Django

Test of Time, In Its Prime

Pascal Sun

UWA NLP-TLP Group

The University of Western Australia

November 29, 2024

Table of Contents

Why Django?

Origin Story: From Newsroom to Open Source

Figure: Django Logo

- Born in 2003 at the Lawrence Journal-World newspaper
- Created by Adrian Holovaty and Simon Willison
- Originally built to manage several news-oriented sites
- Released publicly in July 2005
- Named after Django Reinhardt, the jazz guitarist

Evolution Through the Years

Key Milestones

- 2005: First public release
- 2008: Django 1.0
- 2015: Django 1.8 LTS
- 2017: Django 2.0
- 2019: Django 3.0
- 2022: Django 4.0
- 2023: Django 4.2 LTS

Major Changes

- Native async support
- Modern security defaults
- Enhanced ORM capabilities
- Improved admin interface
- Type hints support

Django in the AI Era

Why Django remains relevant in AI development:

- **Robust ORM** for managing AI model metadata and results
- **Migration Management**: Manage the database changes
- **Admin interface** for monitoring data
- **Built-in Authentication system** for access control
- **RESTful Endpoints**: API side for ML Packages
- **Python**: ML/AI normally are Python First

Common AI Use Cases:

- ML model deployment backends
- AI service orchestration
- Data pipeline management
- Model monitoring dashboards

Framework Comparison

Feature	**Django**	**Flask**	**FastAPI**
Architecture	Full-stack	Micro	API-focused
Learning Curve	Moderate	Low	Low
Built-in Features	Many	Minimal	Moderate
Async Support	Partial	No	Full
Database ORM	Yes	No	No
Admin Interface	Yes	No	No
OpenAPI/Swagger	Plugin	Plugin	Built-in
Best For	Large apps	Small apps	APIs/ML
Community Support	Large	Have	Have
Ecosystem	Mature	Toy	Good

When to Choose Each Framework

Choose Django when:

- Building full-featured web applications
- Need built-in admin interface
- Complex database operations
- Authentication and authorization required

Choose Flask and FastAPI when:

- I do not see any reasons to choose them under our Full Stack Setup
- Maybe some edge cases

Django's Strengths

Technical Advantages

- Batteries included
- Excellent documentation
- Strong security defaults
- Mature ecosystem
- ORM abstraction

Business Advantages

- Rapid development
- Lower maintenance cost
- Large talent pool
- Proven scalability
- Long-term support

Notable Users:

