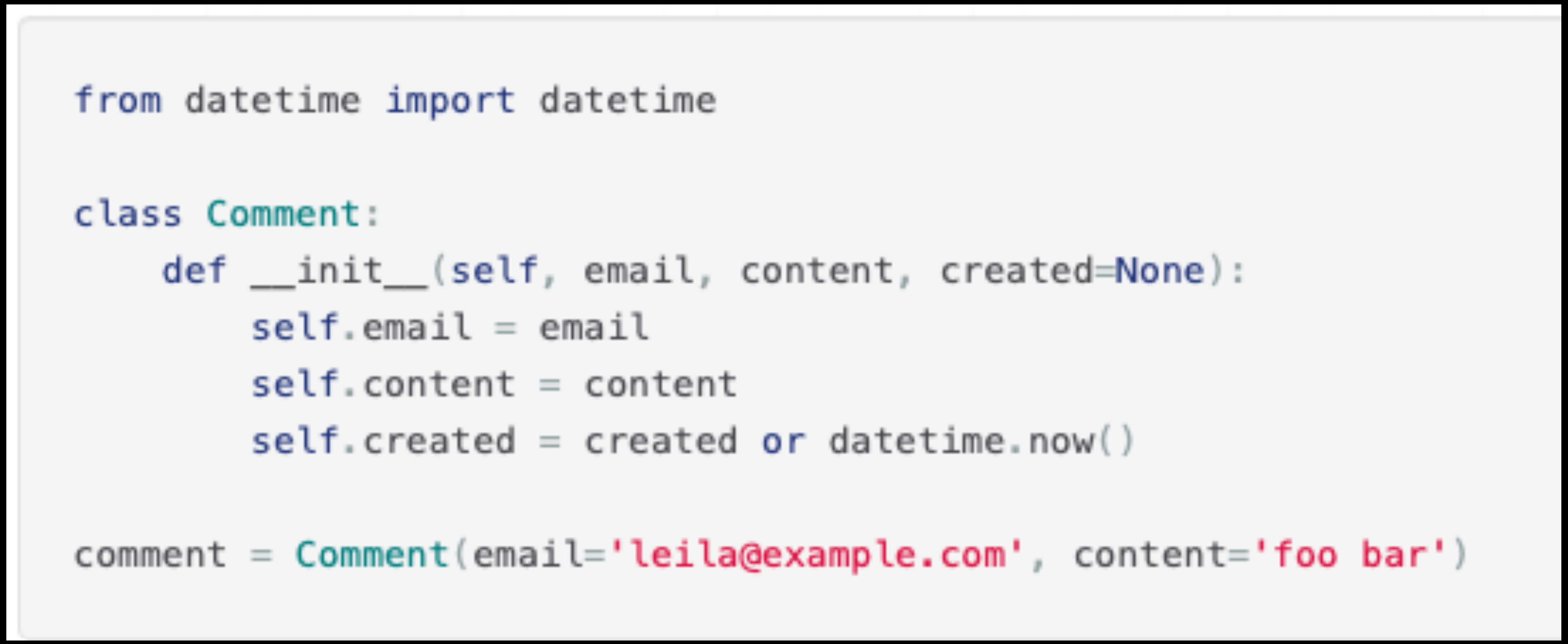
Django REST framework

Serialization

- 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.
- We inherit the Serializer class to create our serializers. We may also use ModelSerializer for generic functionality.
- Our serializers may reside in serializers.py module in the app folder

Django REST framework

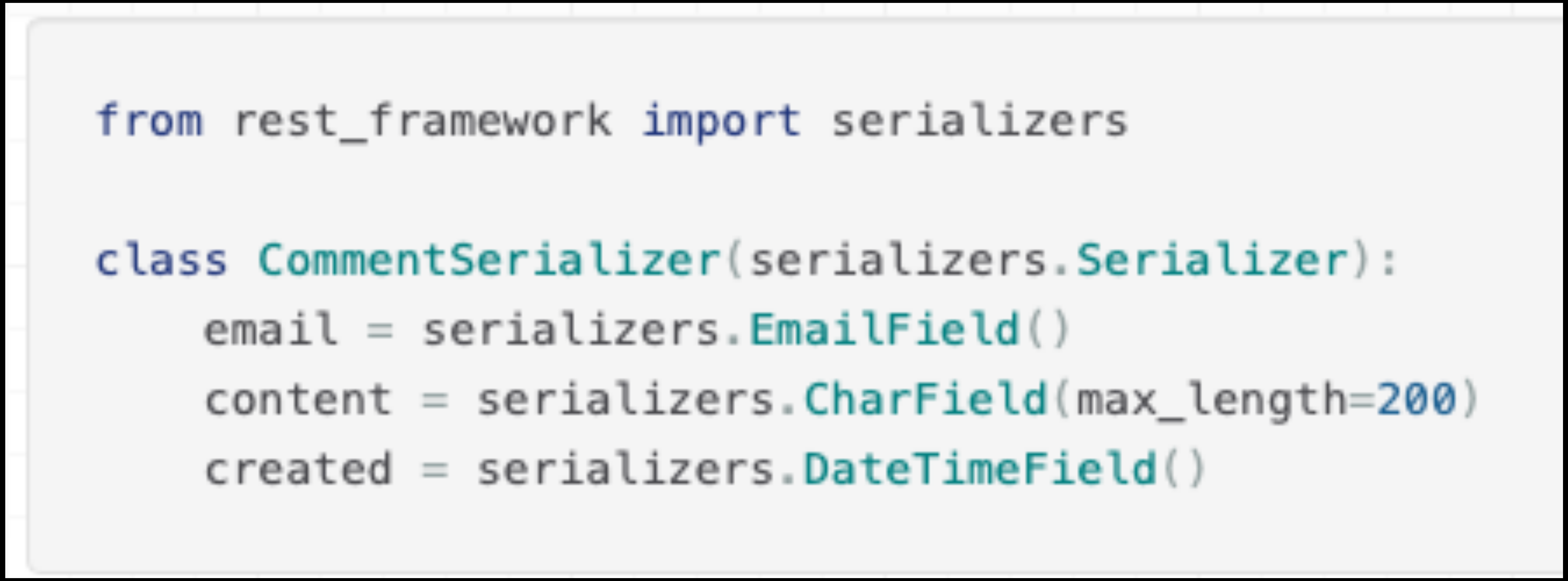
Serialization: Declaring Serializers



```python
from datetime import datetime

class Comment:
    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')
```

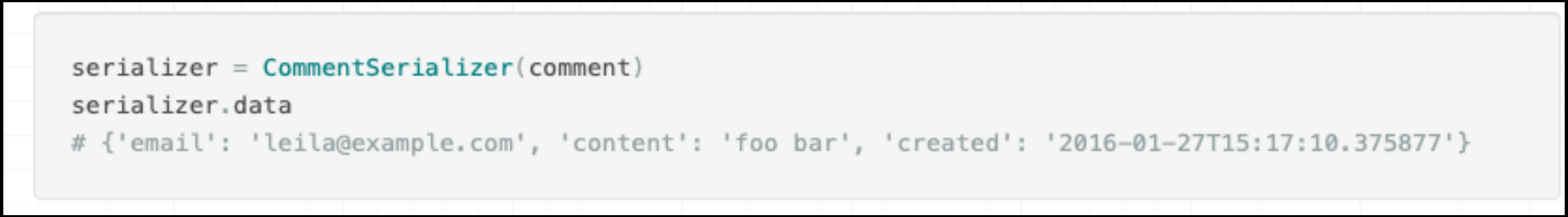


```python
from rest_framework import serializers

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

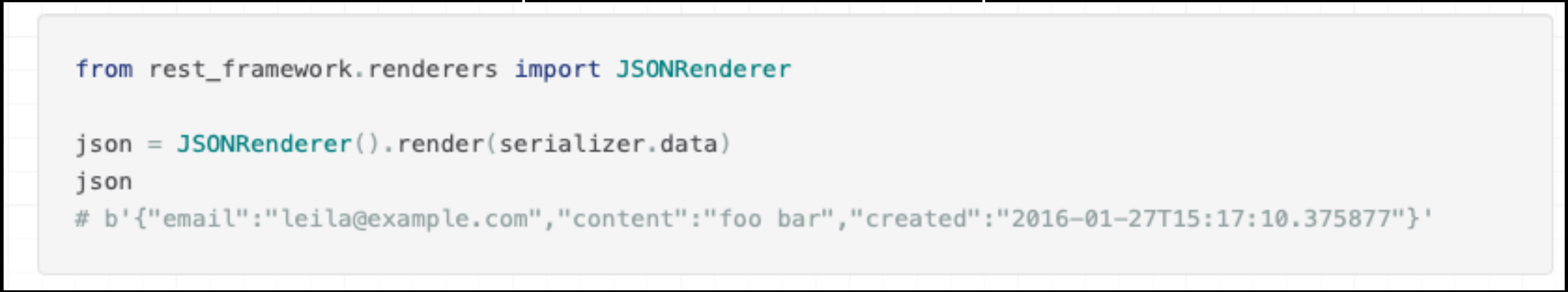
Django REST framework

Serialization: Serializing and Deserializing Objects



