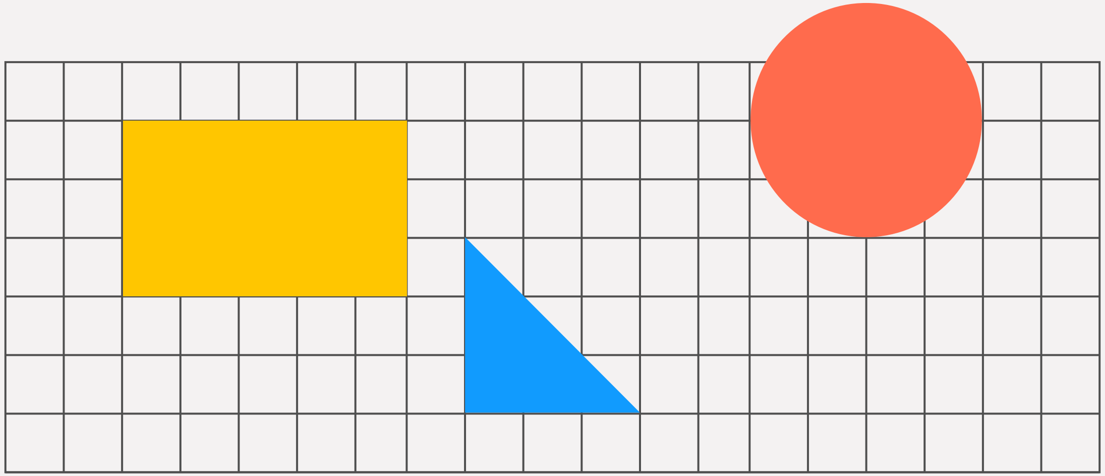
Django Needs Some REST!

Ali Abrishami



In this Lecture You will...

a

Review the concept of REST APIs

b

Meet DRF, Django REST Framework

c

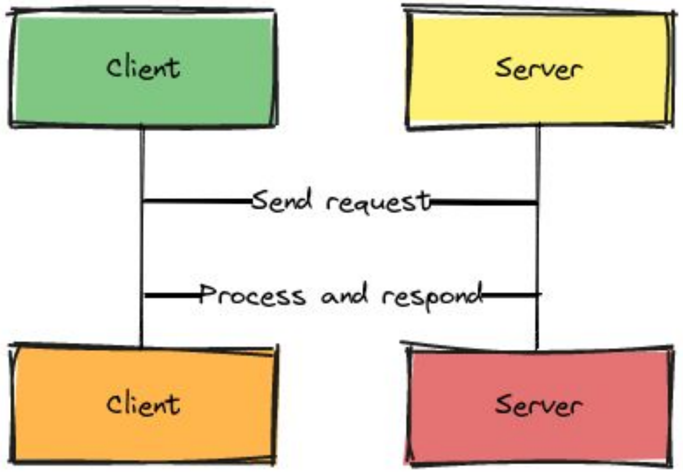
Create your first RESTful Backend!

REMIDER: REST

- REST is an **architectural style**, not a framework, library, or protocol you can install.
- **Resources** are the core concept; **URLs** name things, not actions.
- HTTP methods have meaning: GET reads, POST creates, PUT replaces, PATCH modifies, DELETE removes.
- Statelessness is non-negotiable: each request must contain all required context.
- HTTP status codes are part of the API contract, not decorative numbers.

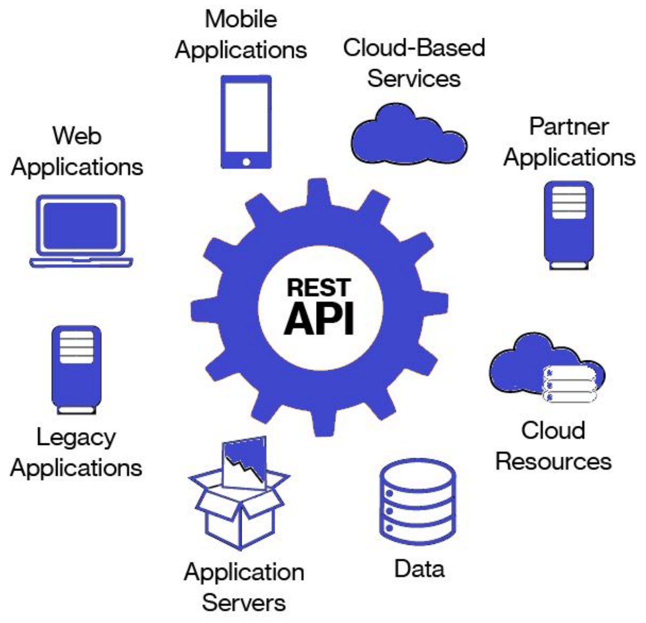
REMIDER: Client/Server

- Client and server are strictly separated.
  UI logic never leaks into backend decisions.
- The server owns data, rules, and validation, the client only requests and presents.
- Communication happens exclusively through a well-defined API contract.
- Either side can evolve independently as long as the contract is respected.
- Tight coupling is a design failure, not a convenience.



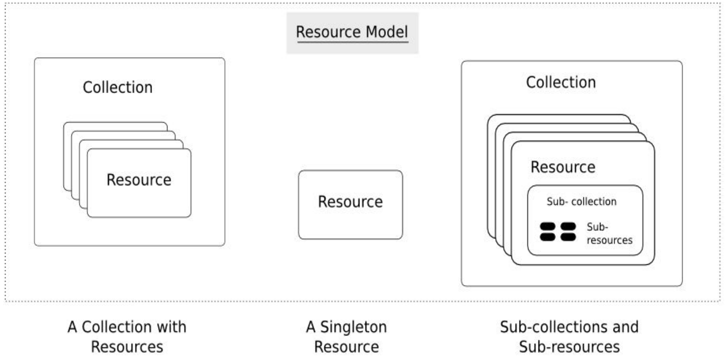
Details of REST

- Resources are identified by **URIs**. Verbs belong in HTTP **methods**, not in the URL path.
- Representations are negotiable: the same resource can be JSON, XML, etc.
- Stateless requests enable horizontal scaling and simplify failure recovery.
- Uniform interface constraints trade short-term convenience for long-term sanity.



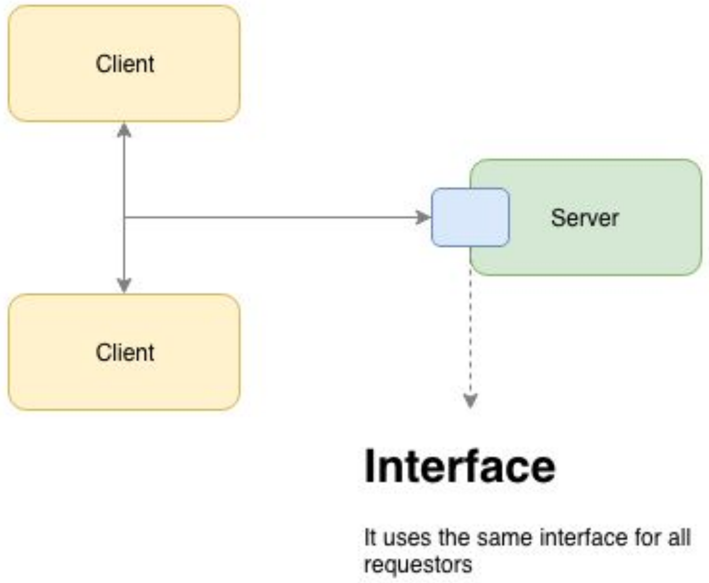
1st Rule of Fight Club!

- Everything is a resource, not an action.
- A resource is identified by a noun-based URI.
- Operations are expressed via HTTP methods, not custom endpoints.
- The same interface applies uniformly across all resources.
- If you need verbs in your URLs, you misunderstood the rule.



Rule No. 2

- The interface is uniform across all resources.
- Clients interact with resources in the same way, regardless of type.
- HTTP methods, headers, and status codes carry semantic meaning.
- Representations describe state, not behavior.
- Consistency beats cleverness every time.



And The 3rd Rule

- The server is stateless between requests.
- Each request must contain all the information needed to process it.
- No server-side session memory about previous requests.
- Authentication state is sent on every request.
- Statelessness enables scalability, reliability, and simplicity.

Class Activity Time!

Discuss the pros and cons of stateless vs. stateful!

REST APIs With Django

- Classic Django is optimized for server-rendered HTML, not API consumers.
- Views and templates tightly couple request handling with presentation.
- Forms and sessions assume browser-based, stateful interaction.
- Returning JSON manually works, but scales poorly and becomes repetitive.
- At this point, the correct conclusion is obvious: this is where DRF enters the picture.

Hi, DRF!

- Django REST Framework is a toolkit for building Web APIs on top of Django.
- It formalizes serialization, validation, and request/response handling.
- HTTP concepts become first-class citizens instead of afterthoughts.
- Authentication, permissions, and throttling are built in, not bolted on.

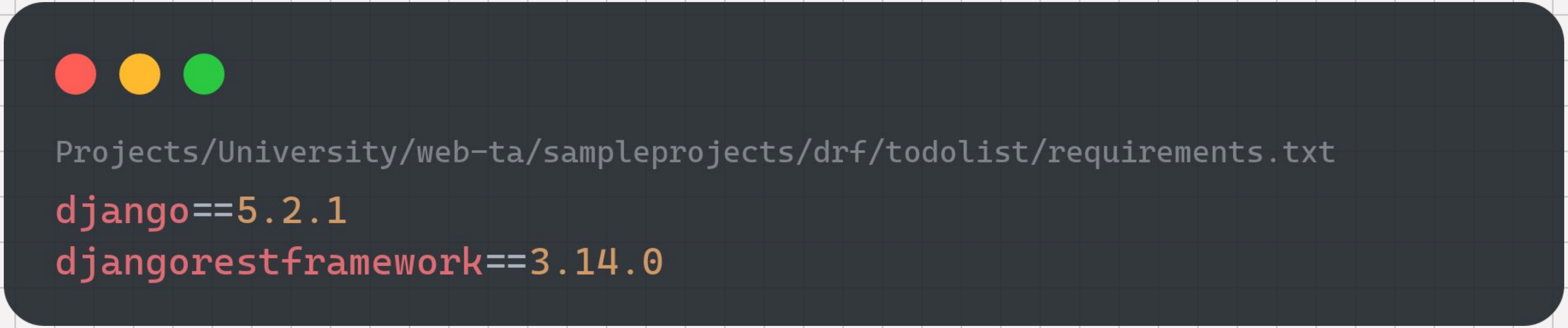
DRF Building Blocks

- **Requests and Responses**: DRF wraps Django’s `HttpRequest` with API awareness.
- **Serializers**: the translation layer between Python objects and representations.
- **Views**: where HTTP methods meet application logic.
- **Routers and URLs**: turning resources into consistent endpoints.
- **Authentication and Permissions**: deciding who can do what.