- Instagram
- Mozilla
- Pinterest
- National Geographic
- NASA

Architecture

MVC before Frontend-Backend Separation

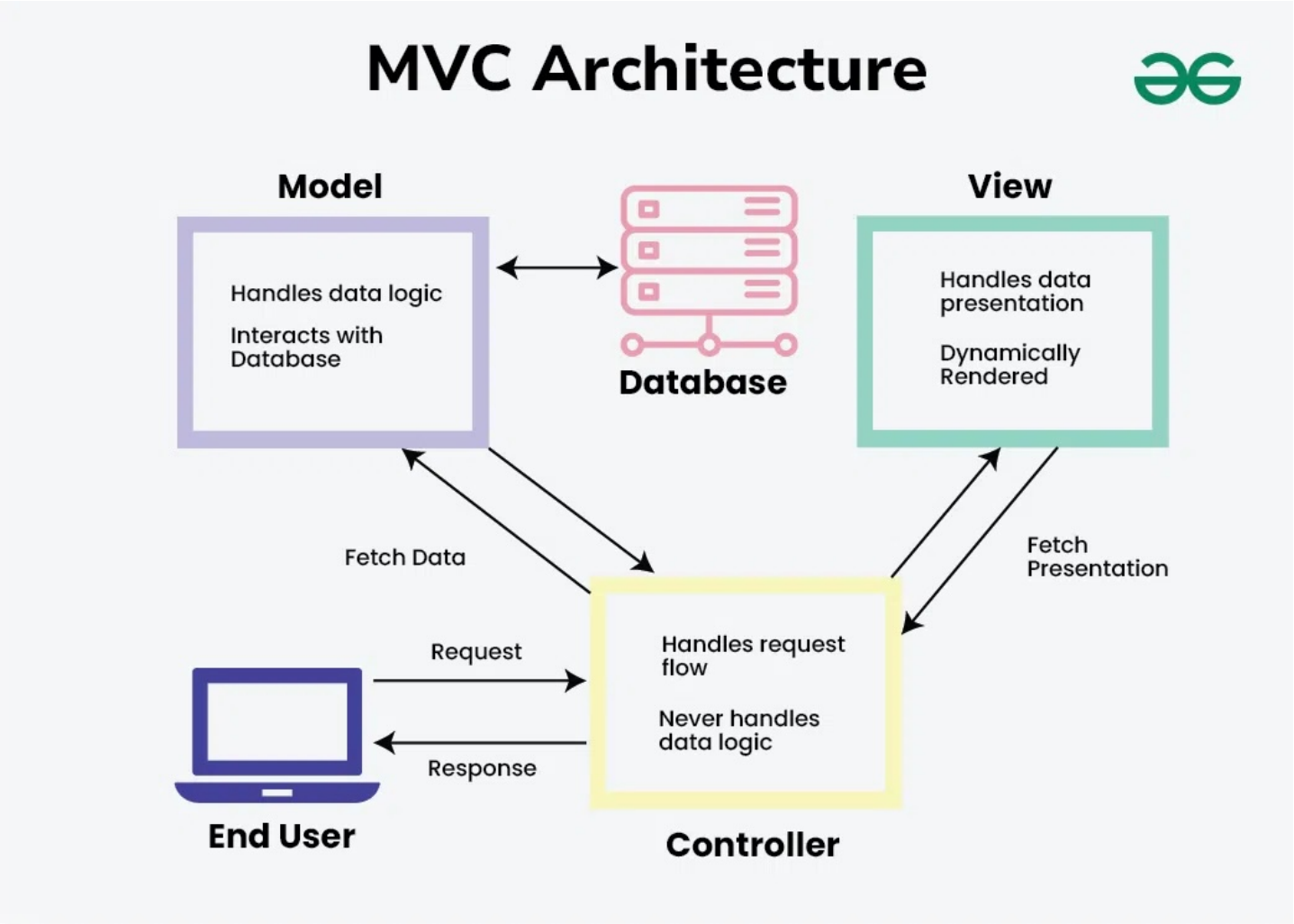

Figure: MVC architecture

Django's Architecture Overview

Model-Template-View (MTV)

Django's interpretation of the MVC pattern

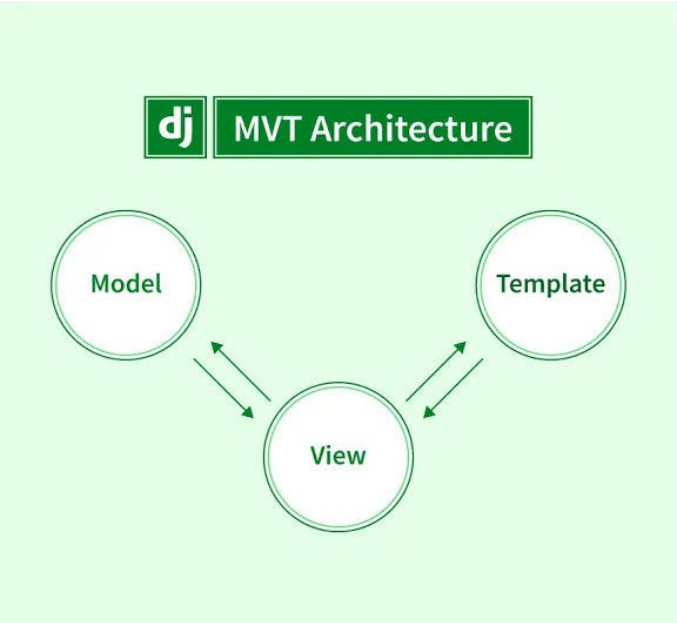

Figure: Django MTV Architecture

- **Model** = Data Structure (equivalent to MVC's Model)
- **Template** = Presentation Layer (equivalent to MVC's View)
- **View** = Business Logic (equivalent to MVC's Controller)

Request-Response Cycle

1. **Web Request** arrives at Django server
2. **URLs** (urls.py) route request to appropriate view
3. **View** (views.py) processes the request
4. **Model** (models.py) handles data operations
5. **Template** (*.html) renders the response
6. **Response** returns to user

$$\text{Browser} \rightarrow \text{URLs} \rightarrow \text{View} \rightarrow \text{Model} \leftrightarrow \text{Database}$$
$$\downarrow$$
$$\text{Response} \leftarrow \text{Template}$$

Request-Response Cycle (REST API)

1. **HTTP Request** arrives at Django server
2. **URLs** (urls.py) route request to appropriate view
3. **View** (views.py) processes the request
4. **Model** (models.py) handles data operations
5. **Serializer** converts model data to JSON
6. **JSONResponse** returns to client

$$\text{Client} \rightarrow \text{URLs} \rightarrow \text{View} \rightarrow \text{Model} \leftrightarrow \text{Database}$$
$$\downarrow$$
$$\text{JSONResponse} \leftarrow \text{Serializer}$$

Component Details

Models (models.py)

- Database table definitions
- Data relationships
- Data validation rules

Views (views.py)

- Request handling
- Business logic
- Data processing

Templates (*.html)

- HTML structure
- Dynamic content rendering
- Template inheritance

Project Structure

```
myproject/
   manage.py
   myproject/
       __init__.py
       settings.py
       urls.py
       asgi.py
       wsgi.py
   myapp/
       __init__.py
       admin.py
       apps.py
       models.py
       views.py
       serializer.py
       urls.py
       tests.py
       templates/
           myapp/
               *.html
```

Project Key Components Explained

- **manage.py**
  - Command-line utility for administrative tasks
- **settings.py**
  - Project configuration
  - Database settings
  - Installed apps
- **urls.py**
  - URL pattern definitions
  - Request routing
- **wsgi.py/asgi.py**
  - Web server gateway interface
  - Async server gateway interface