```
serializer = CommentSerializer(comment)
serializer.data
# {'email': 'leila@example.com', 'content': 'foo bar', 'created': '2016-01-27T15:17:10.375877'}
```

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

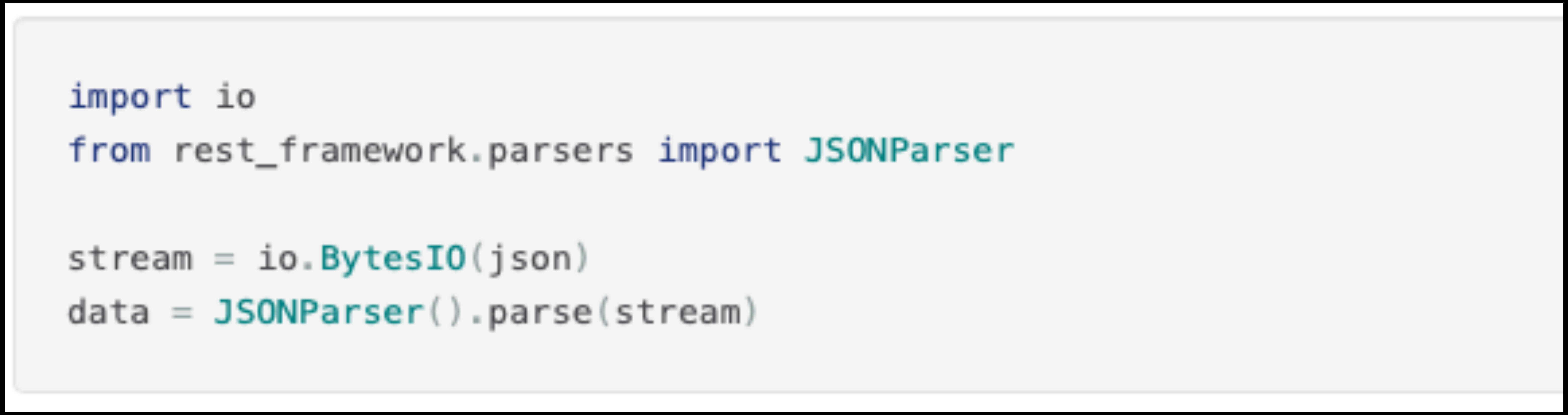


```
from rest_framework.renderers import JSONRenderer

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

Django REST framework

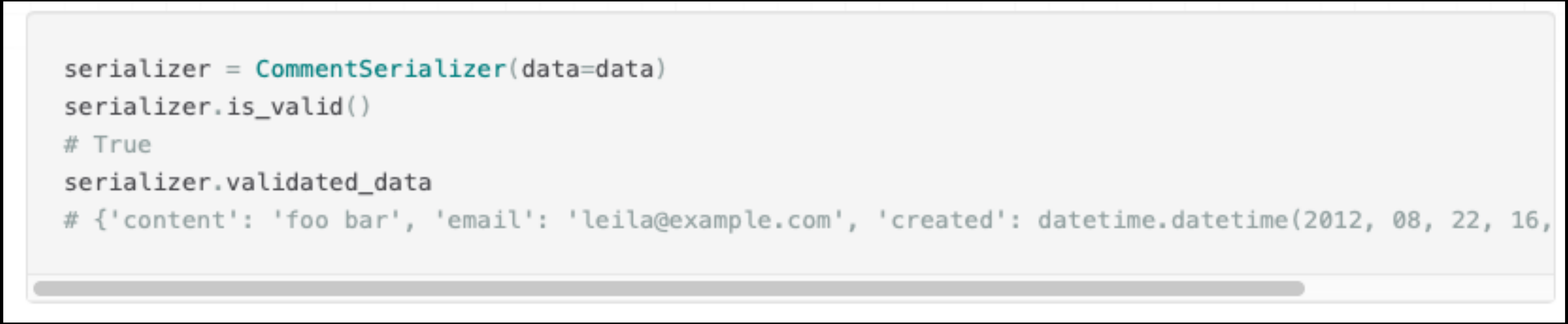
Serialization: Serializing and Deserializing Objects



```
import io
from rest_framework.parsers import JSONParser

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

First we parse a stream into Python Native datatypes and 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, 08, 22, 16,
```

Django REST framework

Serialization: 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. ->Code.

- If your object instances correspond to Django models you'll also want to ensure that these methods save the object to the database.

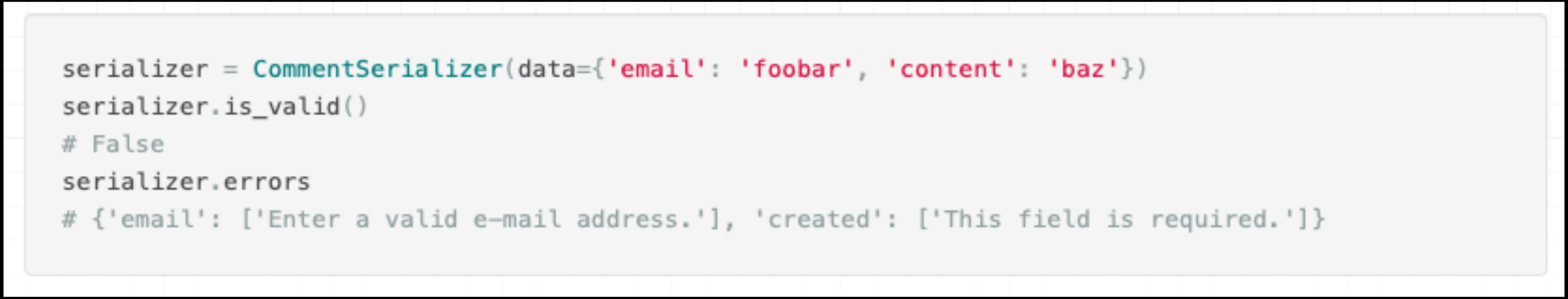
Comment.objects.create(**validated_data) in Create

instance.save() in Update

Django REST framework

Serialization: Validation

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



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

Django REST framework

Serialization: Validation

- Field-level validation:

  You can specify custom field-level validation by adding .validate_<field_name> methods to your Serializer subclass.

  Validate_<field_name> methods should return the validated value or raise a serializers.ValidationError

- 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 dictionary of field values.

Django REST framework

Serialization: Nested Objects

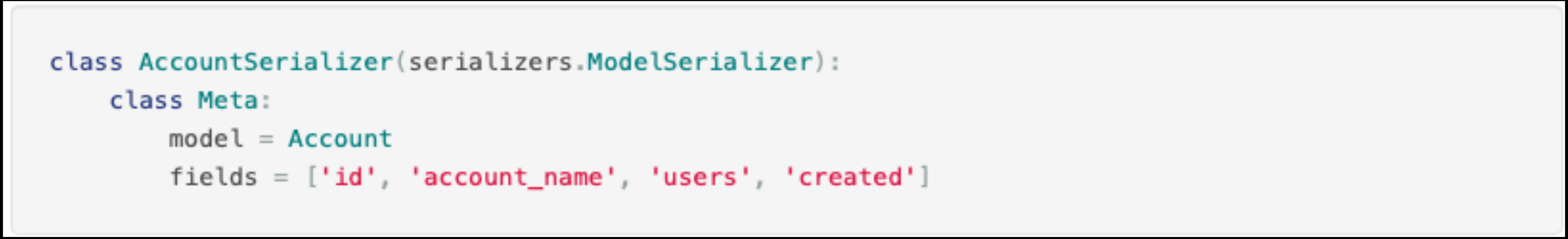
- Serializer class is itself a type of Field, and can be used to represent relationships where one object type is nested inside another.
- if a nested representation should be a list of items, you should pass the many=True flag to the nested serialized.
- If a nested representation may optionally accept the None value you should pass the required=False flag to the nested serializer.

```python
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()
```

Django REST framework

Serialization: ModelSerializer

- The ModelSerializer class provides a shortcut that lets you automatically create a Serializer class with fields that correspond to the Model fields.
- ModelSerializer automatically provides these features:
  - 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().



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

Django REST framework

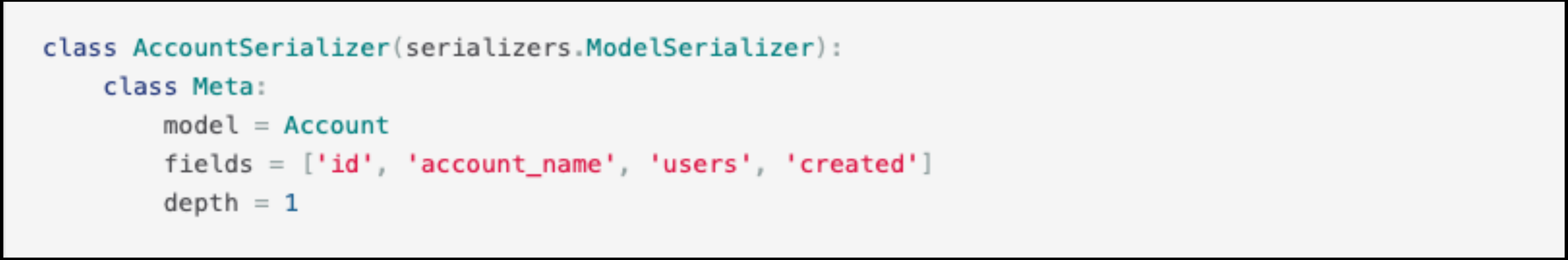
Serialization: ModelSerializer

- Any relationships such as foreign keys on the model will be mapped to PrimaryKeyRelatedField.
- By default, all the model fields on the class will be mapped to a corresponding serializer fields.
- To specify the fields to use we set the fields attributes as:
  - fields = ['id', 'account_name', 'users', ‘created’] : for specific fields
  - fields =‘__all__’ : for all fields
  - exclude = [‘users'] : to exclude a field

Django REST framework

Serialization: ModelSerializer Nested Serialization

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



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

Django REST framework

Serialization: ModelSerializer Fields Modification

- We can add extra fields to a ModelSerializer or override the default fields by declaring fields on the class. -> Code
- We may use the shortcut Meta option, read_only_fields=[‘ ’] to specify fields that we need to be read only.
- Relational Fields:
  - 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.

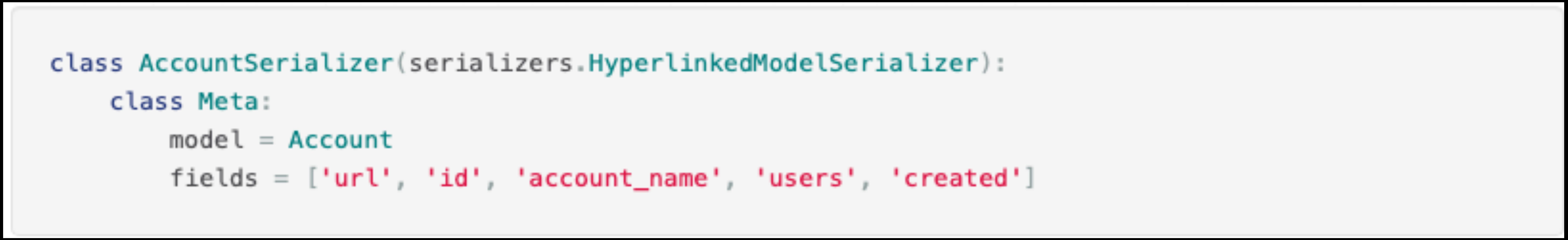
Django REST framework

Serialization: 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.