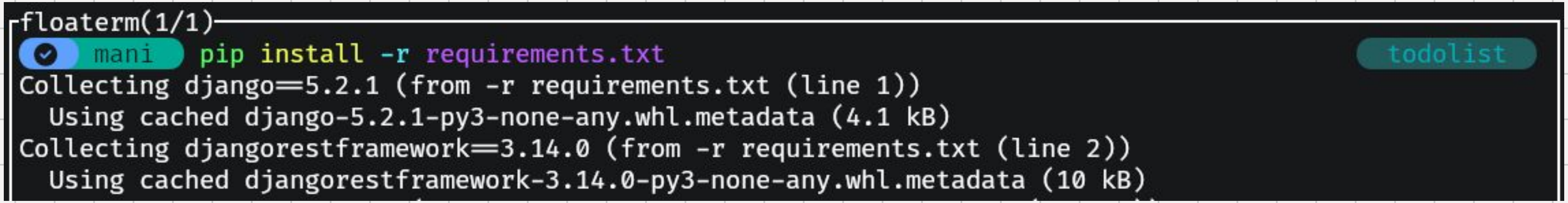
Setting Up DRF

- Install the framework via `pip install djangorestframework`
- Register `rest_framework` in `INSTALLED_APPS` so Django recognizes it.
- This activates DRF’s components without altering existing behavior.
- No configuration means defaults; defaults are intentional.
- From this point on, Django can speak REST natively.

Setting Up DRF (cont.)

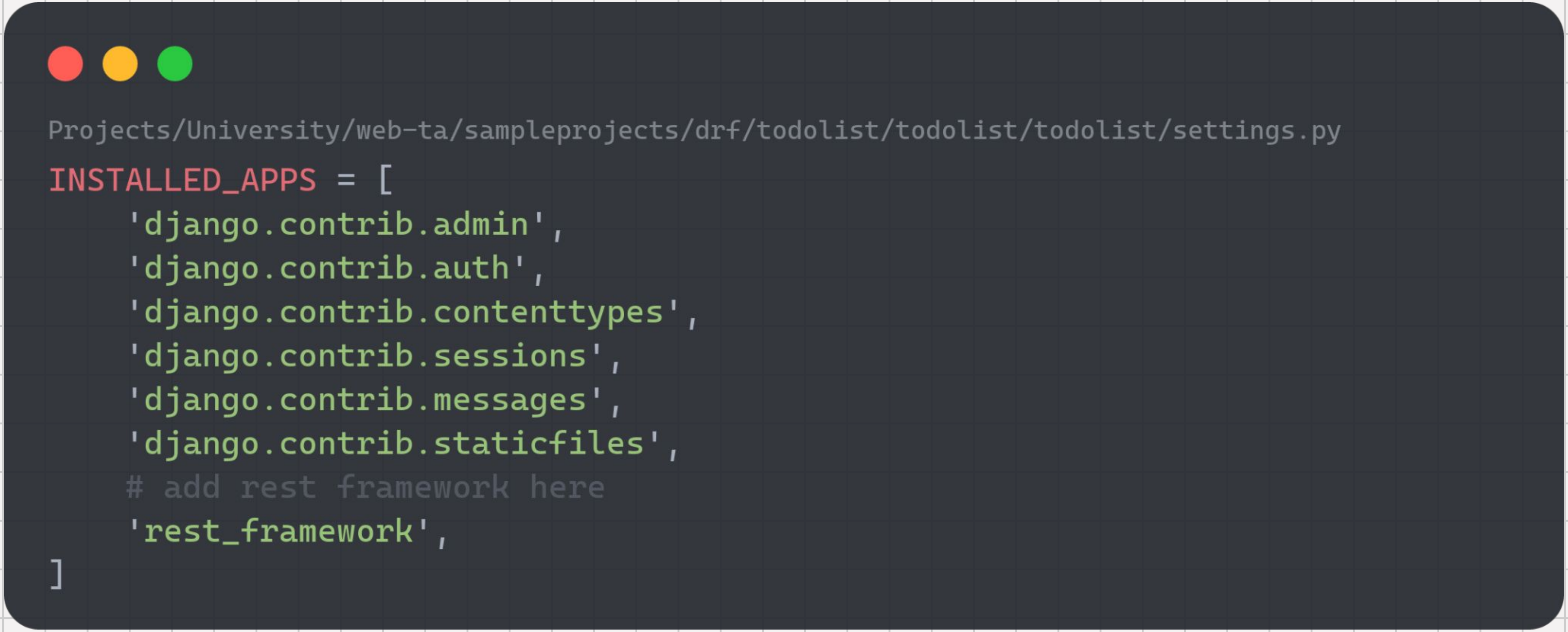


```
Projects/University/web-ta/sampleprojects/drf/todolist/requirements.txt
django==5.2.1
djangorestframework==3.14.0
```



```
floaterm(1/1)
 mani  pip install -r requirements.txt                                          todolist
Collecting django==5.2.1 (from -r requirements.txt (line 1))
  Using cached django-5.2.1-py3-none-any.whl.metadata (4.1 kB)
Collecting djangorestframework==3.14.0 (from -r requirements.txt (line 2))
  Using cached djangorestframework-3.14.0-py3-none-any.whl.metadata (10 kB)
```

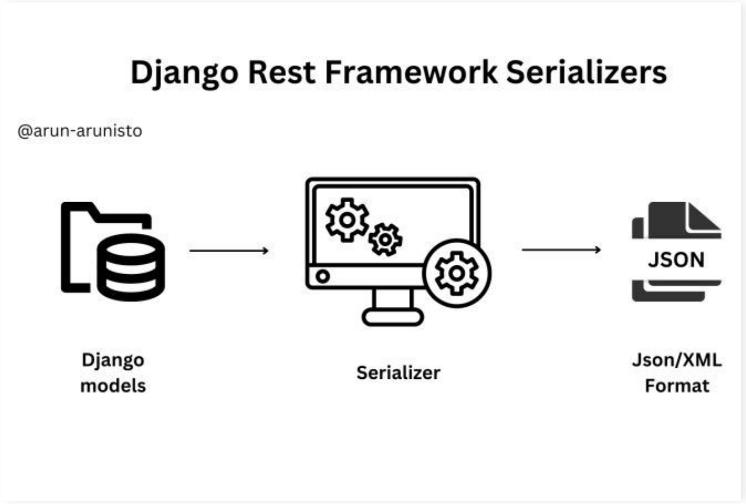
Setting Up DRF (cont.)



```
Projects/University/web-ta/sampleprojects/drf/todolist/todolist/todolist/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # add rest framework here
    'rest_framework',
]
```

Serializers

- Same responsibility as Django Forms: validation and data cleaning.
- No HTML, no widgets, no presentation concerns.
- Designed for JSON and other representations, not browsers.
- Explicitly decouple API input/output from database models.
- Think of them as Forms without opinions about UI.

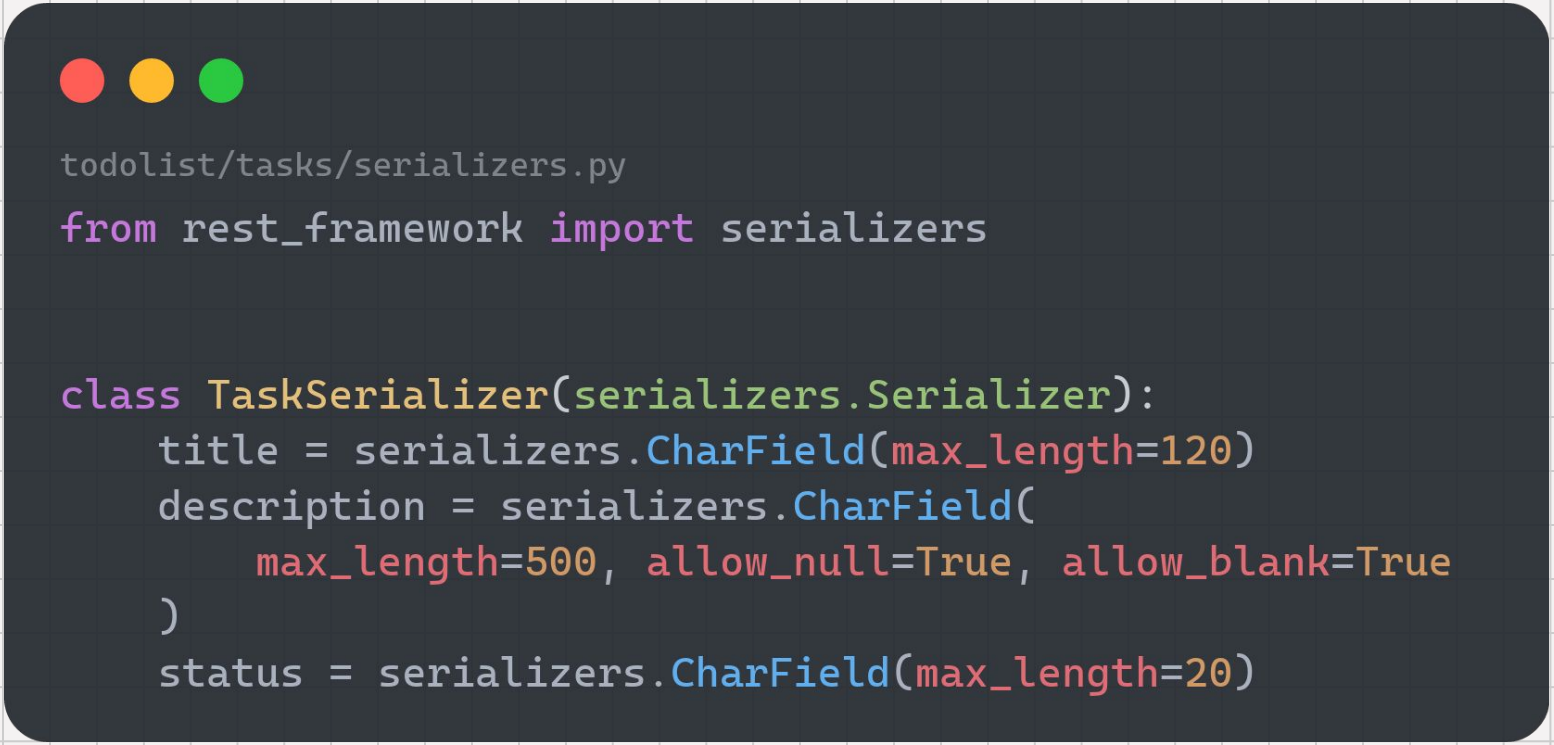


Sample Model

todolist/tasks/models.py

```python
from django.db import models


class Task(models.Model):
    class TaskStatus(models.TextChoices):
        TODO = "TODO"
        IN_PROGRESS = "IN_PROGRESS"
        DONE = "DONE"

    title = models.CharField(max_length=120)
    description = models.CharField(max_length=500, null=True, blank=True)
    status = models.CharField(
        max_length=20, choices=TaskStatus.choices, default=TaskStatus.TODO
    )
    creation_date = models.DateTimeField(auto_now_add=True)
```