Middleware Architecture

- Framework that processes requests/responses
- Executes before/after views
- Common use cases:
  - Authentication
  - Session handling
  - Security features
  - CORS headers
  - Request processing
- Executed in order defined in settings.py

Design Principles

Django's Core Philosophy

- **DRY** (Don't Repeat Yourself)
  - Reusable components
  - Template inheritance
- **Loose Coupling**
  - Independent components
  - Modular design
- **Explicit is better than Implicit**
  - Clear URL routing
  - Obvious data flow

Try first

Development Environment Setup

Prerequisites

- Python 3.8+ installed
- pip (Python package manager)
- Virtual environment tool

```
# Create virtual environment
python -m venv venv

# Activate virtual environment
# On Windows:
venv\Scripts\activate
# On Unix or MacOS:
source venv/bin/activate

# Install Django
pip install django

```

Creating Your First Project

```
# Create a new Django project
django-admin startproject myproject
cd myproject

# Create a new app
python manage.py startapp myapp

```

Important

Add your app to INSTALLED_APPS in settings.py:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    ...
    'myapp',  # Add this line
]

```

Initial Configuration

```python
# settings.py
# Database configuration (PostgreSQL)
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'your_db_name',
        'USER': 'your_db_user',
        'PASSWORD': 'your_db_password',
        'HOST': 'localhost',  # or your DB host
        'PORT': '5432',       # default PostgreSQL port
    }
}
```

Prerequisites

- Install psycopg2: `pip install psycopg2-binary`

Running the Development Server

```
# Apply migrations, these are init migrations from Django
python manage.py migrate

# Create superuser (admin)
python manage.py createsuperuser
# put the email address and password in, you then can login

# Run development server
python manage.py runserver

```

Access Points

- Development server: `http://127.0.0.1:8000/`
- Admin interface: `http://127.0.0.1:8000/admin/`

Project Structure Verification

```
myproject/
 manage.py                  # Command-line utility
 myproject/                 # Project container
      __init__.py
      settings.py           # Project settings
      urls.py               # URL declarations
      asgi.py               # ASGI deployment
      wsgi.py               # WSGI deployment
 myapp/                     # Your application
       __init__.py
       admin.py             # Admin interface
       apps.py              # App configuration
       models.py            # Data models
       views.py             # View functions
       urls.py              # URL patterns
       tests.py             # Unit tests

```

Common Issues & Solutions

Port in Use `python manage.py runserver 8001`

Migration Error `python manage.py migrate --run-syncdb`

Static Files `python manage.py collectstatic`

Requirements Create `requirements.txt`:

```
pip freeze > requirements.txt
# or mannually create one and put all packages in

```

Debug Mode

Remember: Never use `DEBUG = True` in production!

Core Components

Project vs App

Project

- Configuration
- Global settings

App

- Specific functionality
- Reusable component
- Pluggable design
- Focused purpose

App Design Principles

- **Single Responsibility**: One app, one core function
- **Loose Coupling**: Minimal dependencies
- **High Cohesion**: Related features together
- **Reusability**: Portable across projects

Common App Examples

- Blog app for content management
- Auth app for user handling
- Cart app for e-commerce
- API app for REST endpoints

Models Overview

Definition

Models define your database structure and business logic

```python
from django.db import models

class Article(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.title

```

Model Relationships

```
    class Author(models.Model):
        name = models.CharField(max_length=100)
        email = models.EmailField()

    class Article(models.Model):
        author = models.ForeignKey(
            Author,
            on_delete=models.CASCADE
        )
        categories = models.ManyToManyField('Category')

    class Category(models.Model):
        name = models.CharField(max_length=100)

```

Views Basics

Function-Based Views

```
1        from django.shortcuts import render
2
3        def article_list(request):
4            articles = Article.objects.all()
5            return render(request,
6                          'articles/list.html',
7                          {'articles': articles})
8
```

Views Basics

Class-Based Views

```
from django.views.generic import ListView

class ArticleList(ListView):
    model = Article
    template_name = 'articles/list.html'
    context_object_name = 'articles'

```

URL Configuration

Project URLs (urls.py)

```
        from django.urls import path, include

        urlpatterns = [
            path('admin/', admin.site.urls),
            path('articles/', include('articles.urls')),
        ]

```

URL Configuration

App URLs (articles/urls.py)

```
from django.urls import path
from myapp.views import ArticleList, article_detail

app_name = 'articles'
urlpatterns = [
    path('', ArticleList.as_view(),
         name='list'),
    path('<int:pk>/', article_detail,
         name='detail'),
]

```

Template System

```
<!-- base.html -->
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    {% block content %}
    {% endblock %}
</body>
</html>

<!-- article_list.html -->
{% extends "base.html" %}

{% block content %}
{% for article in articles %}
    <h2>{{ article.title }}</h2>
    <p>{{ article.content }}</p>
{% endfor %}
{% endblock %}

```

Template Best Practices

- Use template inheritance
- Keep logic in views, not templates
- Use template tags for complex logic
- Structure templates hierarchically

Template Organization