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

Django REST framework

Serialization: HyperlinkedModelSerializer Hyperlinked Views

- 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.
- Alternatively you can set the fields on the serializer explicitly. ->Code

```python
class AccountSerializer(serializers.HyperlinkedModelSerializer):
    url = serializers.HyperlinkedIdentityField(
        view_name='accounts',
        lookup_field='slug'
    )
```

Django REST framework

Authentication

- Authentication is the mechanism of associating an incoming request with a set of identifying credentials, such as the user the request came from, or the token that it was signed with.
- The permissions can then use those credentials to determine if the request should be permitted.
- REST framework provides a number of authentication schemes out of the box, and also allows you to implement custom schemes.
- The request.user property will typically be set to an instance of the contrib.auth package's User class.
- The request.auth property is used for any additional authentication information, for example, it may be used to represent an authentication token that the request was signed with.
- Note: Authentication by itself won't allow or disallow an incoming request, it simply identifies the credentials that the request was made with.

Django REST framework

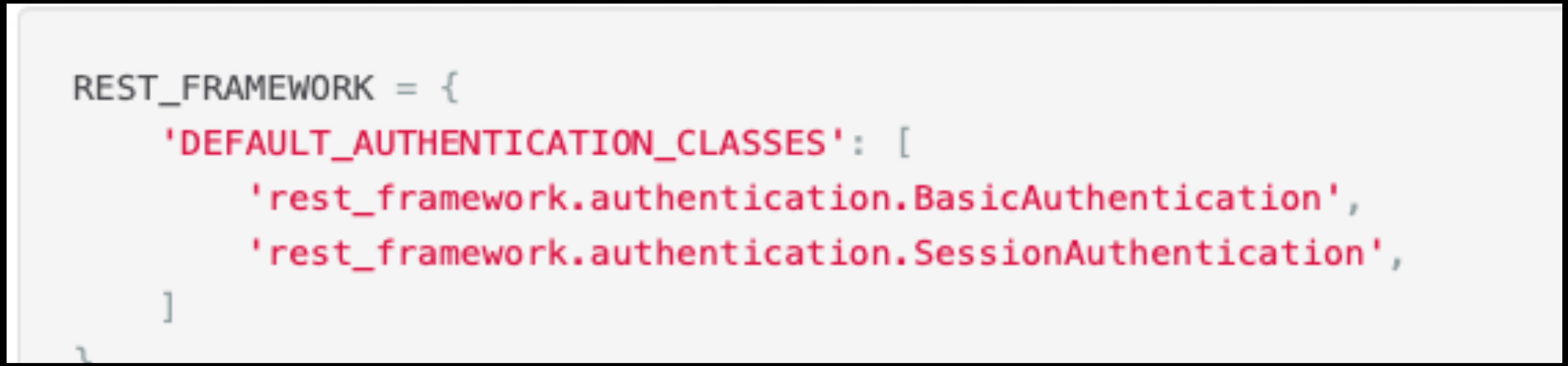
Authentication: Setting Authenticatin Schemes

- The authentication schemes are always defined as a list of classes.
- REST framework will attempt to authenticate with each class in the list, and will set request.user and request.auth using the return value of the first class that successfully authenticates.
- If no class authenticates, request.user will be set to an instance of django.contrib.auth.models.AnonymousUser, and request.auth will be set to None.
- The value of request.user and request.auth for unauthenticated requests can be modified using the UNAUTHENTICATED_USER and UNAUTHENTICATED_TOKEN settings.

Django REST framework

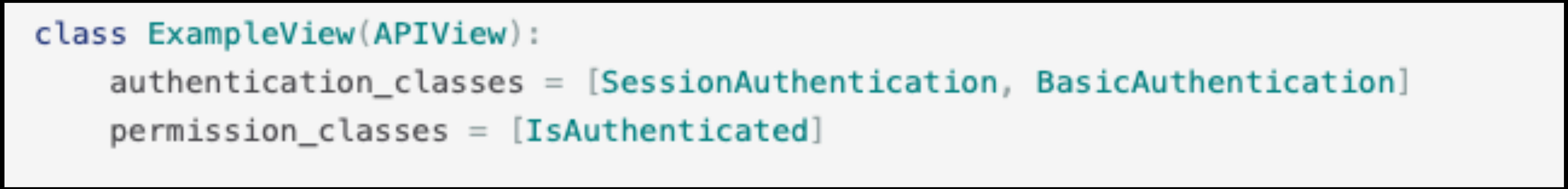
Authentication: Setting Authenticatin Schemes

- The default authentication schemes may be set globally, using the DEFAULT_AUTHENTICATION_CLASSES setting. For example:



```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.BasicAuthentication',
        'rest_framework.authentication.SessionAuthentication',
    ]
}
```

- You can also set the authentication scheme on a per-view or per-viewset basis, using the APIView class-based views.



```
class ExampleView(APIView):
    authentication_classes = [SessionAuthentication, BasicAuthentication]
    permission_classes = [IsAuthenticated]
```

Django REST framework

Authentication: Unauthorized and Forbidden responses

- When an unauthenticated request is denied permission there are two different error codes that may be appropriate.
- HTTP 401 Unauthorized: The request requires user authentication. It must always include a WWW-Authenticate header, that instructs the client how to authenticate.
- HTTP 403 Permission Denied: The server understood the request, but is refusing to fulfill it. Denied permission to perform the request.

Django REST framework

Authentication: BasicAuthentication

- This authentication scheme uses HTTP Basic Authentication, signed against a user's username and password.
- If successfully authenticated, BasicAuthentication provides the following credentials.
  - request.user will be a Django User instance.
  - request.auth will be None.
- Unauthenticated responses that are denied permission will result in an HTTP 401 Unauthorized response with an appropriate WWW-Authenticate header.
- Note: If we use BasicAuthentication in production we must ensure that our API is only available over https. We should also ensure that our API clients will always re-request the username and password at login, and will never store those details to persistent storage.

Django REST framework

Authentication: TokenAuthentication

- This authentication scheme uses a simple token-based HTTP Authentication scheme.
- Token authentication is appropriate for client-server setups, such as native desktop and mobile clients.
- To use the TokenAuthentication scheme you'll need to configure the authentication classes to include TokenAuthentication, and additionally include rest_framework.authtoken in your INSTALLED_APPS setting.
- Note: Make sure to run manage.py migrate after changing your settings. The rest_framework.authtoken app provides Django database migrations.

Django REST framework

Authentication: TokenAuthentication

- We need to create tokens for Users: We may use signals or expose an api endpoint to generate tokens for the user

```
from rest_framework.authtoken.models import Token

token = Token.objects.create(user=...)
print(token.key)
```

- For clients to authenticate, the token key should be included in the Authorization HTTP header. ->Code

```
Authorization: Token 9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b
```

Django REST framework

Authentication: TokenAuthentication

- If successfully authenticated, TokenAuthentication provides the following credentials.
  - request.user will be a Django User instance.
  - request.auth will be a rest_framework.authtoken.models.Token instance
- Unauthenticated responses that are denied permission will result in an HTTP 401 Unauthorized response with an appropriate WWW-Authenticate header. For example:
- Note: If you use TokenAuthentication in production you must ensure that your API is only available over https.

Django REST framework

Authentication: SessionAuthentication

- This authentication scheme uses Django's default session backend for authentication.
- Session authentication is appropriate for AJAX clients that are running in the same session context as your website.
- If successfully authenticated, SessionAuthentication provides the following credentials.
  - request.user will be a Django User instance.
  - request.auth will be None.
- Unauthenticated responses that are denied permission will result in an HTTP 403 Forbidden response.
- If you're using an AJAX style API with SessionAuthentication, you'll need to make sure you include a valid CSRF token for any "unsafe" HTTP method calls

Django REST framework

Authentication: JWT Authentication

- JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object
- JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.
- Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token.

Django REST framework

Authentication: JWT Authentication

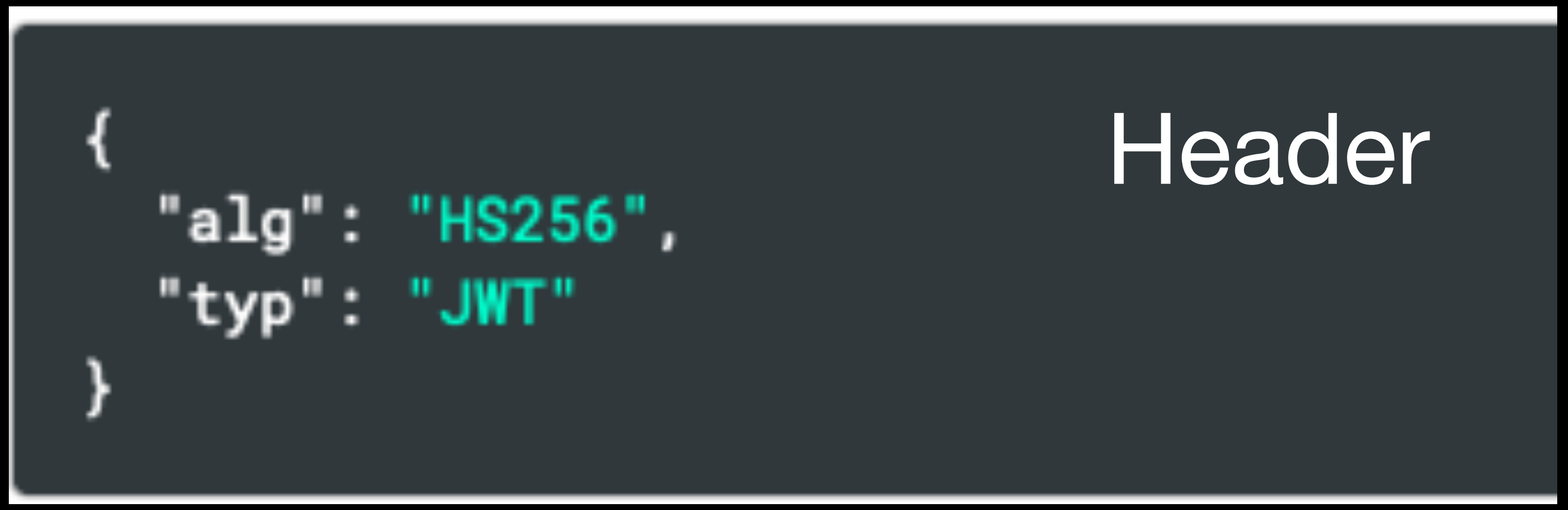
- In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are: Header, Payload and Signature in the following structure:

  xxxxx.yyyyy.zzzzz

  - Headers: The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.
  - Payload: The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data
  - Signature: The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