Sample Serializer



todolist/tasks/serializers.py

```python
from rest_framework import serializers


class TaskSerializer(serializers.Serializer):
    title = serializers.CharField(max_length=120)
    description = serializers.CharField(
        max_length=500, allow_null=True, allow_blank=True
    )
    status = serializers.CharField(max_length=20)
```

Serializer Fields

- `fields` define the public shape of the API.
- `read_only_fields` protect server-controlled data.
- `required` enforces mandatory input.
- `default` supplies implicit values when omitted.
- `many` controls collection vs single-object handling.

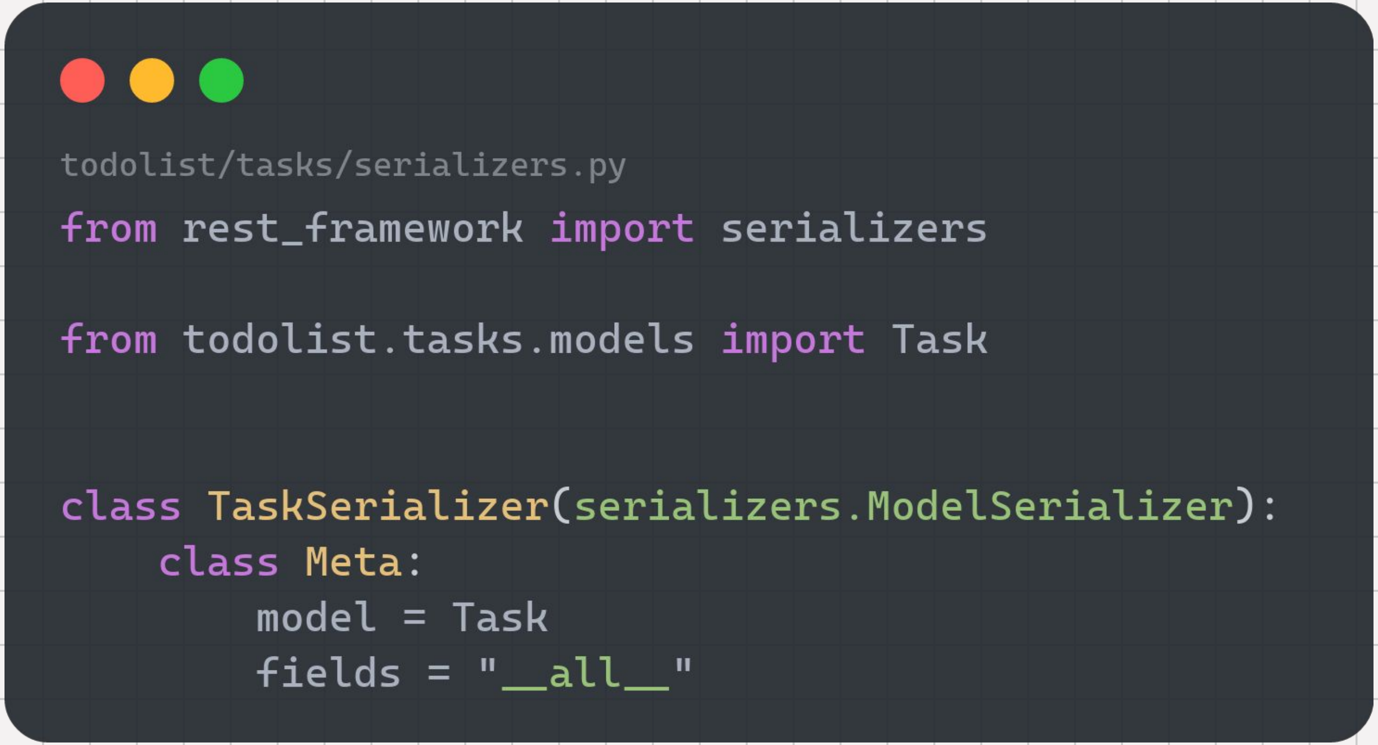
Serializer Methods

- `is_valid()` triggers validation and normalization.
- `validate()` handles cross-field validation.
- `validate_<field>()` enforces field-level rules.
- `create()` defines object creation behavior.
- `update()` defines object modification behavior.

ModelSerializer

- Automatically maps model fields to serializer fields.
- Reduces boilerplate without sacrificing validation.
- Keeps API representation aligned with the data model.
- Allows explicit overrides where behavior must differ.
- Ideal default until you have a concrete reason not to use it.

Simpler Serializer!

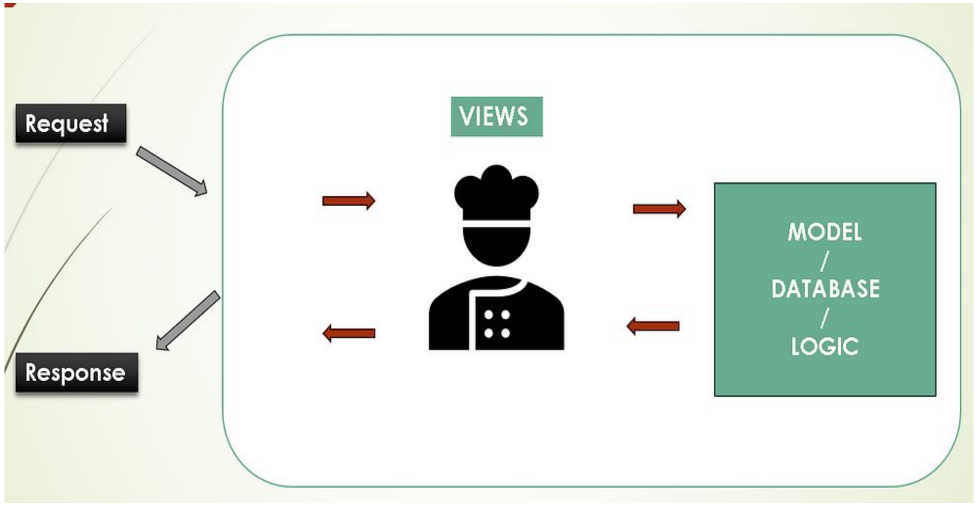


todolist/tasks/serializers.py

```python
from rest_framework import serializers

from todolist.tasks.models import Task


class TaskSerializer(serializers.ModelSerializer):
    class Meta:
        model = Task
        fields = "__all__"
```

Views

- Views are the entry point of every API request.
- They translate HTTP intent into application behavior.
- Each view represents operations on a resource, not arbitrary logic.
- Responsibility boundaries matter.
- A bloated view is a design smell, not a productivity win.



Functional API Views

- Use plain Python functions instead of classes.
- Explicitly map HTTP methods to function logic.
- Minimal abstraction, maximal control.
- Best for very small or highly custom endpoints.
- Scale poorly when complexity or reuse increases.

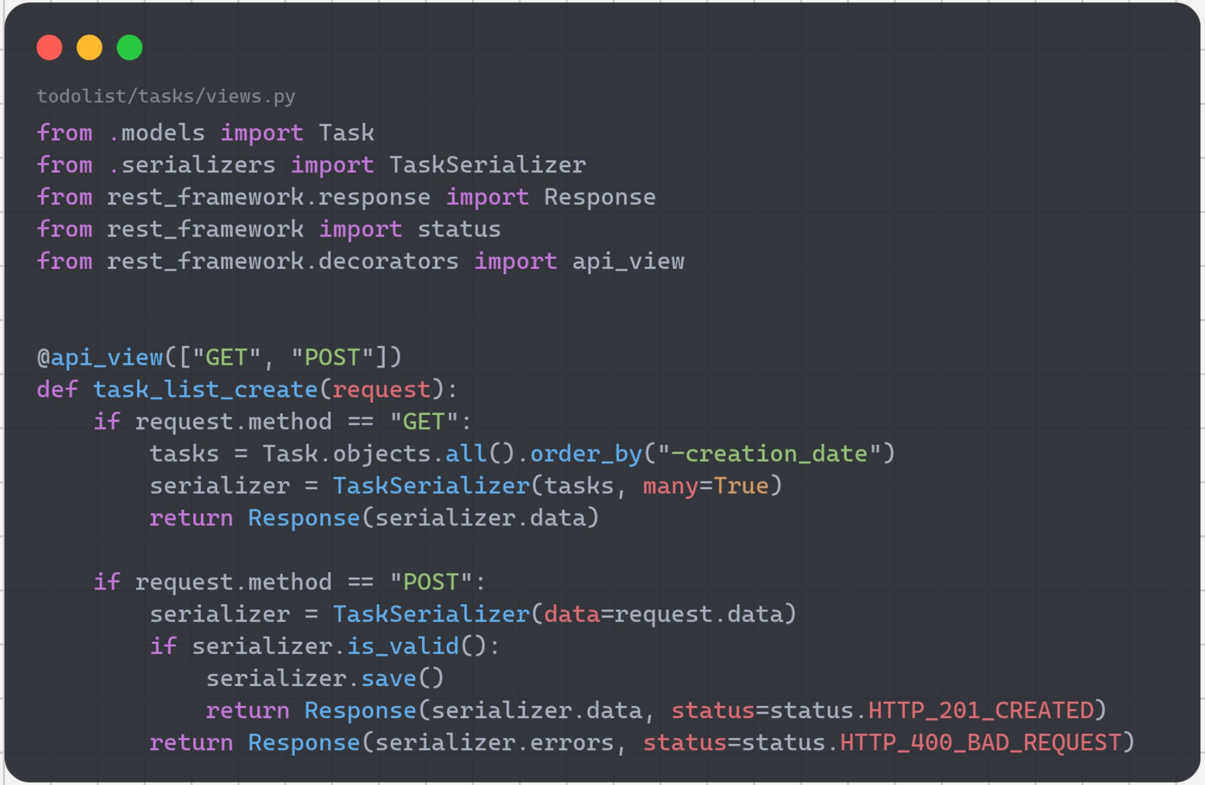
HTTP Status Codes

- DRF defines status codes in rest_framework.status.
- Each code is a named constant, not a magic number.
- Names encode intent: readability beats memorization.
- Used explicitly in responses and implicitly by exceptions.
- This makes HTTP semantics visible in code, not hidden in integers.

