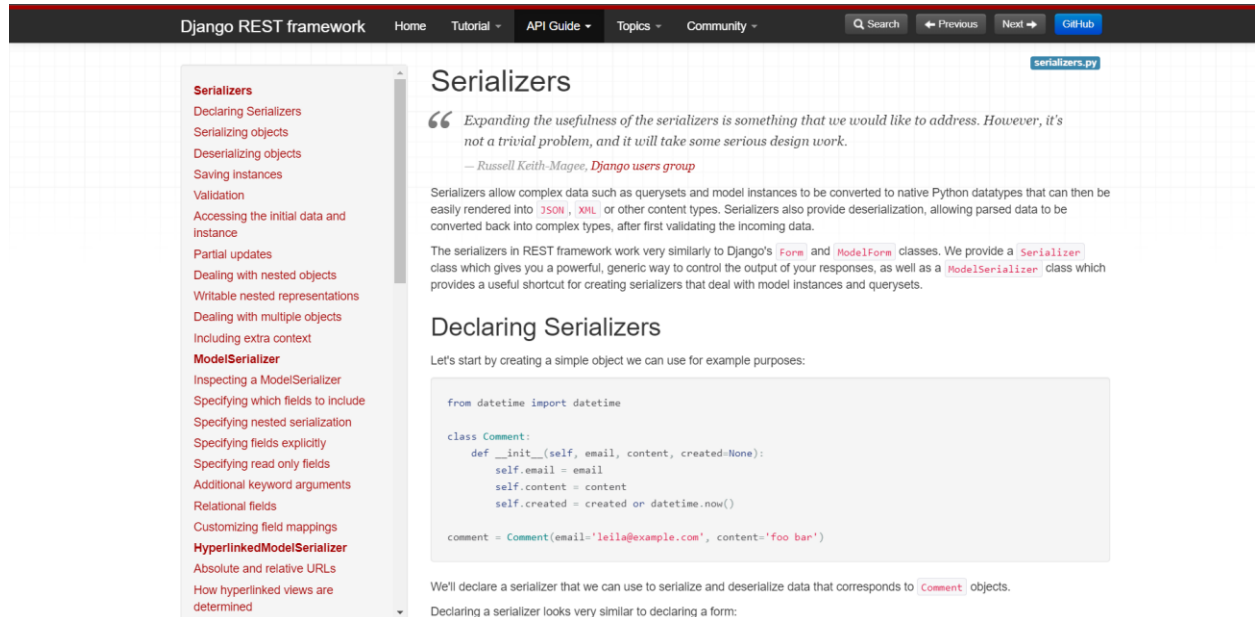


Django serializer:

<https://www.django-rest-framework.org/api-guide/serializers/>



The screenshot shows the 'Serializers' page of the Django REST framework documentation. The page has a dark header with navigation links: Home, Tutorial, API Guide, Topics, and Community. A search bar and navigation buttons (Previous, Next, GitHub) are also present. A sidebar on the left lists various topics under the 'Serializers' heading, including 'Declaring Serializers', 'Serializing objects', 'Deserializing objects', 'Saving instances', 'Validation', 'Accessing the initial data and instance', 'Partial updates', 'Dealing with nested objects', 'Writable nested representations', 'Dealing with multiple objects', 'Including extra context', 'ModelSerializer', 'Inspecting a ModelSerializer', 'Specifying which fields to include', 'Specifying nested serialization', 'Specifying fields explicitly', 'Specifying read only fields', 'Additional keyword arguments', 'Relational fields', 'Customizing field mappings', 'HyperlinkedModelSerializer', 'Absolute and relative URLs', and 'How hyperlinked views are determined'. The main content area is titled 'Serializers' and features a quote from Russell Keith-Magee: 'Expanding the usefulness of the serializers is something that we would like to address. However, it's not a trivial problem, and it will take some serious design work.' Below the quote, the text explains that serializers allow complex data like querysets and model instances to be converted to native Python datatypes. It also mentions that serializers provide deserialization and validation. The page then introduces the 'Serializer' and 'ModelSerializer' classes. A code block shows the declaration of a 'Comment' serializer. The page concludes with a note that declaring a serializer looks very similar to declaring a form.

Django REST framework Home Tutorial API Guide Topics Community Search Previous Next GitHub

Serializers

serializers.py

Serializers

- Declaring Serializers
- Serializing objects
- Deserializing objects
- Saving instances
- Validation
- Accessing the initial data and instance
- Partial updates
- Dealing with nested objects
- Writable nested representations
- Dealing with multiple objects
- Including extra context
- ModelSerializer**
- Inspecting a ModelSerializer
- Specifying which fields to include
- Specifying nested serialization
- Specifying fields explicitly
- Specifying read only fields
- Additional keyword arguments
- Relational fields
- Customizing field mappings
- HyperlinkedModelSerializer**
- Absolute and relative URLs
- How hyperlinked views are determined