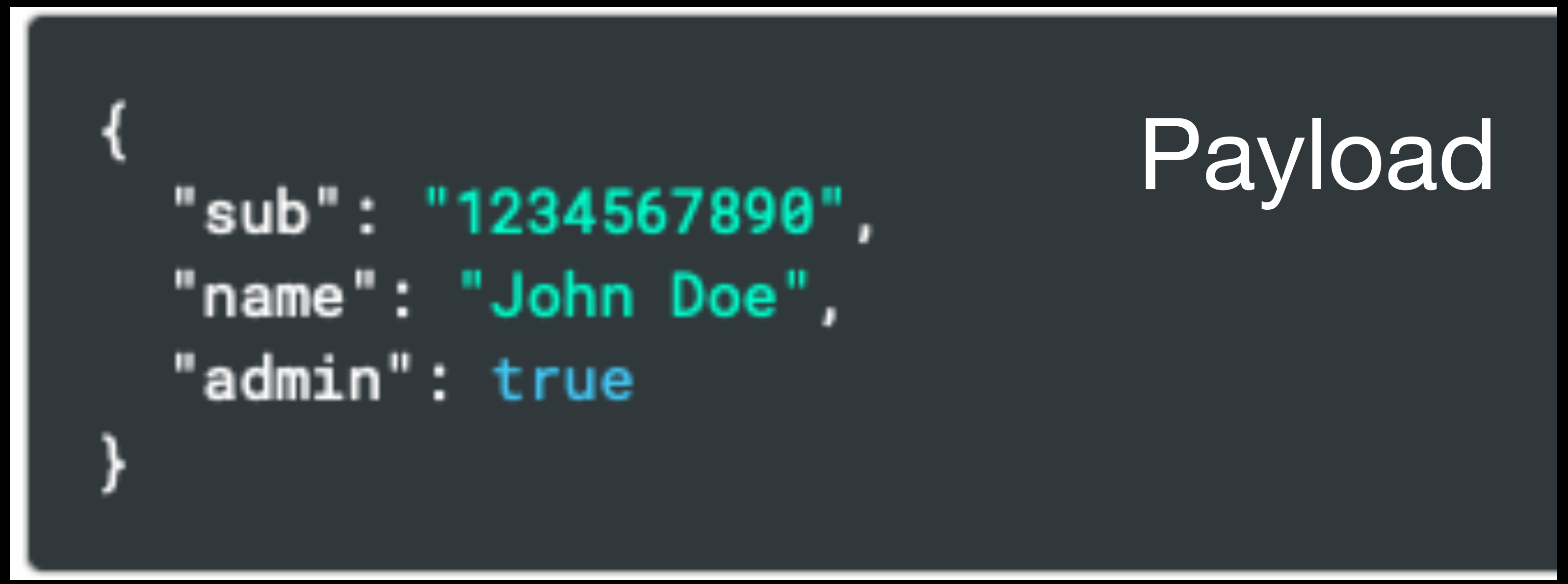
Django REST framework

Authentication: JWT Authentication



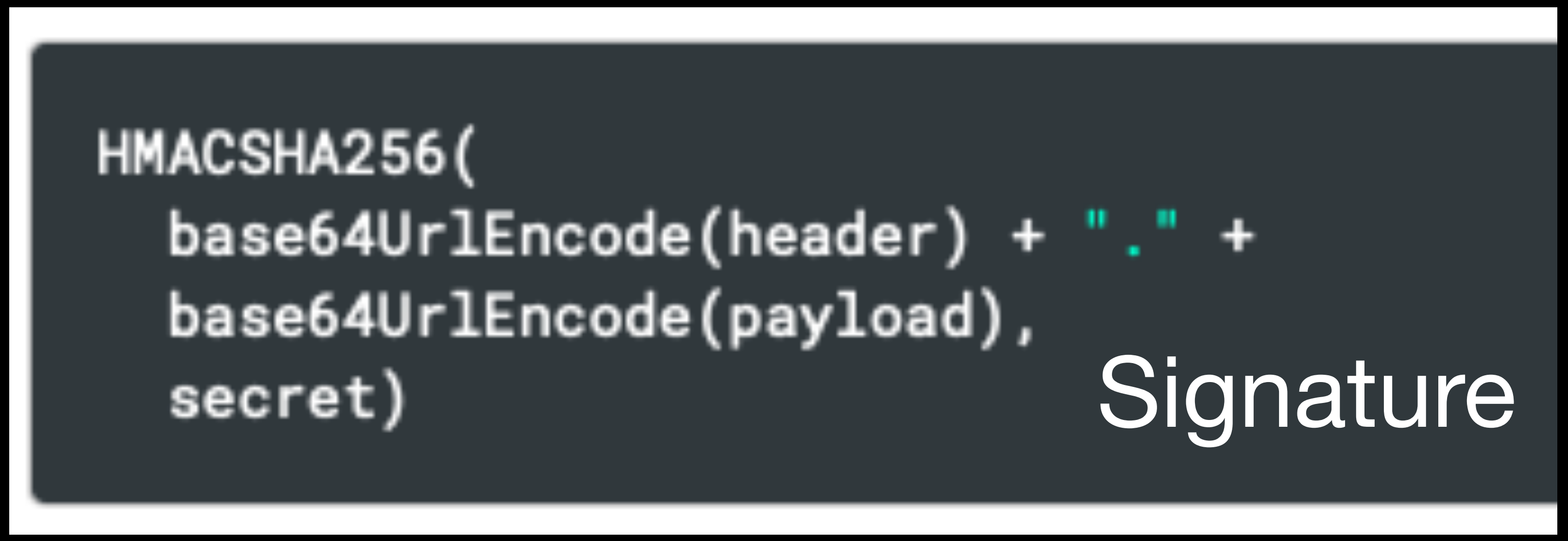
Header

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```



Payload

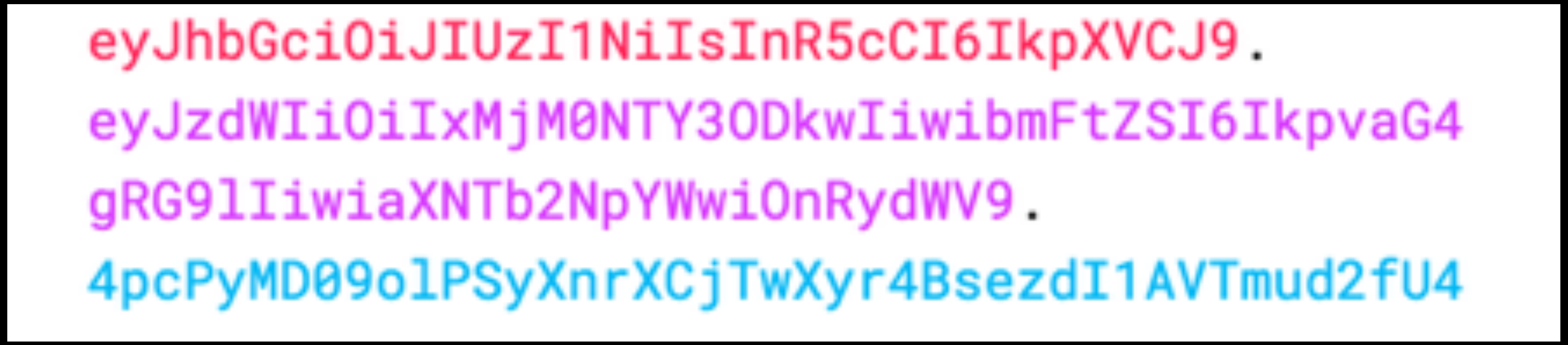
```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```



Signature

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Encoded and Signed →



```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

Django REST framework

Authentication: JWT Authentication Working

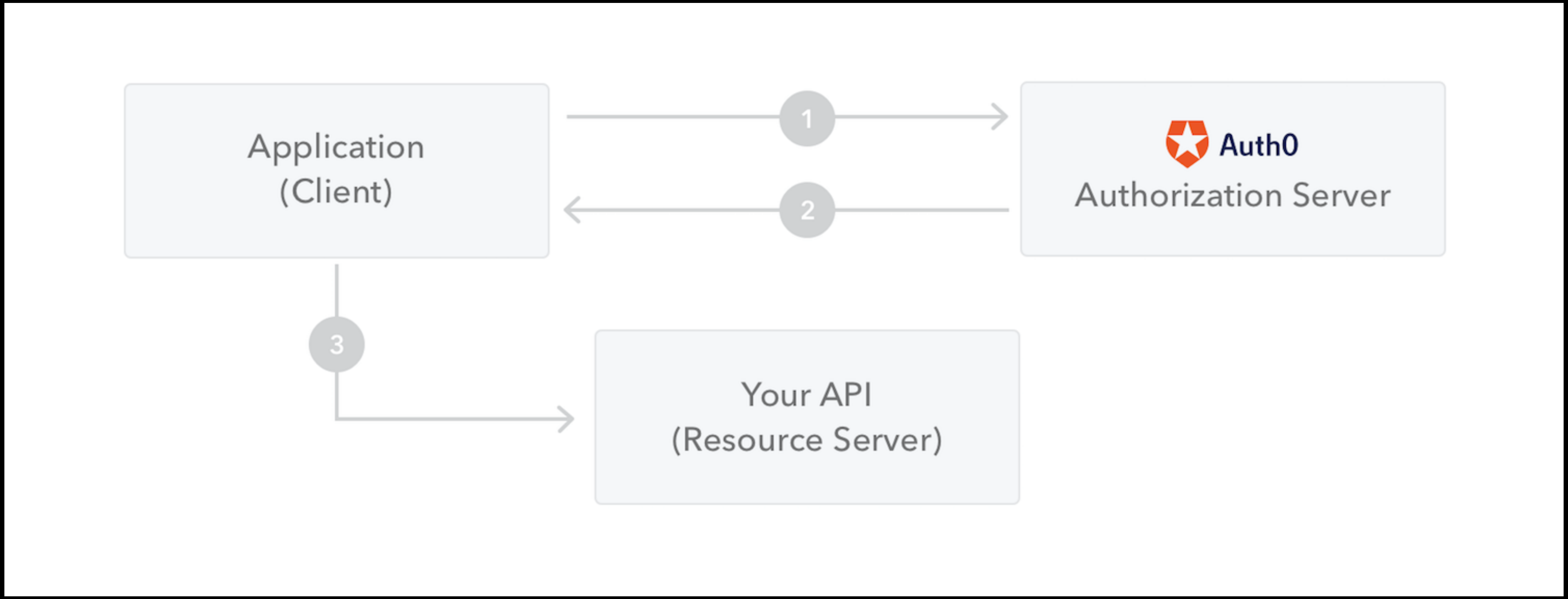
- Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the Authorization header using the Bearer schema.