Examples of HTTP Codes

- `HTTP_200_OK`: successful read
- `HTTP_201_CREATED`: successful creation
- `HTTP_400_BAD_REQUEST`: validation or client error
- `HTTP_401_UNAUTHORIZED` / `HTTP_403_FORBIDDEN`: auth vs permission
- `HTTP_404_NOT_FOUND`: resource does not exist

Sample Functional API



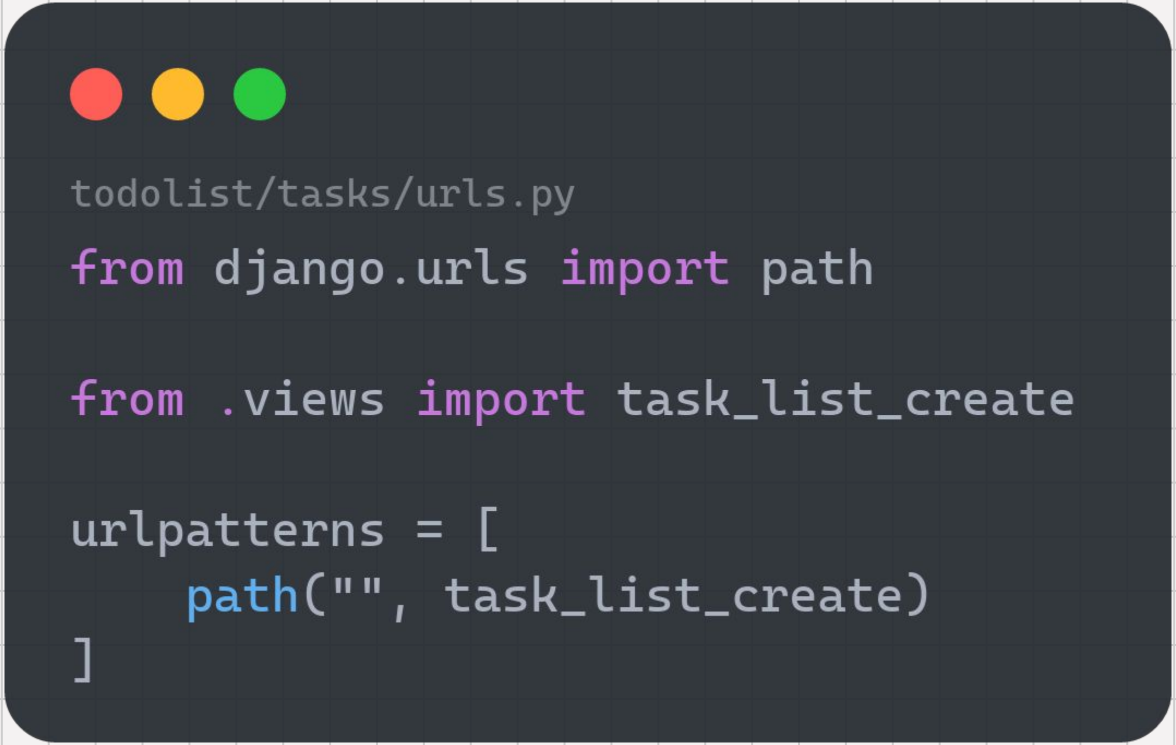
todolist/tasks/views.py

```python
from .models import Task
from .serializers import TaskSerializer
from rest_framework.response import Response
from rest_framework import status
from rest_framework.decorators import api_view


@api_view(["GET", "POST"])
def task_list_create(request):
    if request.method == "GET":
        tasks = Task.objects.all().order_by("-creation_date")
        serializer = TaskSerializer(tasks, many=True)
        return Response(serializer.data)

    if request.method == "POST":
        serializer = TaskSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Routing with URLs

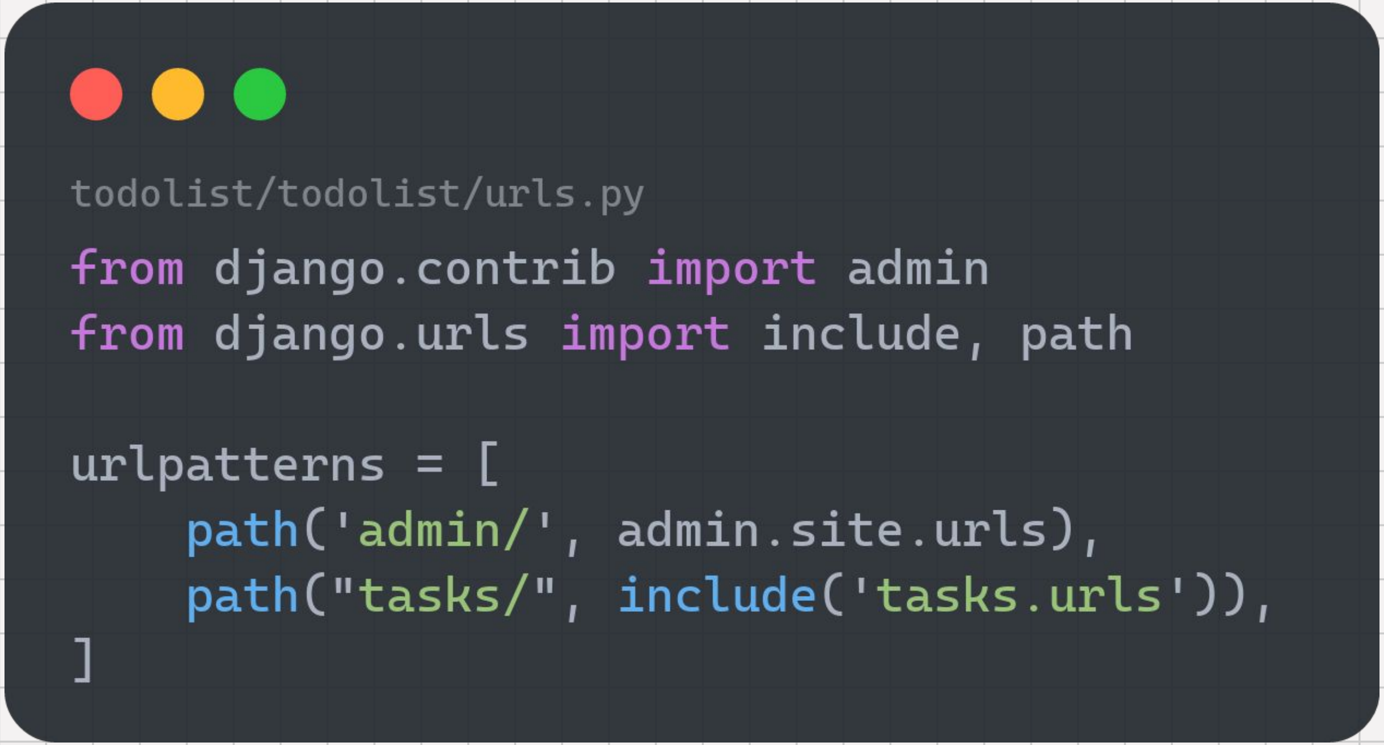
- URLs expose resources, not internal view structure.
- Routers generate consistent endpoint patterns automatically.
- Path structure reflects the API surface, not implementation details.
- Changing views should not require rewriting URLs.
- Predictable routing is a DRF design goal, not a side effect.

Routing with URLs



```
todolist/tasks/urls.py
from django.urls import path

from .views import task_list_create

urlpatterns = [
    path("", task_list_create)
]
```

Routing with URLs



todolist/todolist/urls.py

```python
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('admin/', admin.site.urls),
    path("tasks/", include('tasks.urls')),
]
```

Class-based API View

- Encapsulate behavior and state in a single abstraction.
- Map HTTP methods to class methods consistently.
- Encourage reuse through inheritance and composition.
- Integrate cleanly with authentication and permissions.
- Scale better as API complexity grows.

ViewSets

- Most APIs repeat the same operations on every resource.
- Treating each HTTP method as a separate view leads to duplication.
- ViewSets group related behaviors around a single resource.
- They enforce consistenc across list, retrieve, create, update, delete.
- This is convention replacing boilerplate, not hiding complexity.

The Same Thing, Simpler!

todolist/tasks/views.py

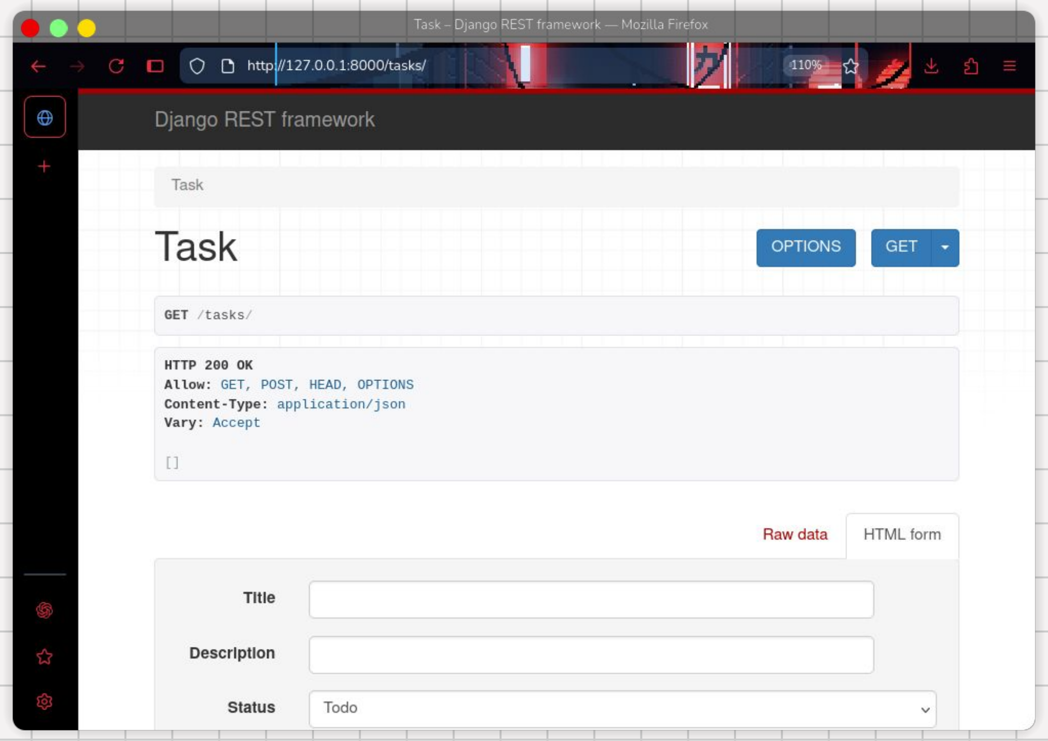
```python
from rest_framework.viewsets import ModelViewSet

from .models import Task
from .serializers import TaskSerializer


class TaskViewSet(ModelViewSet):
    queryset = Task.objects.all().order_by("-creation_date")
    serializer_class = TaskSerializer
```

Tweak the URLs

todolist/tasks/urls.py

```python
from django.urls import path

from .views import TaskViewSet



urlpatterns = [
    path("", TaskViewSet.as_view({"get": "list", "post": "create"})),
]
```

IT WORKS!