```
templates/
    base.html
    articles/
        list.html
        detail.html
```

Django Admin Example

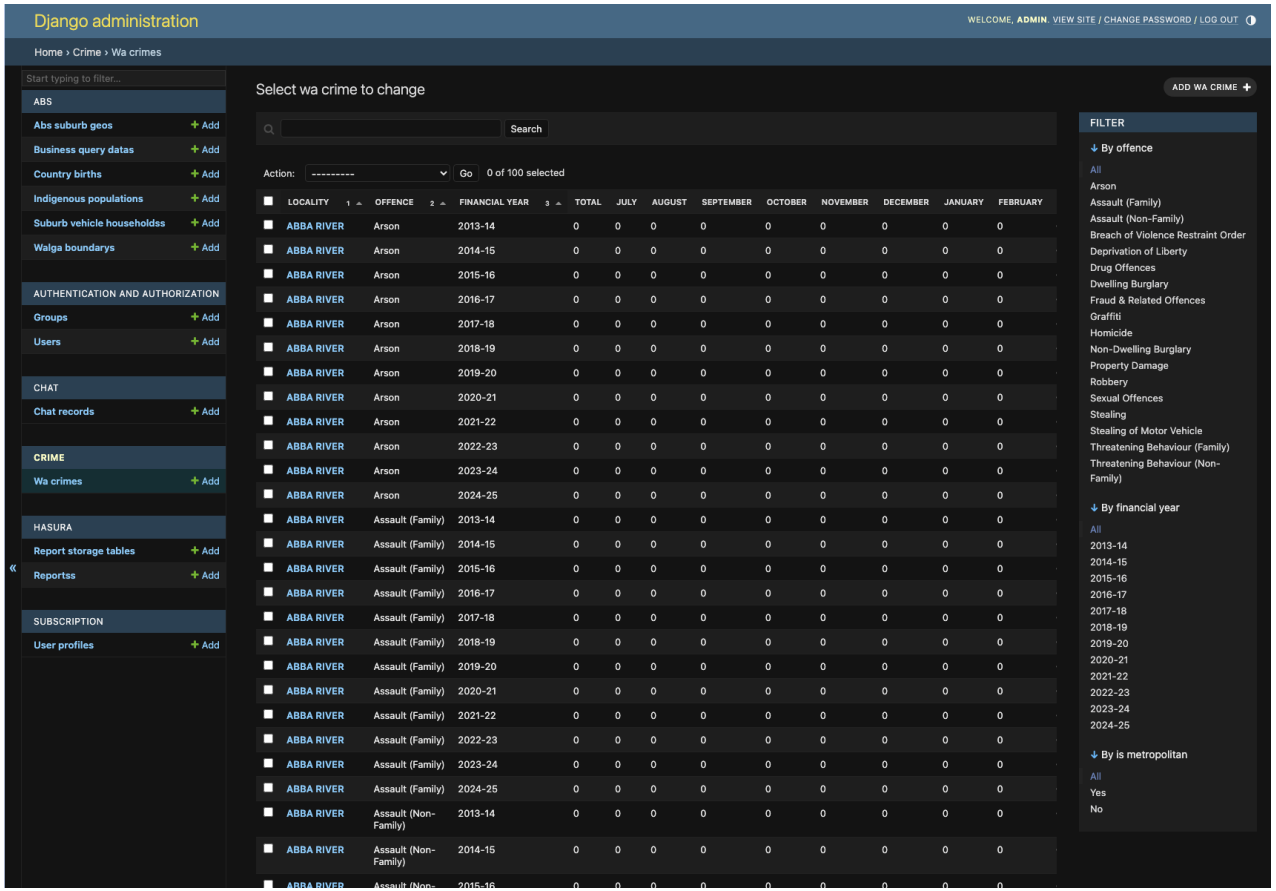

Figure: Django Admin Example List

Django Admin Example

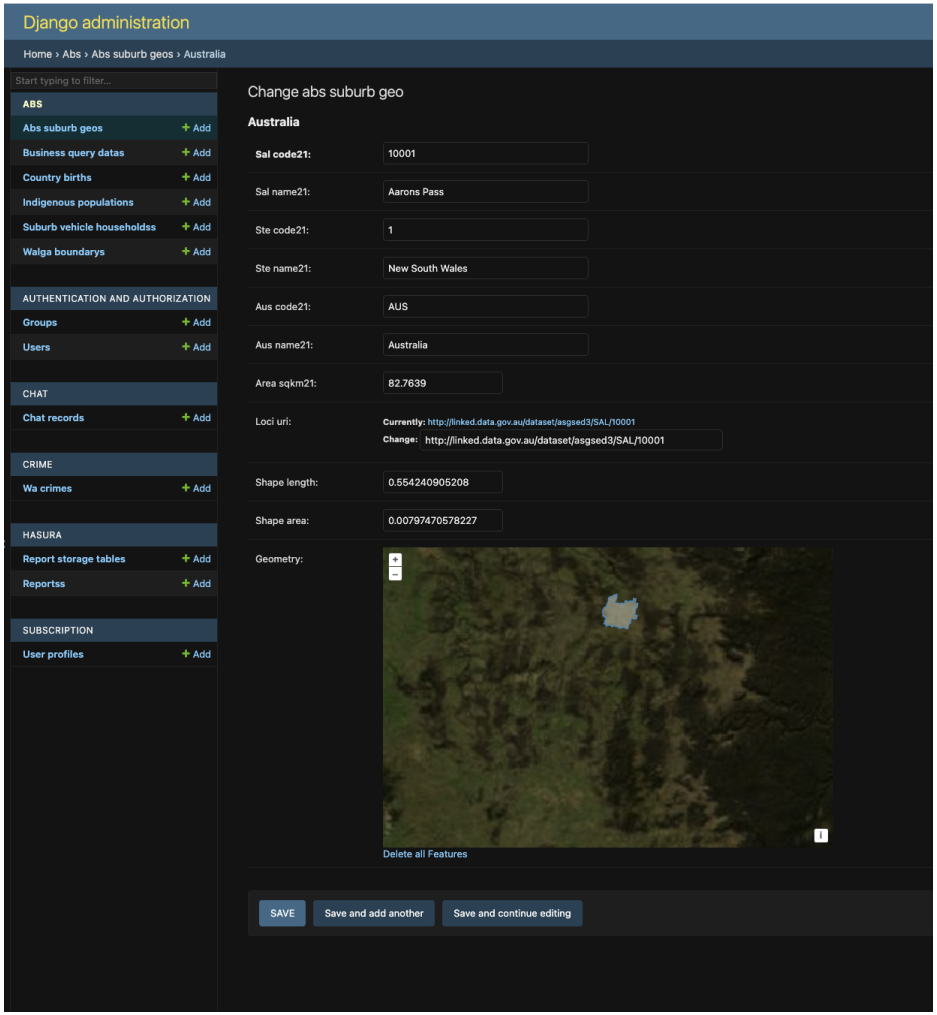

Figure: Django Admin Example Update

Django Admin Overview

Key Features

- Automatic CRUD interface
- User authentication & permissions
- Model registration & customization
- List display & filtering
- Search functionality

Quick Setup