- The server's protected routes will check for a valid JWT in the Authorization header, and if it's present, the user will be allowed to access protected resources.

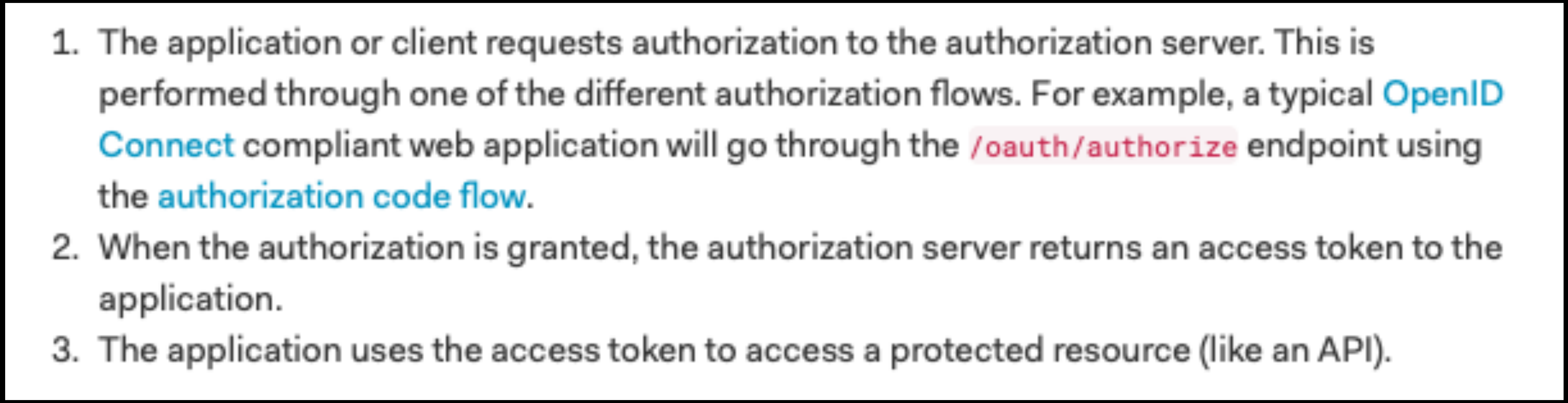
- If the token is sent in the Authorization header, Cross-Origin Resource Sharing (CORS) won't be an issue as it doesn't use cookies.

Django REST framework

Authentication: JWT Authentication Working



•



1. The application or client requests authorization to the authorization server. This is performed through one of the different authorization flows. For example, a typical OpenID Connect compliant web application will go through the `/oauth/authorize` endpoint using the authorization code flow.
2. When the authorization is granted, the authorization server returns an access token to the application.
3. The application uses the access token to access a protected resource (like an API).

Django REST framework

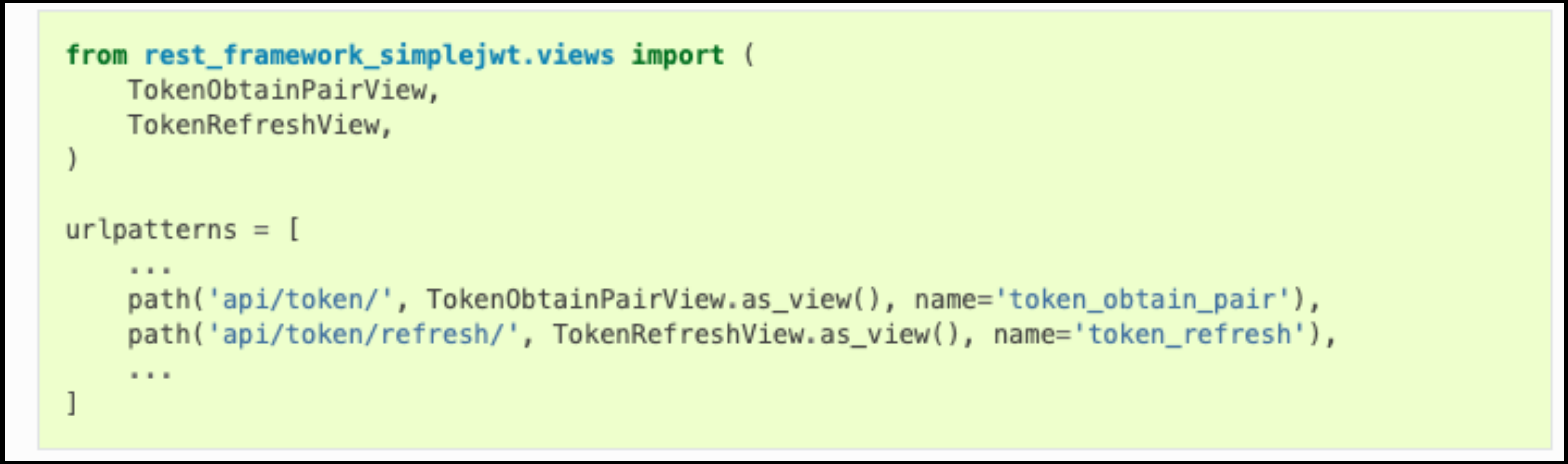
Authentication: JWT Authentication in Django REST

- The following steps can be taken to use JWT in django:

  $ pip install djangorestframework-simplejwt

  Then, your django project must be configured to use the library. In settings.py, add rest_framework_simplejwt.authentication.JWTAuthentication to the list of authentication classes.

  Include routes for Simple JWT's TokenObtainPairView and TokenRefreshView views in the root urls.py file



```
from rest_framework_simplejwt.views import (
    TokenObtainPairView,
    TokenRefreshView,
)

urlpatterns = [
    ...
    path('api/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
    ...
]
```

Django REST framework

Permissions

- Together with authentication and throttling, permissions determine whether a request should be granted or denied access.
- Permission checks are always run at the very start of the view, before any other code is allowed to proceed.
- It uses the authentication information in the request.user and request.auth properties to determine if the incoming request should be permitted.
- Permissions are used to grant or deny access for different classes of users to different parts of the API.

Django REST framework

Permissions: How Permissions are determined?

- Permissions in REST framework are always defined as a list of permission classes.
- Before running the main body of the view each permission in the list is checked. If any permission check fails an exceptions.PermissionDenied or exceptions.NotAuthenticated exception will be raised, and the main body of the view will not run.
- Rules for permission check:
  - Request was authenticated but permission was denied: HTTP 403 Forbidden
  - Request not authenticated and highest priority authentication class doesn’t use ww-Authenticate headers: HTTP 403 Forbidden
  - Request authenticated and highest priority authentication class does use ww-Authenticate headers: HTTP 401 Unauthorized with an WWW-Authenticate header

Django REST framework

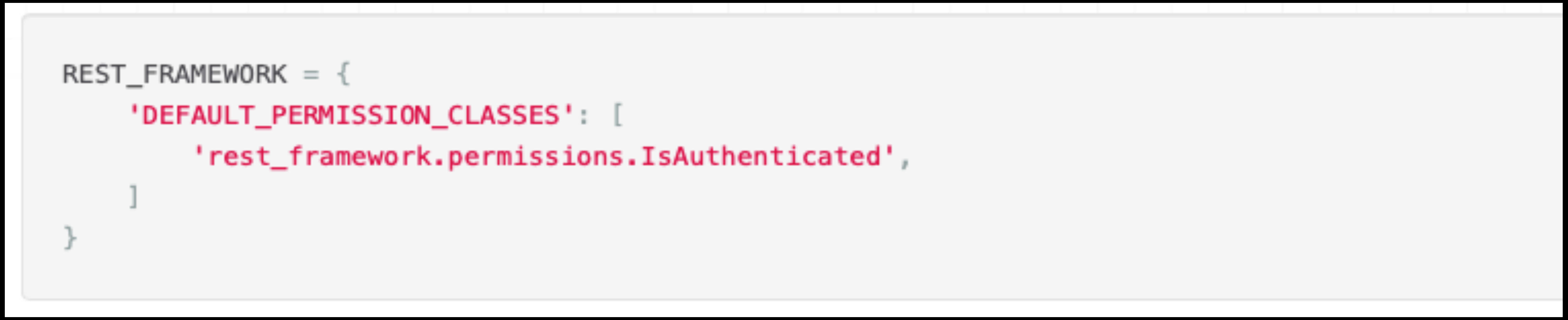
Permissions: Object Level Permissions

- Object level permissions are used to determine if a user should be allowed to act on a particular object, which will typically be a model instance.
- Object level permissions are run by REST framework's generic views when .get_object() is called.
- As with view level permissions, an exceptions.PermissionDenied exception will be raised if the user is not allowed to act on the given object.
- For performance reasons the generic views will not automatically apply object level permissions to each instance in a queryset when returning a list of objects.