```
GET /tasks/
```

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[]
```

DRF Exceptions

- Exceptions represent API-level failure, not random Python crashes.
- DRF translates exceptions into structured HTTP responses.
- Status codes communicate intent; messages communicate cause.
- Validation errors are first-class, not special cases.
- Centralized exception handling keeps views clean and honest.

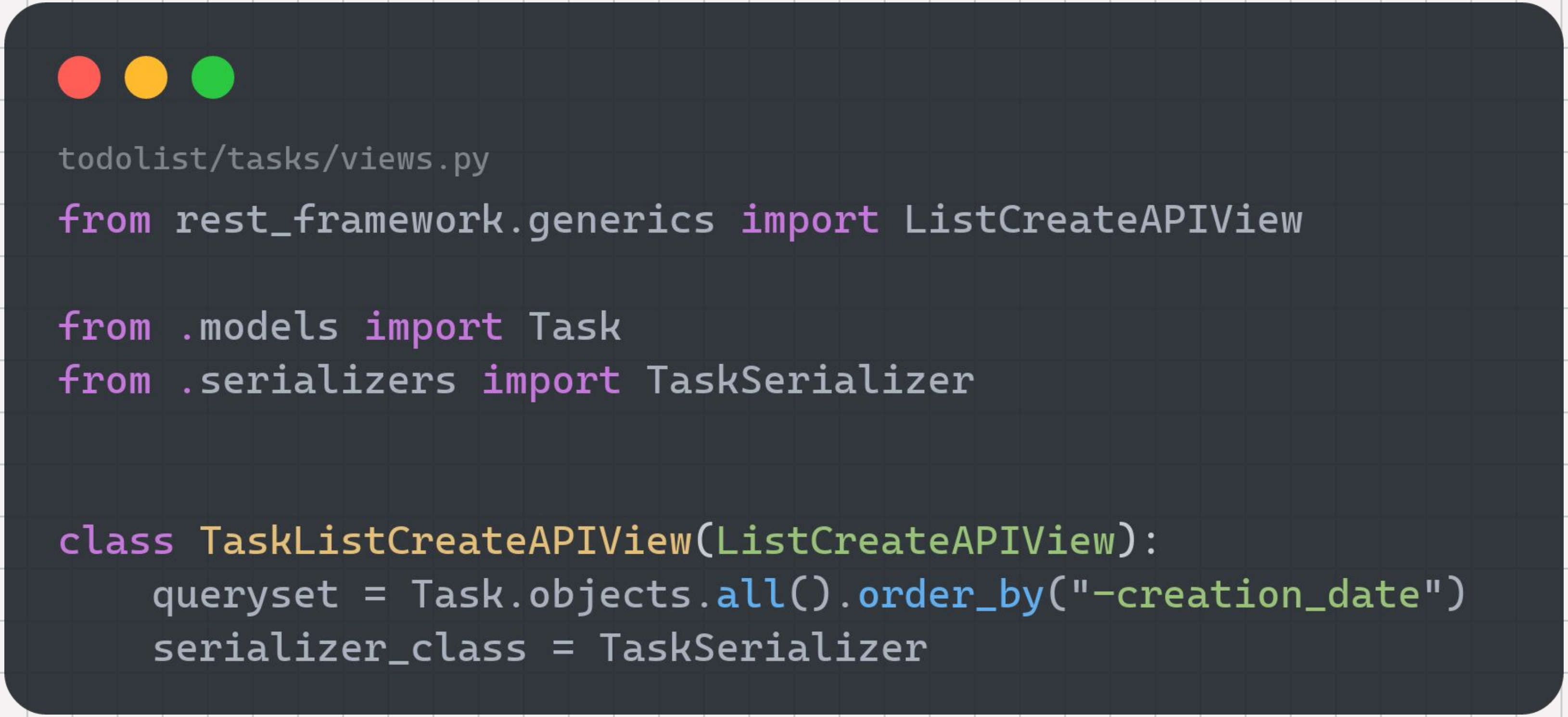
Mixin Design Pattern

- A mixin is a class that provides reusable behavior, not a complete object.
- It is meant to be combined with other classes, not used on its own.
- Each mixin should represent a single, focused capability.
- Composition through mixins avoids deep, rigid inheritance trees.
- Multiple inheritance is acceptable when responsibilities are small and clear.

Generic APIViews

- Encode common REST patterns instead of rewriting them.
- Provide structure without forcing a full ViewSet.
- Combine mixins to express intent: list, create, retrieve, update, delete.
- Keep behavior explicit while removing boilerplate.
- Use them when ViewSets feel excessive but `APIView` feels primitive.

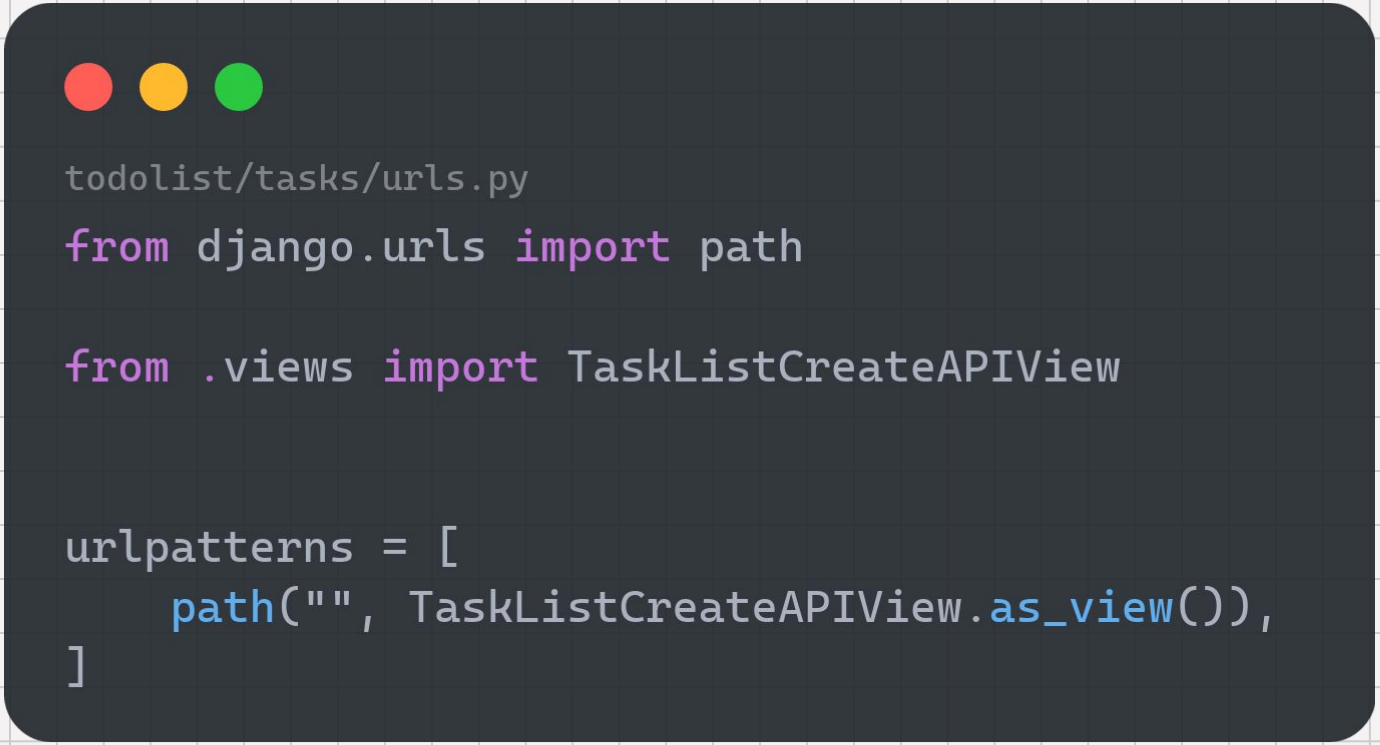
Even Simpler!