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

— Russell Keith-Magee, *Django users group*

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

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

Declaring Serializers

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

```
from datetime import datetime

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

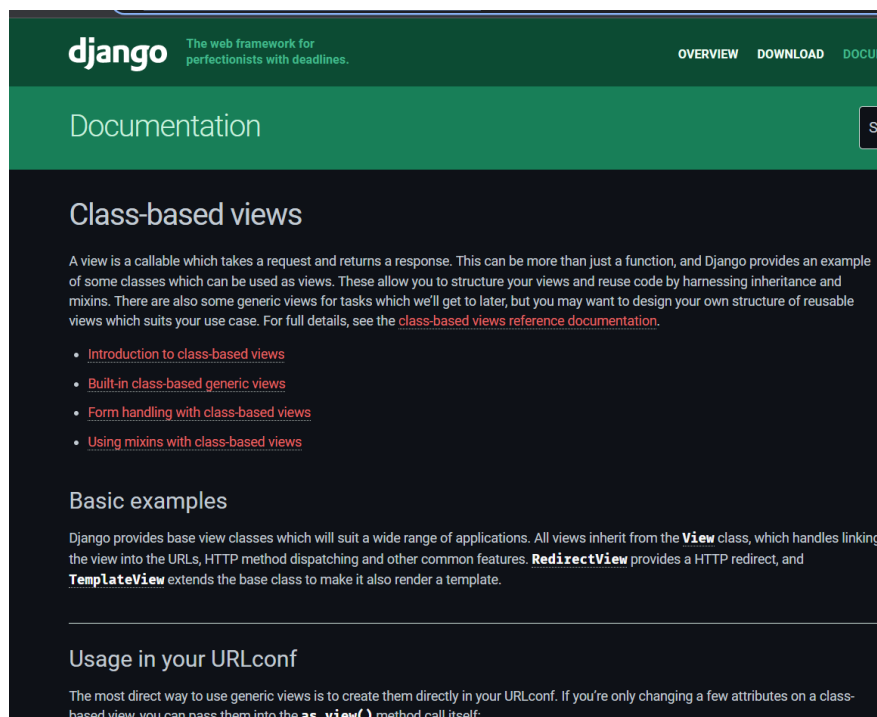
comment = Comment(email='leila@example.com', content='foo bar')
```

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

Declaring a serializer looks very similar to declaring a form:

Class based view:

<https://docs.djangoproject.com/en/4.1/topics/class-based-views/>



The screenshot shows the 'Class-based views' page in the Django documentation. The page has a dark header with the Django logo and tagline 'The web framework for perfectionists with deadlines.' Navigation links for 'OVERVIEW', 'DOWNLOAD', and 'DOCUMENTATION' are present. The main content area is titled 'Class-based views' and explains that a view is a callable which takes a request and returns a response. It mentions that Django provides an example of some classes which can be used as views, and that these allow you to structure your views and reuse code by harnessing inheritance and mixins. It also notes that there are some generic views for tasks which will be covered later, but that you may want to design your own structure of reusable views which suits your use case. A list of links is provided: 'Introduction to class-based views', 'Built-in class-based generic views', 'Form handling with class-based views', and 'Using mixins with class-based views'. The page then has a section titled 'Basic examples' which explains that Django provides base view classes which will suit a wide range of applications. It mentions that all views inherit from the `View` class, which handles linking the view into the URLs, HTTP method dispatching and other common features. It also mentions that `RedirectView` provides a HTTP redirect, and `TemplateView` extends the base class to make it also render a template. The page concludes with a section titled 'Usage in your URLconf' which explains that the most direct way to use generic views is to create them directly in your URLconf. It also mentions that if you're only changing a few attributes on a class-based view, you can pass them into the `as_view()` method call itself.

django The web framework for perfectionists with deadlines. OVERVIEW DOWNLOAD DOCUMENTATION

Documentation

Class-based views

A view is a callable which takes a request and returns a response. This can be more than just a function, and Django provides an example of some classes which can be used as views. These allow you to structure your views and reuse code by harnessing inheritance and mixins. There are also some generic views for tasks which we'll get to later, but you may want to design your own structure of reusable views which suits your use case. For full details, see the [class-based views reference documentation](#).

- [Introduction to class-based views](#)
- [Built-in class-based generic views](#)
- [Form handling with class-based views](#)
- [Using mixins with class-based views](#)

Basic examples

Django provides base view classes which will suit a wide range of applications. All views inherit from the `View` class, which handles linking the view into the URLs, HTTP method dispatching and other common features. `RedirectView` provides a HTTP redirect, and `TemplateView` extends the base class to make it also render a template.

Usage in your URLconf

The most direct way to use generic views is to create them directly in your URLconf. If you're only changing a few attributes on a class-based view, you can pass them into the `as_view()` method call itself.

API Integration:

<https://www.youtube.com/watch?v=D5nVnRjPrKA&t=1047s>

Carousel Slider:

<https://www.youtube.com/watch?v=p3gFikowJVI&t=2s>