Django REST framework

Permissions: Setting Permission Policy

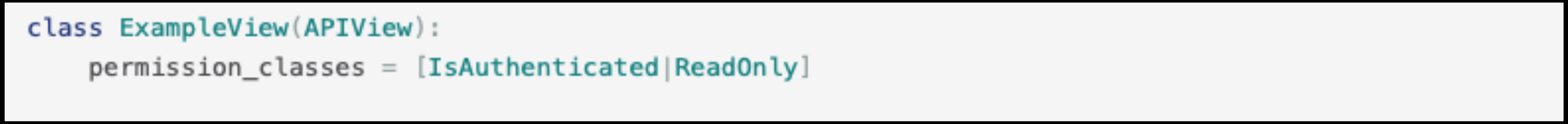
- The default permission policy may be set globally, using the DEFAULT_PERMISSION_CLASSES setting. For example:



```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated',
    ]
}
```

- If not specified it defaults to unrestricted access or AllowAny

- We may override the default list using per view basis permissions:



```
class ExampleView(APIView):
    permission_classes = [IsAuthenticated|ReadOnly]
```

Django REST framework

Permissions: Some Permission Classes

- AllowAny
- IsAuthenticated
- IsAdminUser
- IsAuthenticatedOrReadOnly
- DjangoModelPermissions

Django REST framework

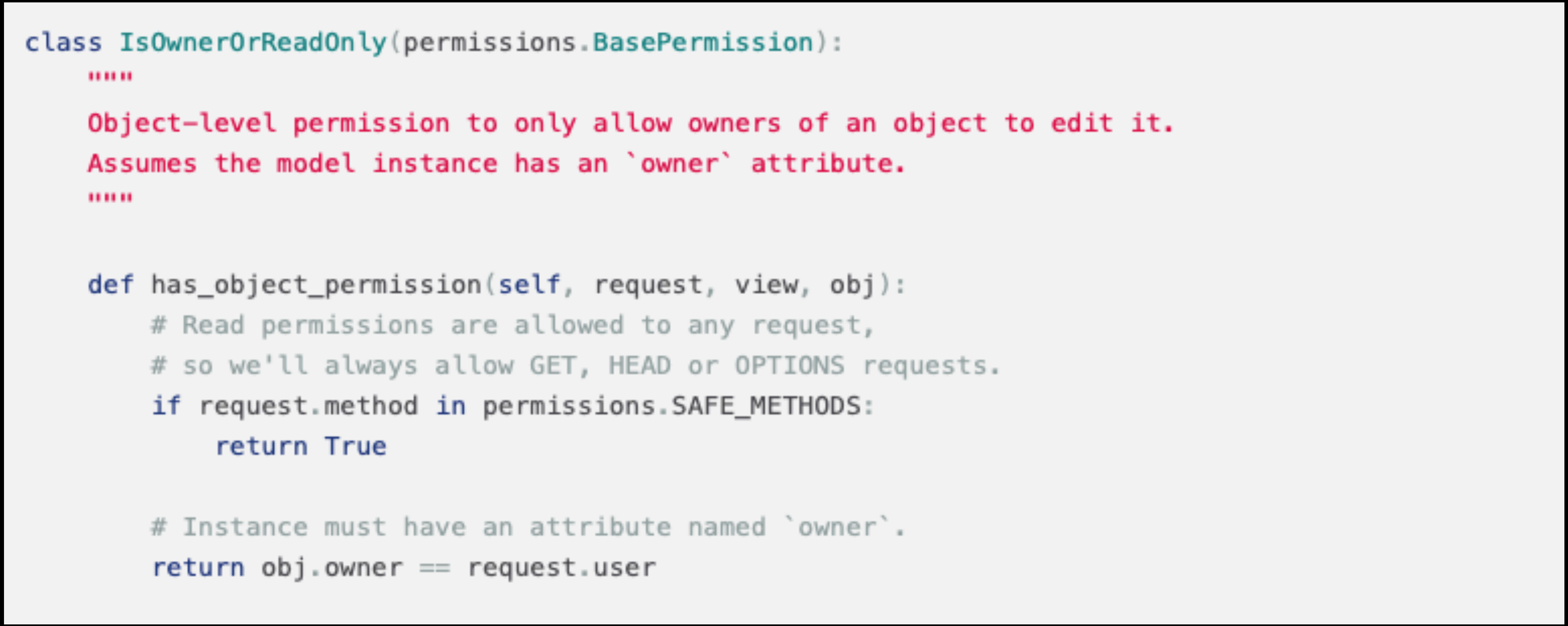
Permissions: Custom Permissions

- To implement a custom permission, we override BasePermission and implement either, or both, of the following methods:
  - .has_permission(self, request, view)
  - .has_object_permission(self, request, view, obj)
- The methods should return True if the request should be granted access, and False otherwise.
- We may check the request method against the constant SAFE_METHODS to identify if it is read or write request.
- The instance-level has_object_permission method will only be called if the view-level has_permission checks have already passed.

Django REST framework

Permissions: Custom Permissions

- Custom permissions will raise a PermissionDenied exception if the test fails.
- Example:



```python
class IsOwnerOrReadOnly(permissions.BasePermission):
    """
    Object-level permission to only allow owners of an object to edit it.
    Assumes the model instance has an `owner` attribute.
    """

    def has_object_permission(self, request, view, obj):
        # Read permissions are allowed to any request,
        # so we'll always allow GET, HEAD or OPTIONS requests.
        if request.method in permissions.SAFE_METHODS:
            return True

        # Instance must have an attribute named `owner`.
        return obj.owner == request.user
```

Django REST framework

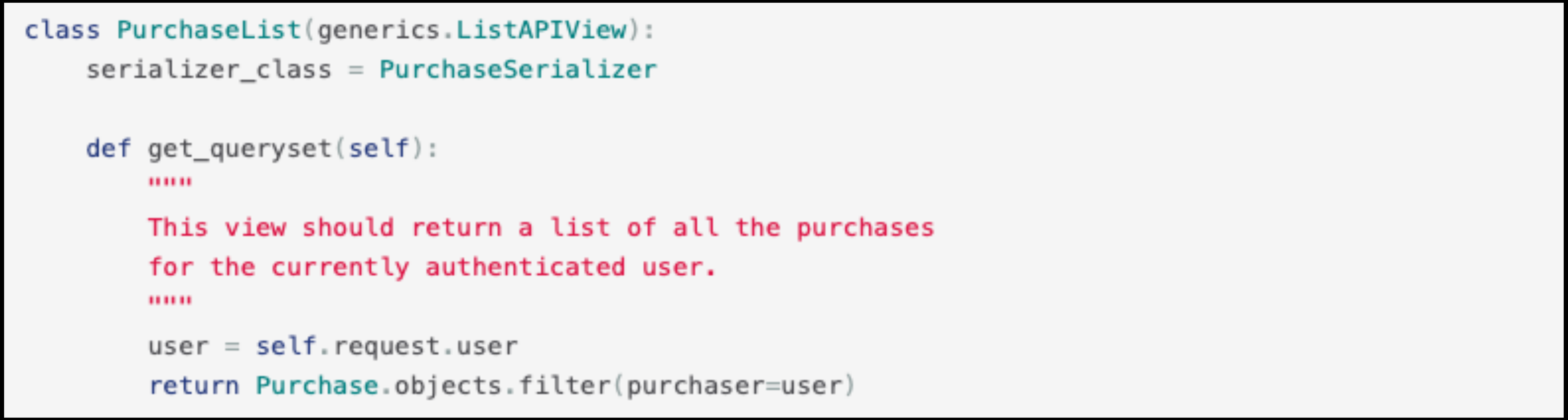
Filtering

- We can use filters to restrict items returned by the queryset rather than returning the entrie queryset.
- The simplest way to filter the queryset of any view that subclasses GenericAPIView is to override the .get_queryset() method.
- Overriding this method allows you to customize the queryset returned by the view in a number of different ways.

Django REST framework

Filtering: Filtering against the current user

- We might want to filter the queryset to ensure that only results relevant to the currently authenticated user making the request are returned.
- We can do so by filtering based on the value of request.user.



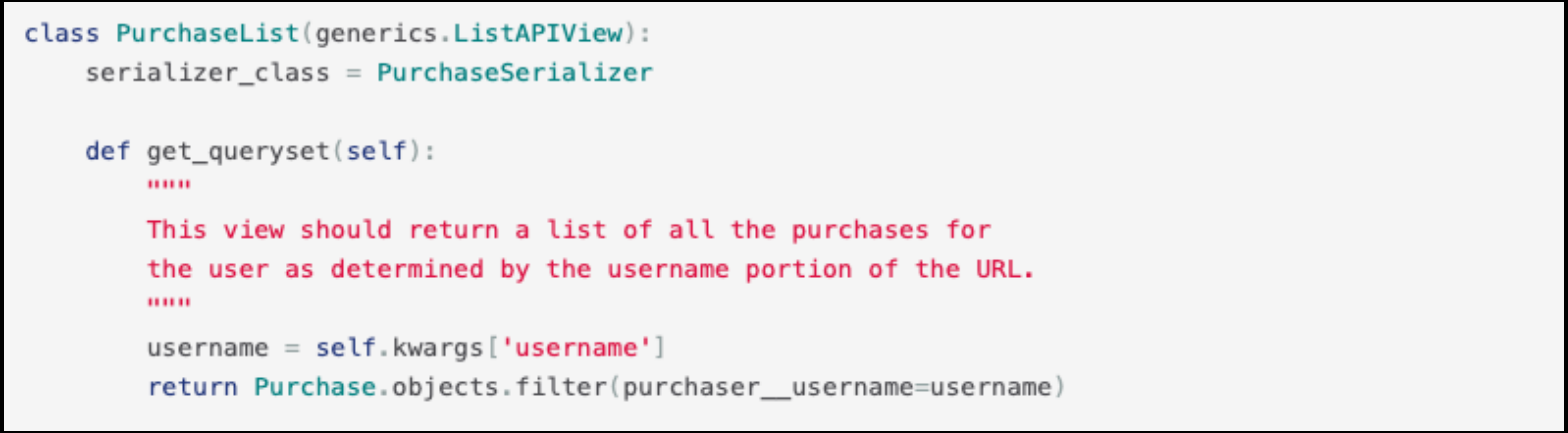
```python
class PurchaseList(generics.ListAPIView):
    serializer_class = PurchaseSerializer

    def get_queryset(self):
        """
        This view should return a list of all the purchases
        for the currently authenticated user.
        """
        user = self.request.user
        return Purchase.objects.filter(purchaser=user)
```