todolist/tasks/views.py

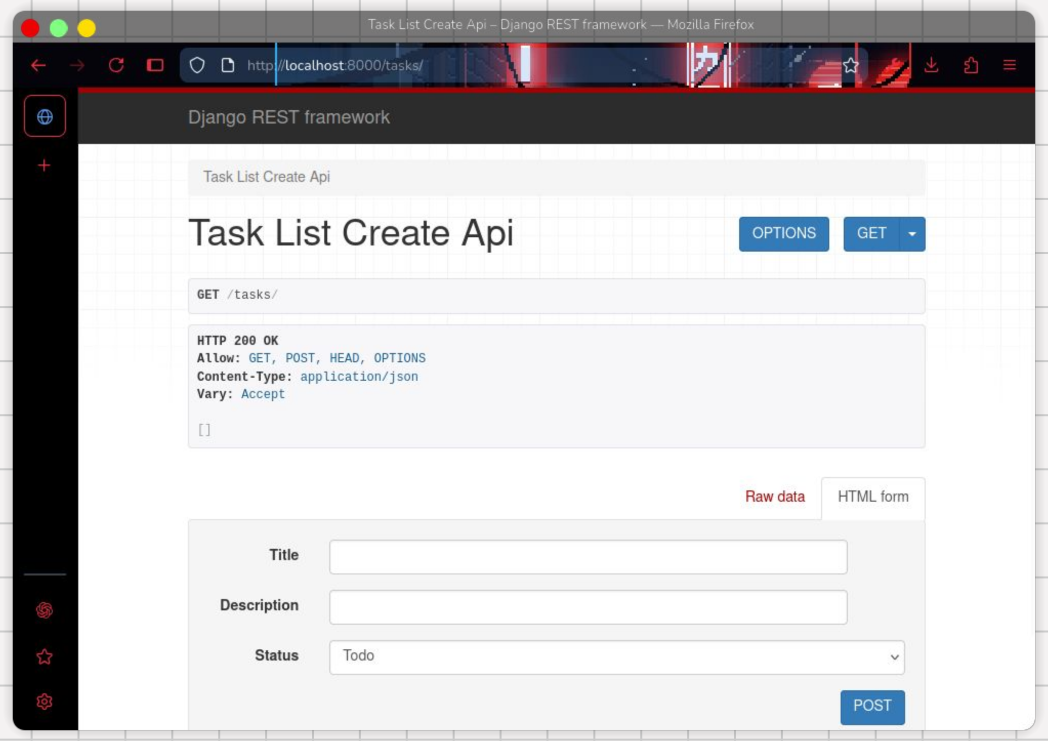
```python
from rest_framework.generics import ListCreateAPIView

from .models import Task
from .serializers import TaskSerializer


class TaskListCreateAPIView(ListCreateAPIView):
    queryset = Task.objects.all().order_by("-creation_date")
    serializer_class = TaskSerializer
```

Adjust The URLs…



```
todolist/tasks/urls.py
from django.urls import path

from .views import TaskListCreateAPIView


urlpatterns = [
    path("", TaskListCreateAPIView.as_view()),
]
```

Works Like Magic!


Task List Create Api – Django REST framework — Mozilla Firefox
http://localhost:8000/tasks/
Django REST framework
Task List Create Api
OPTIONS
GET
Raw data
HTML form
POST

Task List Create Api

GET /tasks/

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[]
```

Title	
Description	
Status	Todo

Pagination

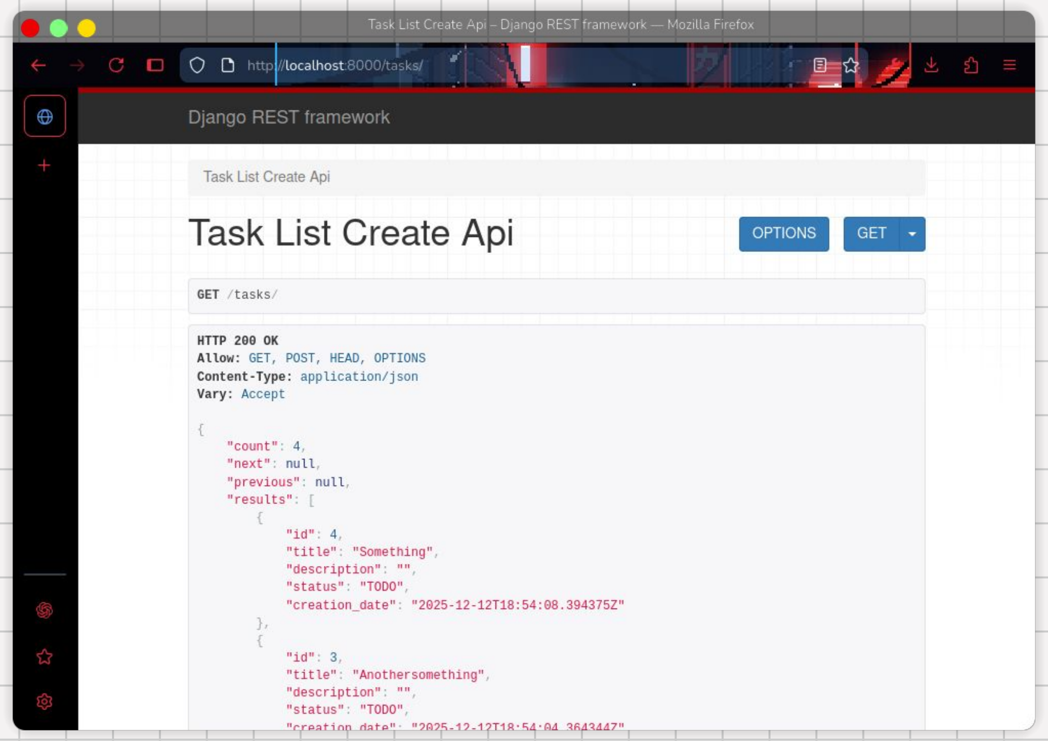
- Pagination controls how large result sets are exposed.
- It protects both the server and the client from excessive payloads.
- DRF provides pagination as a first-class concern, not an afterthought.
- Pagination behavior is configurable globally or per view.
- The response structure includes metadata, not just raw data.

View Changes A Bit…

todolist/tasks/views.py

```python
from rest_framework.generics import ListCreateAPIView
from rest_framework.pagination import PageNumberPagination

from .models import Task
from .serializers import TaskSerializer


class TaskPagination(PageNumberPagination):
    page_size = 10
    page_size_query_param = "page_size"
    max_page_size = 100


class TaskListCreateAPIView(ListCreateAPIView):
    queryset = Task.objects.all().order_by("-creation_date")
    serializer_class = TaskSerializer
    pagination_class = TaskPagination
```

Works Like a Charm!


Task List Create Api – Django REST framework — Mozilla Firefox
http://localhost:8000/tasks/
Django REST framework
Task List Create Api
OPTIONS
GET

Task List Create Api

```
GET /tasks/
```

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "count": 4,
    "next": null,
    "previous": null,
    "results": [
        {
            "id": 4,
            "title": "Something",
            "description": "",
            "status": "TODO",
            "creation_date": "2025-12-12T18:54:08.394375Z"
        },
        {
            "id": 3,
            "title": "Anothersomething",
            "description": "",
            "status": "TODO",
            "creation_date": "2025-12-12T18:54:04.364344Z"
```

Separation of Concerns

- Views coordinate requests, they do not implement business rules.
- Serializers handle validation and data representation.
- Models represent persistence, not API behavior.
- Authentication and permissions are enforced orthogonally.
- When these lines blur, DRF becomes unusable noise.

API Documentation

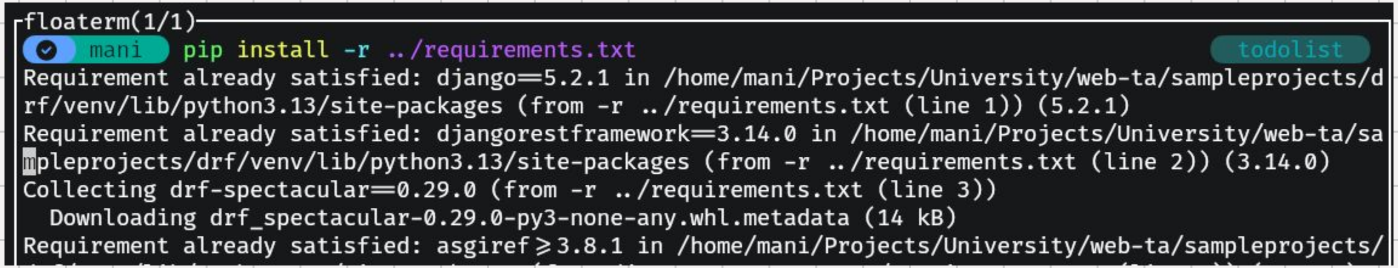
- Documentation is part of the API contract, not an optional extra.
- Endpoints must be discoverable without reading source code.
- Schemas describe structure, validation rules, and responses.
- Accurate docs prevent misuse better than comments ever will.
- An undocumented API is a private API pretending to be public.

OpenAPI & Swagger

- OpenAPI defines a machine-readable contract for the API.
- Swagger UI turns that contract into an interactive explorer.
- drf-spectacular generates OpenAPI schemas directly from DRF.
- Documentation stays synchronized with serializers and views.
- This is documentation that executes, not a PDF that stays still.

Install DRF Spectacular

```
Projects/University/web-ta/sampleprojects/drf/todolist/requirements.txt
django==5.2.1
djangorestframework==3.14.0
drf-spectacular==0.29.0
```