```
# admin.py
from django.contrib import admin
from myapp.models import Article

admin.site.register(Article)

```

Custom Admin Classes

```python
@admin.register(Article)
class ArticleAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'status',
                    'created_at')
    list_filter = ('status', 'created_at')
    search_fields = ('title', 'content')
    date_hierarchy = 'created_at'
    ordering = ('-created_at',)

    # Custom fields layout
    fieldsets = (
        ('Content', {
            'fields': ('title', 'content')
        }),
        ('Metadata', {
            'fields': ('author', 'status')
        })
    )

```

Django Import-Export

Setup

```
pip install django-import-export
```

```
# settings.py
INSTALLED_APPS = [
    ...
    'import_export',
]
```

Django Import-Export

```
from import_export import resources
from import_export.admin import ImportExportModelAdmin

class ArticleResource(resources.ModelResource):
    class Meta:
        model = Article
        fields = ('id', 'title', 'content',
                  'author', 'status')

@admin.register(Article)
class ArticleAdmin(ImportExportModelAdmin):
    resource_class = ArticleResource

```

Advanced Filtering

```python
from django.contrib import admin

class PublishedFilter(admin.SimpleListFilter):
    title = 'publication status'
    parameter_name = 'status'

    def lookups(self, request, model_admin):
        return (
            ('published', 'Published'),
            ('draft', 'Draft'),
        )

    def queryset(self, request, queryset):
        if self.value() == 'published':
            return queryset.filter(status='published')
        if self.value() == 'draft':
            return queryset.filter(status='draft')

@admin.register(Article)
class ArticleAdmin(admin.ModelAdmin):
    list_filter = (PublishedFilter,)

```

Admin Actions

```
    @admin.register(Article)
    class ArticleAdmin(admin.ModelAdmin):
        actions = ['make_published', 'make_draft']

        @admin.action(description='Mark selected as published')
        def make_published(self, request, queryset):
            queryset.update(status='published')
            self.message_user(request,
                              'Articles marked as published')

        @admin.action(description='Mark selected as draft')
        def make_draft(self, request, queryset):
            queryset.update(status='draft')
            self.message_user(request,
                              'Articles marked as draft')

```

Admin Security Best Practices

Important Security Measures

- Change admin URL from /admin/
- Use strong passwords
- Limit staff user permissions
- Enable HTTPS
- Set up proper user groups

```
# urls.py
path('custom-admin/', admin.site.urls),

# settings.py
ADMIN_URL = 'custom-admin/'
SECURE_SSL_REDIRECT = True  # Force HTTPS

```

RESTful

Django REST Framework Setup

Installation

```
        pip install djangorestframework
        pip install drf-yasg  # for Swagger/OpenAPI

```

Django REST Framework Setup

Configuration

```
# settings.py
INSTALLED_APPS = [
    ...
    'rest_framework',
    'drf_yasg',
]

REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
        'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 10,
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',
        'rest_framework.authentication.SessionAuthentication',
    ],
}

```

Serializers

```python
# serializers.py
from rest_framework import serializers
from myapp.models import Article

class ArticleSerializer(serializers.ModelSerializer):
    author_name = serializers.CharField(
        source='author.username', read_only=True)

    class Meta:
        model = Article
        fields = ['id', 'title', 'content',
                  'author_name', 'created_at']
        read_only_fields = ['created_at']

    def validate_title(self, value):
        if len(value) < 10:
            raise serializers.ValidationError(
                "Title must be at least 10 characters")
        return value

```

ViewSets