Django REST framework

Filtering: Filtering against the URL

- This method involves restricting the queryset based on some part of the URL.
- Example: for url such as (‘purchases/{username}’)



```python
class PurchaseList(generics.ListAPIView):
    serializer_class = PurchaseSerializer

    def get_queryset(self):
        """
        This view should return a list of all the purchases for
        the user as determined by the username portion of the URL.
        """
        username = self.kwargs['username']
        return Purchase.objects.filter(purchaser__username=username)
```

Django REST framework

Filtering: Filtering against query parameters

- We may also determine the initial queryset based on query parameters in the url.

```
class PurchaseList(generics.ListAPIView):
    serializer_class = PurchaseSerializer

    def get_queryset(self):
        """
        Optionally restricts the returned purchases to a given user,
        by filtering against a `username` query parameter in the URL.
        """
        queryset = Purchase.objects.all()
        username = self.request.query_params.get('username', None)
        if username is not None:
            queryset = queryset.filter(purchaser__username=username)
        return queryset
```

Django REST framework

Filtering: Generic Filtering

- REST framework also includes support for generic filtering backends that allows to easily construct complex searches and filters.
- The default filter backends may be set globally, using the DEFAULT_FILTER_BACKENDS setting. For example:

```
  REST_FRAMEWORK = {

      'DEFAULT_FILTER_BACKENDS':
  ['django_filters.rest_framework.DjangoFilterBackend']

}
```

- Or we may also set it per view using filter_backends=[‘DjangoFilterBackend’]

Django REST framework

Filtering: Generic Filtering

- If simple equality-based filtering is needed, we can set a filterset_fields attribute on the view, or viewset, listing the set of fields we wish to filter against.

```
class ProductList(generics.ListAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    filter_backends = [DjangoFilterBackend]
    filterset_fields = ['category', 'in_stock']
```

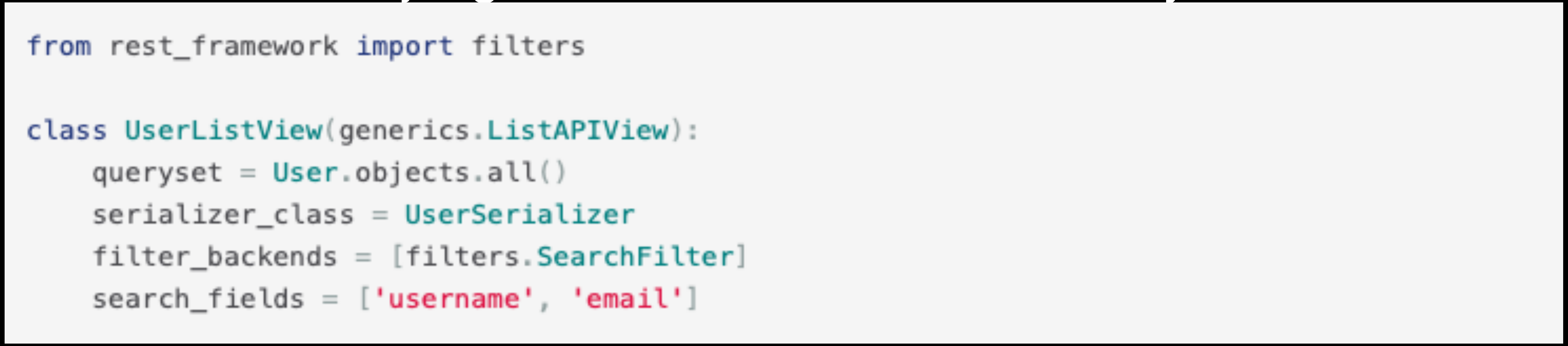
- This creates a FilterSet class for the given fields and will allow to make requests such as:

http://example.com/api/products?category=clothing&in_stock=True

Django REST framework

Filtering: Generic Filtering SearchFilter

- The SearchFilter class supports simple single query parameter based searching, and is based on the Django admin's search functionality.->CODE



```
from rest_framework import filters

class UserListView(generics.ListAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    filter_backends = [filters.SearchFilter]
    search_fields = ['username', 'email']
```

- This will allow the client to filter the items in the list by making queries such as:

http://example.com/api/users?search=russell

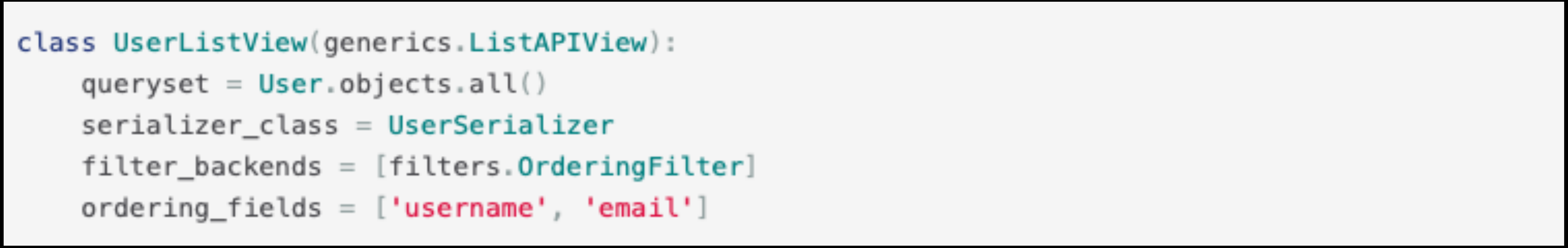
- You can also perform a related lookup on a ForeignKey or ManyToManyField with the lookup API double-underscore notation. Also modify search behaviours(^,=,@ etc)

search_fields = ['^username', '=email', ‘profile__profession']

Django REST framework

Filtering: Generic Filtering OrderingFilter

- The OrderingFilter class supports simple query parameter controlled ordering of results.



```python
class UserListView(generics.ListAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    filter_backends = [filters.OrderingFilter]
    ordering_fields = ['username', 'email']
```

- This will allow the client to filter the items in the list by making queries such as:

  http://example.com/api/users?ordering=username

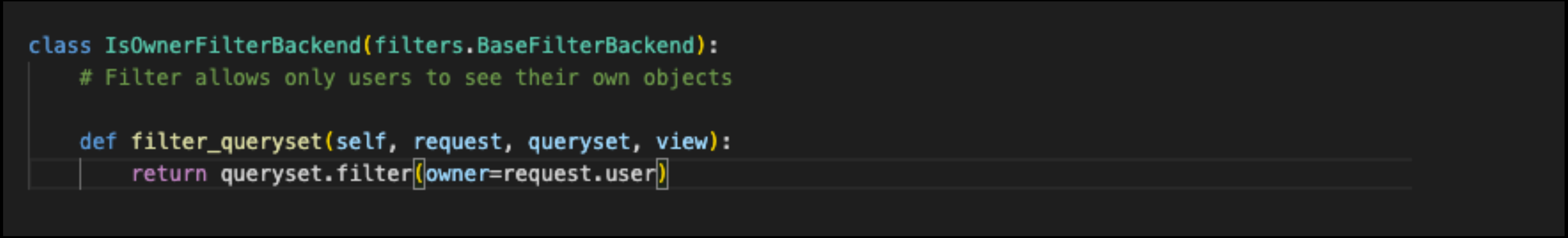
- If we don't specify an ordering_fields attribute on the view, the filter class will default to allowing the user to filter on any readable fields on the serializer specified by the serializer_class attribute.

- Specifying fields explicitly prevents data leaks.

Django REST framework

Filtering: Custom Filtering

- We can also provide our own generic filtering backend
- To do so we override BaseFilterBackend, and override the .filter_queryset(self, request, queryset, view) method. The method should return a new, filtered queryset.
- Example:



```python
class IsOwnerFilterBackend(filters.BaseFilterBackend):
    # Filter allows only users to see their own objects

    def filter_queryset(self, request, queryset, view):
        return queryset.filter(owner=request.user)
```

Django REST framework

Views

- REST framework provides an APIView class, which subclasses Django's View class.
- APIView classes are different from regular View classes in the following ways:
  - Requests passed to the handler methods will be REST framework's Request instances, not Django's HttpRequest instances.
  - Handler methods may return REST framework's Response, instead of Django's HttpResponse. The view will manage content negotiation and setting the correct renderer on the response.
  - Any APIException exceptions will be caught and mediated into appropriate responses.
  - Incoming requests will be authenticated and appropriate permission and/or throttle checks will be run before dispatching the request to the handler method.

Django REST framework

Views

- Function based views with @api_view
- Class based views with APIView
- Mixins
- GenericAPIView
- Concrete View Classes