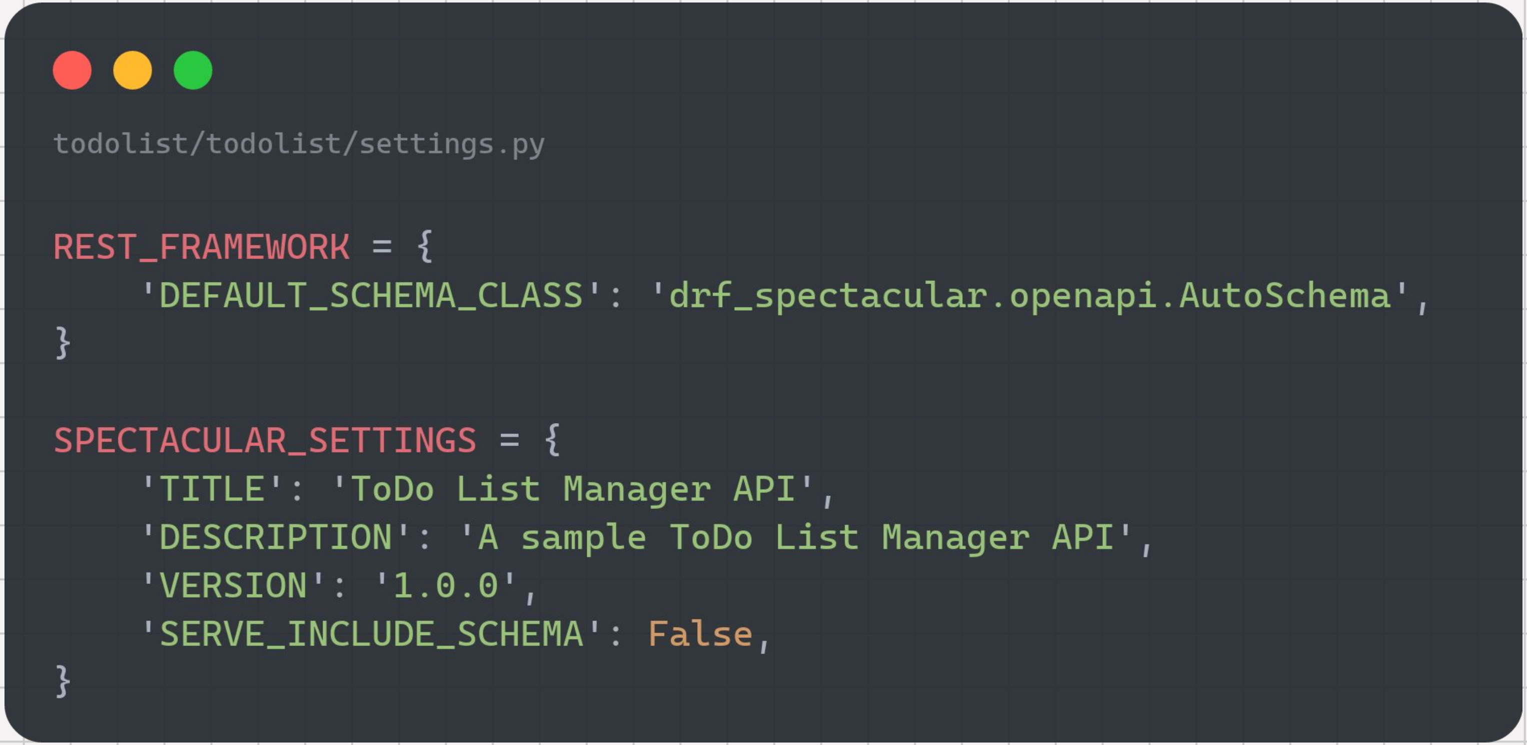


```
floaterm(1/1)
 mani  pip install -r ../requirements.txt                                                     todolist
Requirement already satisfied: django==5.2.1 in /home/mani/Projects/University/web-ta/sampleprojects/d
rf/venv/lib/python3.13/site-packages (from -r ../requirements.txt (line 1)) (5.2.1)
Requirement already satisfied: djangorestframework==3.14.0 in /home/mani/Projects/University/web-ta/sa
mpleprojects/drf/venv/lib/python3.13/site-packages (from -r ../requirements.txt (line 2)) (3.14.0)
Collecting drf-spectacular==0.29.0 (from -r ../requirements.txt (line 3))
  Downloading drf_spectacular-0.29.0-py3-none-any.whl.metadata (14 kB)
Requirement already satisfied: asgiref>=3.8.1 in /home/mani/Projects/University/web-ta/sampleprojects/
```

Add It To Installed Apps

```
todolist/todolist/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    # add drf spectacular here
    'drf_spectacular',

    'tasks'
]
```

Set It Up

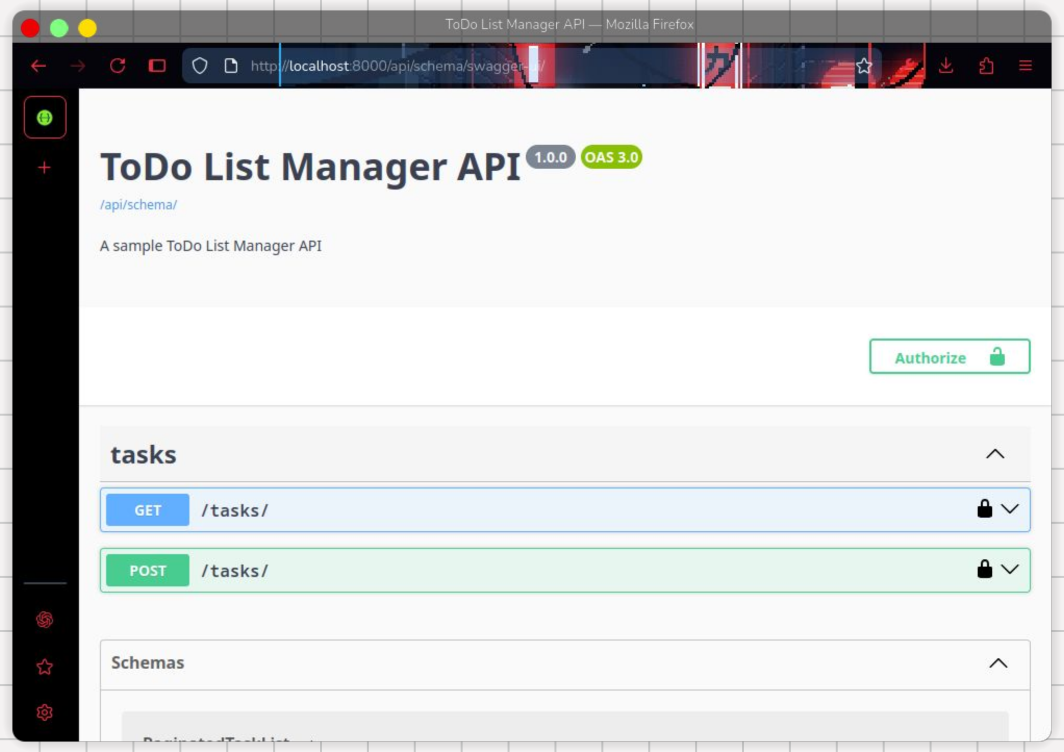


```
todolist/todolist/settings.py

REST_FRAMEWORK = {
    'DEFAULT_SCHEMA_CLASS': 'drf_spectacular.openapi.AutoSchema',
}

SPECTACULAR_SETTINGS = {
    'TITLE': 'ToDo List Manager API',
    'DESCRIPTION': 'A sample ToDo List Manager API',
    'VERSION': '1.0.0',
    'SERVE_INCLUDE_SCHEMA': False,
}
```

Map The URLs

```python
todolist/todolist/urls.py
from django.contrib import admin
from django.urls import include, path
from drf_spectacular.views import SpectacularAPIView, SpectacularRedocView, SpectacularSwaggerView

urlpatterns = [
    path('admin/', admin.site.urls),
    path("tasks/", include('tasks.urls')),
    # OpenAPI Schema
    path('api/schema/', SpectacularAPIView.as_view(), name='schema'),
    # Swagger UI
    path('api/schema/swagger-ui/', SpectacularSwaggerView.as_view(url_name='schema'), name='swagger-ui'),
]
```

MOM I HAVE AN API DOC!


ToDo List Manager API — Mozilla Firefox
http://localhost:8000/api/schema/swagger-ui/

ToDo List Manager API 1.0.0 OAS 3.0

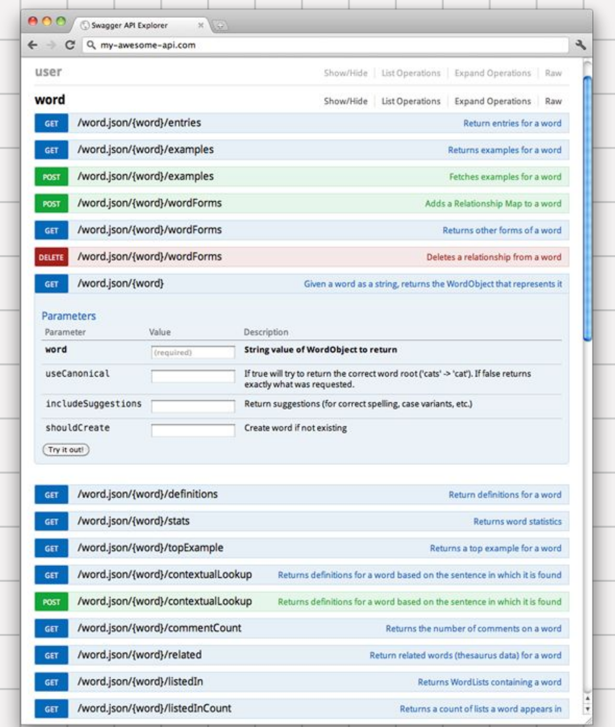
/api/schema/

A sample ToDo List Manager API

Authorize
tasks
GET /tasks/
POST /tasks/
Schemas

Extend Your Schemas!

- Auto-generated schemas fail on non-trivial endpoints.
- Complex validation rules are invisible without annotations.
- Multiple response shapes need explicit documentation.
- Query parameters are often lost without manual hints.
- `@extend_schema` restores accuracy to the API contract (remember Python decorators?)


Swagger API Explorer
my-awesome-api.com
user
Show/Hide | List Operations | Expand Operations | Raw
word
Show/Hide | List Operations | Expand Operations | Raw
Try it out!

GET	/word.json/{word}/entries	Return entries for a word
GET	/word.json/{word}/examples	Returns examples for a word
POST	/word.json/{word}/examples	Fetches examples for a word
POST	/word.json/{word}/wordForms	Adds a Relationship Map to a word
GET	/word.json/{word}/wordForms	Returns other forms of a word
DELETE	/word.json/{word}/wordForms	Deletes a relationship from a word
GET	/word.json/{word}	Given a word as a string, returns the WordObject that represents it

Parameters

Parameter	Value	Description
word	(required)	**String value of WordObject to return**
useCanonical		If true will try to return the correct word root ('cats' -> 'cat'). If false returns exactly what was requested.
includeSuggestions		Return suggestions (for correct spelling, case variants, etc.)
shouldCreate		Create word if not existing

GET	/word.json/{word}/definitions	Return definitions for a word
GET	/word.json/{word}/stats	Returns word statistics
GET	/word.json/{word}/topExample	Returns a top example for a word
GET	/word.json/{word}/contextualLookup	Returns definitions for a word based on the sentence in which it is found
POST	/word.json/{word}/contextualLookup	Returns definitions for a word based on the sentence in which it is found
GET	/word.json/{word}/commentCount	Returns the number of comments on a word
GET	/word.json/{word}/related	Return related words (thesaurus data) for a word
GET	/word.json/{word}/listedIn	Returns WordLists containing a word
GET	/word.json/{word}/listedInCount	Returns a count of lists a word appears in