```python
# views.py
from rest_framework import viewsets
from rest_framework.permissions import IsAuthenticated
from rest_framework.decorators import action
from rest_framework.response import Response

class ArticleViewSet(viewsets.ModelViewSet):
    queryset = Article.objects.all()
    serializer_class = ArticleSerializer
    permission_classes = [IsAuthenticated]
    def get_queryset(self):
        queryset = Article.objects.all()
        status = self.request.query_params.get('status')
        if status:
            queryset = queryset.filter(status=status)
        return queryset

```

ViewSets

```python
# views.py
from rest_framework import viewsets
from rest_framework.permissions import IsAuthenticated
from rest_framework.decorators import action
from rest_framework.response import Response

class ArticleViewSet(viewsets.ModelViewSet):
    queryset = Article.objects.all()
    serializer_class = ArticleSerializer
    permission_classes = [IsAuthenticated]

    ...

    @action(detail=True, methods=['post'])
    def publish(self, request, pk=None):
        article = self.get_object()
        article.status = 'published'
        article.save()
        return Response({'status': 'published'})
```

URL Routing

```
# urls.py
from rest_framework.routers import DefaultRouter
from django.urls import path, include

router = DefaultRouter()
router.register(r'articles', ArticleViewSet)

urlpatterns = [
    path('api/', include(router.urls)),
]

```

Swagger/OpenAPI Integration

```python
# urls.py
from drf_yasg.views import get_schema_view
from drf_yasg import openapi