Django REST framework

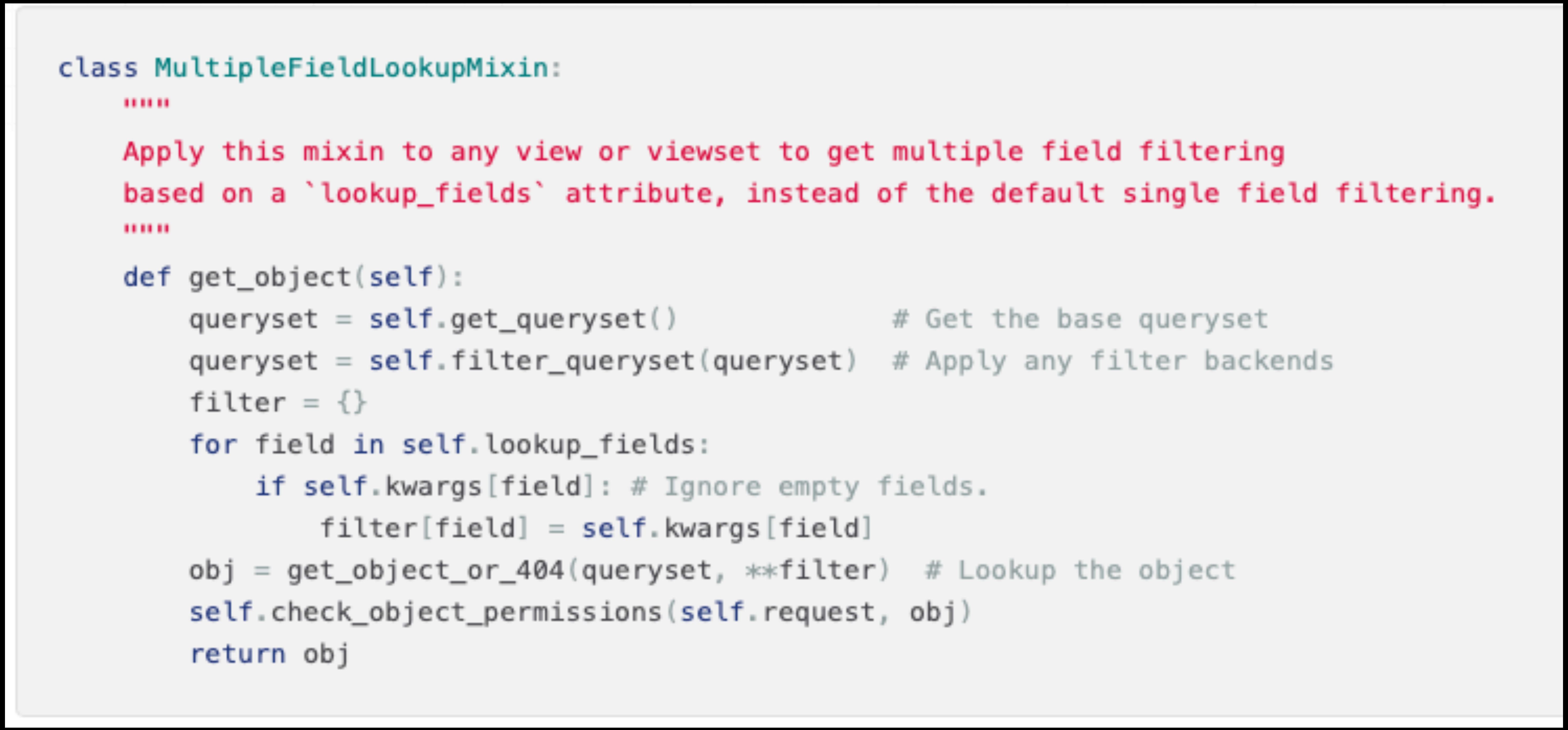
Views: Customizing the generic views

- We might want to refactor reused behavior into a common class that we can then just apply to any view or viewset as needed.
- We may either:
  - Create a custom mixin
  - Create a custom base class

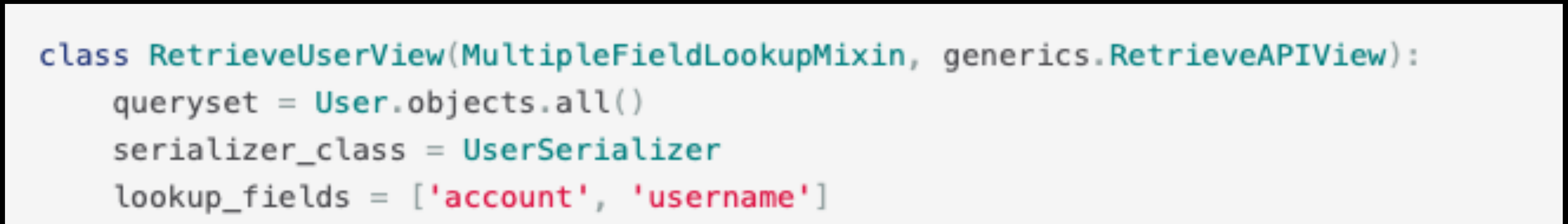
Django REST framework

Views: Customizing the generic views

- Creating custom mixins: to lookup objects based on multiple fields in the URL conf, we could create a mixin class like the following:



```python
class MultipleFieldLookupMixin:
    """
    Apply this mixin to any view or viewset to get multiple field filtering
    based on a `lookup_fields` attribute, instead of the default single field filtering.
    """
    def get_object(self):
        queryset = self.get_queryset()             # Get the base queryset
        queryset = self.filter_queryset(queryset)  # Apply any filter backends
        filter = {}
        for field in self.lookup_fields:
            if self.kwargs[field]: # Ignore empty fields.
                filter[field] = self.kwargs[field]
        obj = get_object_or_404(queryset, **filter)  # Lookup the object
        self.check_object_permissions(self.request, obj)
        return obj
```

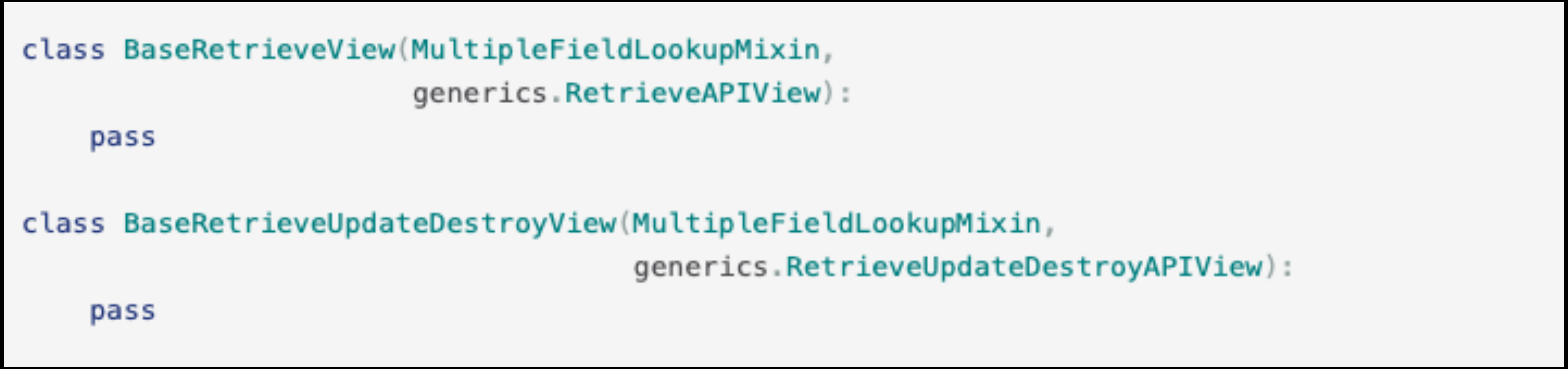


```python
class RetrieveUserView(MultipleFieldLookupMixin, generics.RetrieveAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    lookup_fields = ['account', 'username']
```

Django REST framework

Views: Customizing the generic views

- Creating custom base classes: For recurring mixins ,we can create our own set of base views that can then be used throughout your project. For example:



```python
class BaseRetrieveView(MultipleFieldLookupMixin,
                       generics.RetrieveAPIView):
    pass

class BaseRetrieveUpdateDestroyView(MultipleFieldLookupMixin,
                                    generics.RetrieveUpdateDestroyAPIView):
    pass
```

Django REST framework

Views: Viewsets

- Django REST framework allows you to combine the logic for a set of related views in a single class, called a ViewSet.
- Viewsets provide actions such as .list() and .create(). It does not provide any method handlers such as .get() or .post()
- Rather than explicitly registering the views in a viewset in the urlconf, we'll register the viewset with a router class, that automatically determines the urlconf for us.

Django REST framework

Views: Viewsets

- Example:

```python
class UserViewSet(viewsets.ViewSet):
    """
    A simple ViewSet for listing or retrieving users.
    """
    def list(self, request):
        queryset = User.objects.all()
        serializer = UserSerializer(queryset, many=True)
        return Response(serializer.data)

    def retrieve(self, request, pk=None):
        queryset = User.objects.all()
        user = get_object_or_404(queryset, pk=pk)
        serializer = UserSerializer(user)
        return Response(serializer.data)
```

- Then we register the viewset with a router and allow the urlconf to be automatically generated -> Code

Django REST framework

Views: Viewset Actions

- list()
- create()
- retrieve()
- update()
- partial_update()
- destroy()

Django REST framework

Views: Viewset ModelViewSet

- The ModelViewSet class inherits from GenericAPIView and includes implementations for various actions, by mixing in the behavior of the various mixin classes.

- The actions provided by the ModelViewSet class are .list(), .retrieve(), .create(), .update(), .partial_update(), and .destroy().

- Because ModelViewSet extends GenericAPIView, we'll normally need to provide at least the queryset and serializer_class attributes.

- ReadOnlyModelVIewSet: Unlike ModelViewSet only provides the 'read-only' actions, .list() and .retrieve().

Django REST Framework

Custom User Model

- We can override Django’s built in user model to use our own custom user model.
- User is tightly interwoven with the rest of Django internally. It is challenging to switch over to a custom user model mid- project. So it is recommended to start first with custom user model.
- Choices: either extend AbstractUser which keeps the default User fields and permissions or extend AbstractBaseUser which is more granular, and flexible.

Django REST Framework

Custom User Model

- There are four steps for adding a custom user model to our project:

  1. Create a CustomUser model
  2. Update settings.py
  3. Customize UserCreationForm and UserChangeForm
  4. Add the custom user model to admin.py

Django REST Framework

Custom User Model

1. Create a CustomUser model: Separate app for 'Users' and create 'CustomUser' model inside the app.

2. Register the app and set: AUTH_USER_MODEL=‘users.CustomUser’. Then Migrate

3. A user model can be both created and edited within the Django admin. So we’ll need to update the built-in forms too to point to CustomUser instead of User.

 - Create a users/forms.py file.

 - Inherit UserCreationForm and UserChangeForm and override it

4.The admin is a common place to manipulate user data and there is tight coupling between the built-in User and the admin. We change admin.py to make these changes

 We’ll extend the existing UserAdmin into CustomUserAdmin and tell Django to use our new forms, custom user model, and list only the email and username of a user.

Thank You!!

References

- https://restfulapi.net
- https://en.wikipedia.org/wiki/Representational_state_transfer
- https://medium.com/swlh/build-your-first-rest-api-with-django-rest-framework-e394e39a482c
- https://jwt.io/introduction/
- https://django-rest-framework-simplejwt.readthedocs.io/en/latest/getting_started.html#requirements
- https://www.django-rest-framework.org/api-guide/serializers/
- https://krakensystems.co/blog/2020/custom-users-using-django-rest-framework
- https://simpleisbetterthancomplex.com/tutorial/2018/12/19/how-to-use-jwt-authentication-with-django-rest-framework.html
- https://django-rest-auth.readthedocs.io/en/latest/installation.html#registration-optional