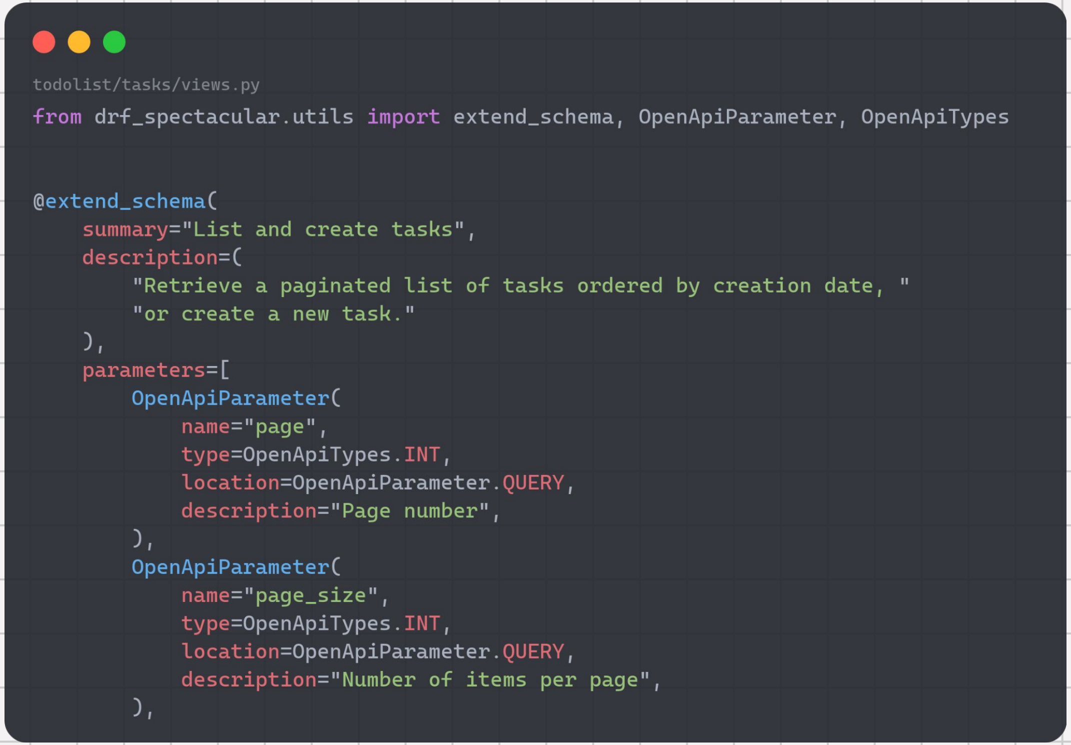
Parameters!

- `request`: serializer or schema for request body
- `responses`: mapping of status codes to serializers/schemas
- `parameters`: query/path/header parameters
- `examples`: request/response examples
- `description`: detailed operation description

Parameters! (cont.)

- `summary`: short operation summary
- `tags`: grouping in Swagger UI
- `deprecated`: mark endpoint as deprecated

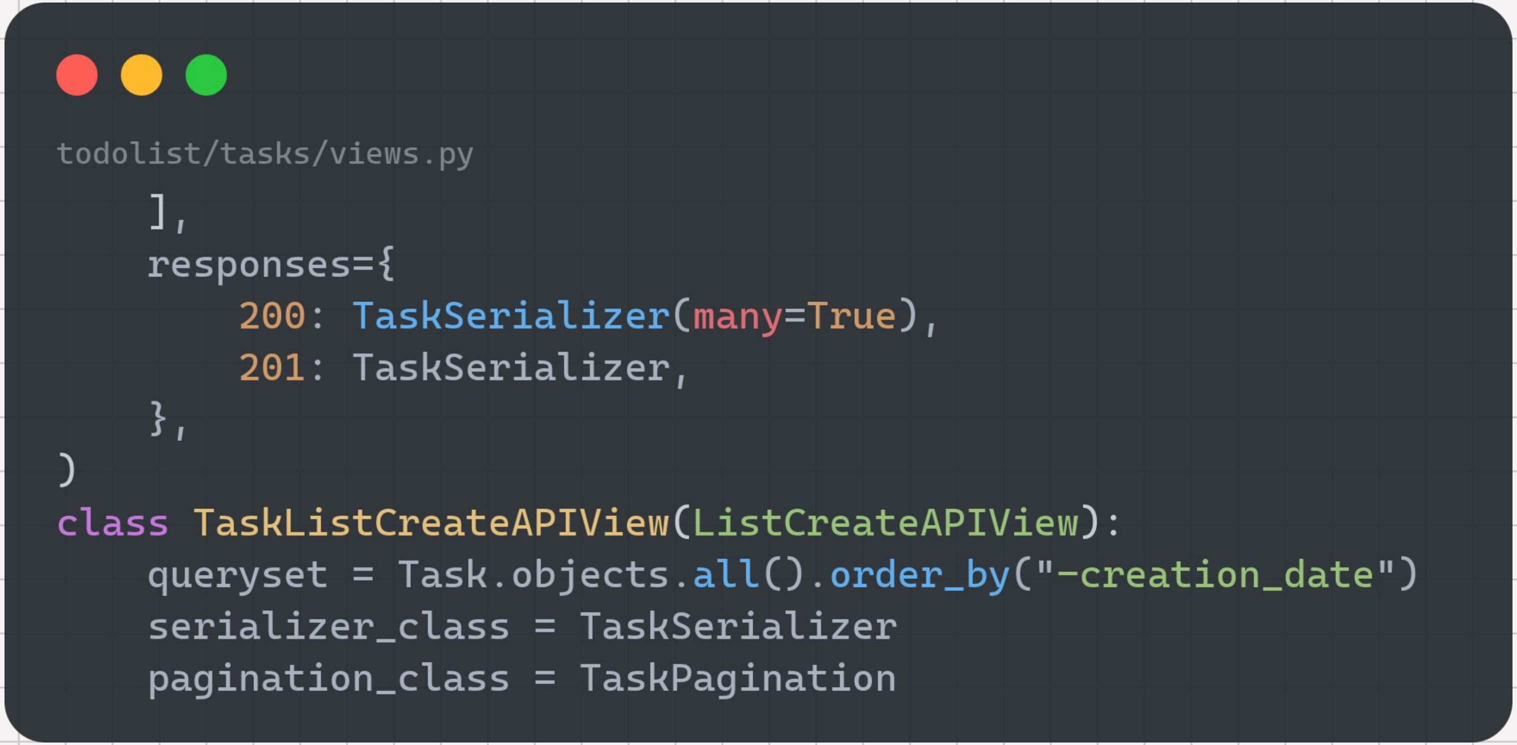
Extend The Doc!



todolist/tasks/views.py

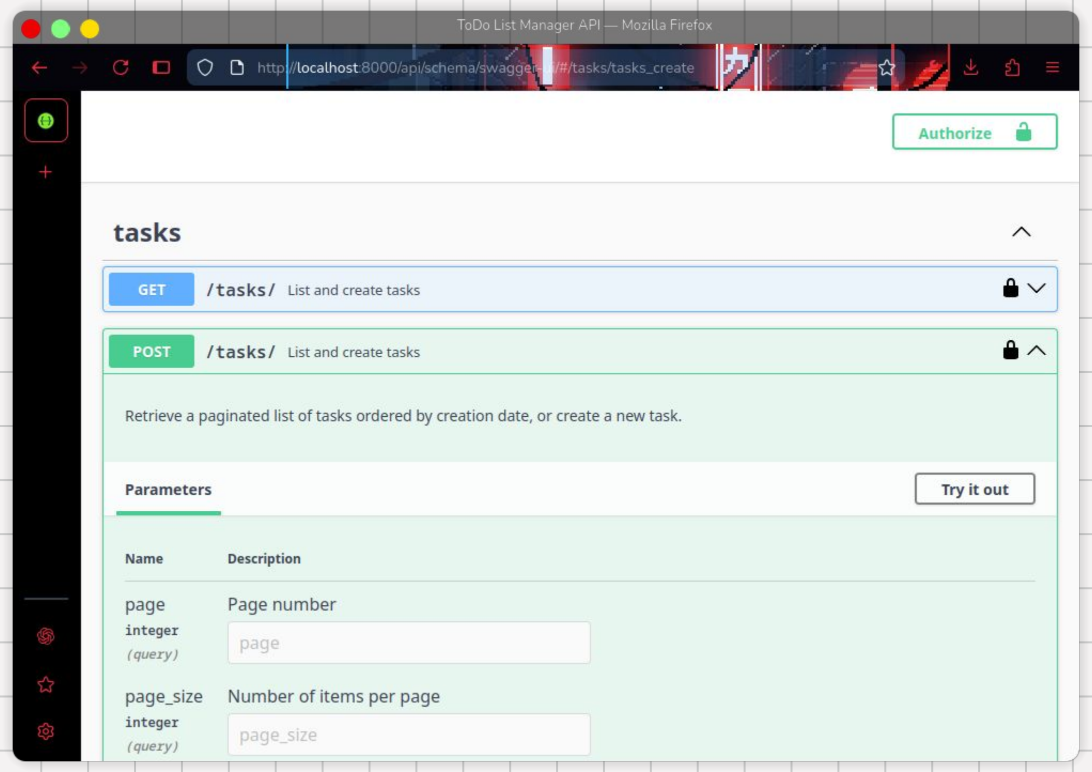
```python
from drf_spectacular.utils import extend_schema, OpenApiParameter, OpenApiTypes


@extend_schema(
    summary="List and create tasks",
    description=(
        "Retrieve a paginated list of tasks ordered by creation date, "
        "or create a new task."
    ),
    parameters=[
        OpenApiParameter(
            name="page",
            type=OpenApiTypes.INT,
            location=OpenApiParameter.QUERY,
            description="Page number",
        ),
        OpenApiParameter(
            name="page_size",
            type=OpenApiTypes.INT,
            location=OpenApiParameter.QUERY,
            description="Number of items per page",
        ),
```

Extend The Doc! (cont.)


todolist/tasks/views.py

```
    ],
    responses={
        200: TaskSerializer(many=True),
        201: TaskSerializer,
    },
)
class TaskListCreateAPIView(ListCreateAPIView):
    queryset = Task.objects.all().order_by("-creation_date")
    serializer_class = TaskSerializer
    pagination_class = TaskPagination
```

Voilà!


ToDo List Manager API — Mozilla Firefox

http://localhost:8000/api/schema/swagger-ui/#/tasks/tasks_create

Authorize

tasks

GET /tasks/ List and create tasks

POST /tasks/ List and create tasks

Retrieve a paginated list of tasks ordered by creation date, or create a new task.

Parameters

Try it out

Name	Description
page integer (query)	Page number
page_size integer (query)	Number of items per page

Class Activity Time!

What is the difference between `description` and `summary`?