schema_view = get_schema_view(
    openapi.Info(
        title="API Documentation",
        default_version='v1',
        description="API documentation",
        terms_of_service="https://www.example.com/terms/",
        contact=openapi.Contact(email="contact@example.com"),
        license=openapi.License(name="BSD License"),
    ),
    public=True,
)
urlpatterns += [
    path('swagger/', schema_view.with_ui(
        'swagger', cache_timeout=0),
        name='schema-swagger-ui'),
    path('redoc/', schema_view.with_ui(
        'redoc', cache_timeout=0),
        name='schema-redoc'),
]
```

Redoc

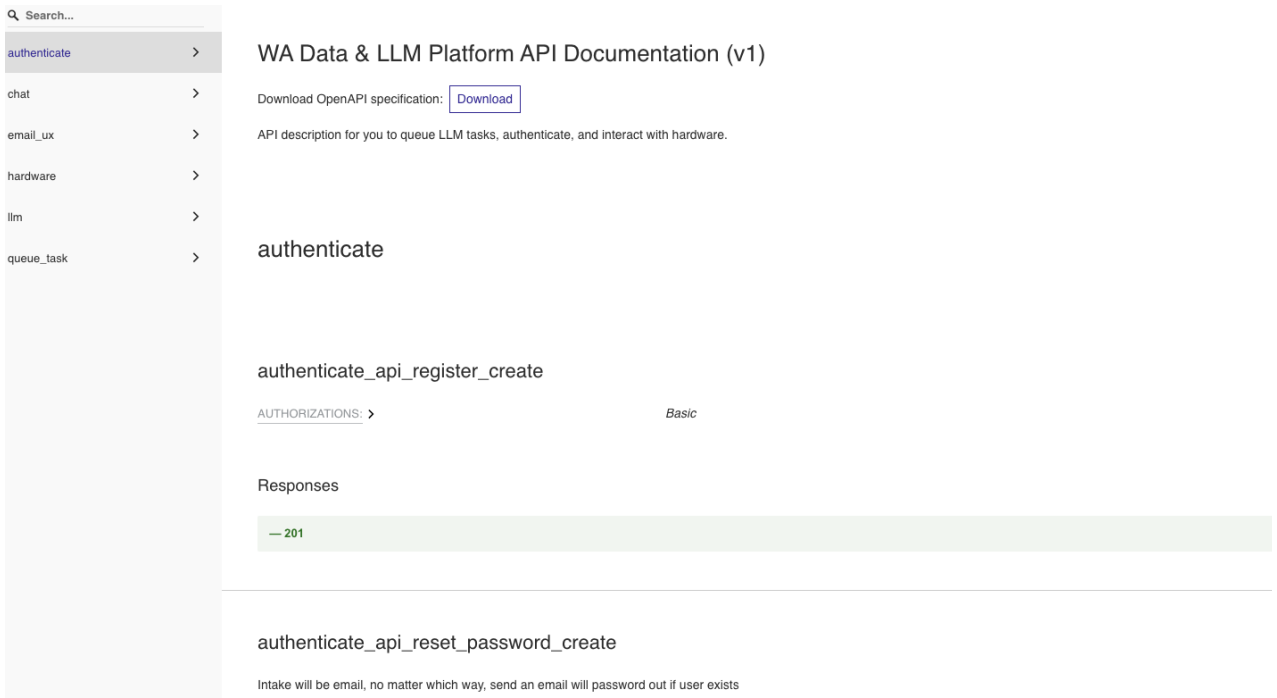

Figure: Redoc API documentation page

Swagger

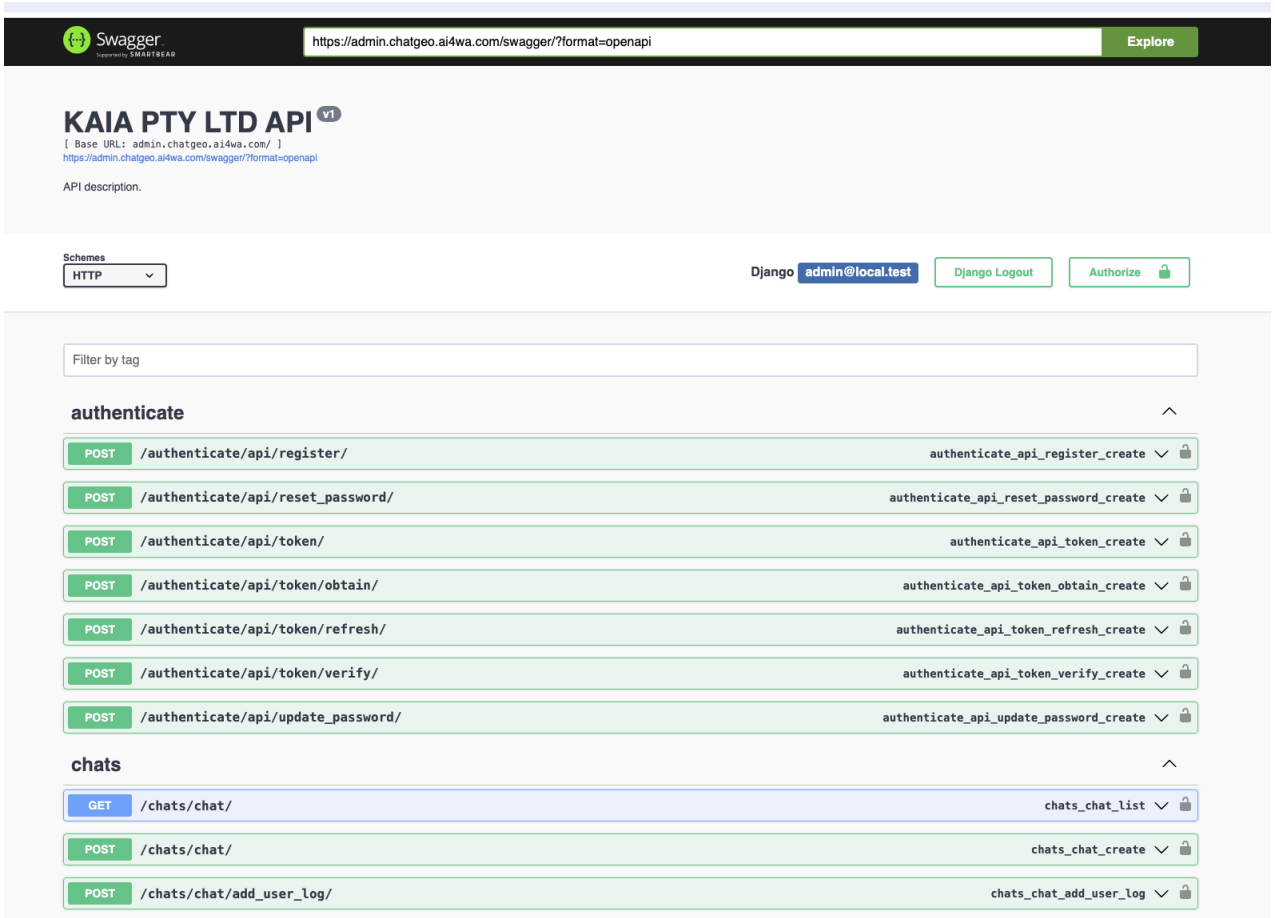

Figure: Swagger API Documentation page

API Versioning

```python
# settings.py
REST_FRAMEWORK = {
    'DEFAULT_VERSIONING_CLASS':
        'rest_framework.versioning.URLPathVersioning',
    'DEFAULT_VERSION': 'v1',
    'ALLOWED_VERSIONS': ['v1', 'v2'],
}

# views.py
class ArticleViewSet(viewsets.ModelViewSet):
    def get_serializer_class(self):
        if self.request.version == 'v2':
            return ArticleSerializerV2
        return ArticleSerializer

```

API Best Practices

- **Authentication & Security**
  - Use Token or JWT authentication
  - Implement proper permissions
  - Rate limiting/throttling
- **Performance**
  - Use pagination
  - Optimize queries
  - Cache responses
- **Documentation**
  - Maintain OpenAPI/Swagger docs
  - Document all endpoints
  - Include example requests/responses

Management Command

Django Management Commands Overview

What are Management Commands?

- Command-line utilities for Django projects
- Accessible via `python manage.py`
- Can be built-in or custom
- Automate routine tasks

Common Built-in Commands

- `migrate`: Database migrations
- `makemigrations`: Create migrations
- `createsuperuser`: Create admin user
- `shell`: Interactive Python shell
- `runserver`: Development server

Creating Custom Commands

File Structure

```
myapp/
    management/
        __init__.py
        commands/
            __init__.py
            my_command.py
```

```python
# my_command.py
from django.core.management.base import BaseCommand

class Command(BaseCommand):
    help = 'Description of your command'

    def handle(self, *args, **options):
        self.stdout.write('Command executed')
```

Command Arguments

```python
class Command(BaseCommand):
    help = 'Closes polls older than given days'

    def add_arguments(self, parser):
        # Positional arguments
        parser.add_argument('days', type=int)

        # Named (optional) arguments
        parser.add_argument(
            '--delete',
            action='store_true',
            help='Delete instead of close',
        )

    def handle(self, *args, **options):
        days = options['days']
        if options['delete']:
            # Delete logic
            self.stdout.write(
                'Deleting old polls...')
        else:
            # Close logic
            self.stdout.write('Closing old polls...')
```

Practical Example: Data Import

```
import csv
...
class Command(BaseCommand):
    help = 'Import products from CSV file'
    def add_arguments(self, parser):
        parser.add_argument('csv_file', type=str)
    def handle(self, *args, **options):
        file_path = options['csv_file']
        counter = 0
        with open(file_path) as file:
            reader = csv.DictReader(file)
            for row in reader:
                Product.objects.create(
                    name=row['name'],
                    price=float(row['price']),
                    description=row['description']
                )
                counter += 1
        self.stdout.write(self.style.SUCCESS(f'Successfully imported {counter} products'))

```

Best Practices

- **Naming Conventions**
  - Use clear, descriptive names
  - Follow verb_noun pattern
- **Error Handling**
  - Catch and handle exceptions
  - Provide clear error messages
- **Progress Feedback**
  - Show progress for long operations
  - Use appropriate styling
- **Documentation**
  - Clear help text
  - Document arguments
  - Usage examples

Common Use Cases

1. **Data Management**
   - Import/Export data
   - Data cleanup
   - Database maintenance
2. **System Tasks**
   - Backup creation
   - Cache clearing
   - System checks
3. **Scheduled Operations**
   - Regular cleanups
   - Report generation
   - Email notifications

Testing

Django Testing Overview

Testing Framework

- Built on Python's unittest library
- Extends TestCase for Django features
- Supports unit and integration tests
- Includes test client for HTTP testing

```
# Running Tests
python manage.py test
python manage.py test myapp
python manage.py test myapp.tests.test_models

```

Test Case Structure

```python
from django.test import TestCase
from .models import Article

class ArticleTests(TestCase):
    def setUp(self):
        # Runs before each test method
        self.article = Article.objects.create(
            title="Test Article",
            content="Test Content"
        )

    def test_article_creation(self):
        # Test case method
        self.assertEqual(
            self.article.title, "Test Article"
        )

    def tearDown(self):
        # Cleanup after each test
        pass

```

Model Testing

```
    class ArticleModelTests(TestCase):
        def setUp(self):
            ...
        def test_string_representation(self):
            self.assertEqual(
                str(self.article),
                "Test Article"
            )

        def test_publish_method(self):
            self.article.publish()
            self.assertEqual(
                self.article.status,
                "published"
            )
        ...

```

View Testing

```python
from django.urls import reverse
from django.test import Client

class ArticleViewTests(TestCase):
    def setUp(self):
        self.client = Client()
        self.article = Article.objects.create(
            title="Test Article"
        )

    def test_article_list_view(self):
        response = self.client.get(
            reverse('article_list')
        )
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(
            response,
            'article_list.html'
        )
        self.assertContains(
            response,
            "Test Article"
        )
```

API Testing

```python
from rest_framework.test import APITestCase
from rest_framework import status

class ArticleAPITests(APITestCase):
    def setUp(self):
        self.article_data = {
            'title': 'API Test Article',
            'content': 'Test Content'
        }

    def test_create_article(self):
        response = self.client.post(
            reverse('article-list'),
            self.article_data,
            format='json'
        )
        self.assertEqual(
            response.status_code,
            status.HTTP_201_CREATED
        )
        self.assertEqual(
            Article.objects.count(),
            1)
```

Test Coverage

Setting Up Coverage

```
pip install coverage
coverage run manage.py test
coverage report
coverage html  # detailed HTML report

```

Coverage Configuration

```
# .coveragerc
[run]
source = myapp
omit =
    */migrations/*
    */tests/*
    manage.py

```

Testing Best Practices

- **Test Organization**
  - One test class per model/view
  - Clear test method names
  - Focused test cases
- **Test Data**
  - Use factories (e.g., Factory Boy)
  - Avoid hard-coded data
  - Clean up test data
- **Performance**
  - Use TestCase.setUpTestData()
  - Mock external services
  - Optimize database queries

Q&A

Authentication and Security

We will talk about this in the Authentication session.

Homework

1. **Implement a Django Template based view**
2. **Implement a Django RESTful endpoint**
3. **Django Admin setup for at least a model**
4. **Setup swagger or redoc**
5. **Create a Django Management Command**
6. **Write a test**

Q&A

Questions?

Feel free to ask anything