**Module 1: Introduction to APIs and REST - Detailed Guide**

**1.1 Understanding APIs**

**What are APIs?**

**API (Application Programming Interface)** is a set of rules, protocols, and tools that allows different software applications to communicate with each other. Think of an API as a waiter in a restaurant:

* **You (Client)** - Order food from the menu
* **Waiter (API)** - Takes your order to the kitchen and brings back your food
* **Kitchen (Server)** - Prepares the food based on the order

**Real-World API Examples**

1. **Weather App**: Uses weather service APIs to get current conditions
2. **Social Media**: Twitter API allows apps to post tweets, get user data
3. **Payment Processing**: PayPal API handles payment transactions
4. **Maps**: Google Maps API provides location and routing services

**Why Are APIs Important?**

**1. Separation of Concerns**

Frontend (React/Vue/Angular) ←→ API ←→ Backend (Django/Database)

* Frontend focuses on user interface
* Backend focuses on business logic and data
* API acts as a bridge between them

**2. Reusability**

* One API can serve multiple clients:
  + Web application
  + Mobile app (iOS/Android)
  + Desktop application
  + Third-party integrations

**3. Scalability**

* Different teams can work on different parts
* Each part can be scaled independently
* Technology flexibility (different languages/frameworks)

**4. Integration**

* Connect with external services
* Share data with partners
* Create ecosystems of applications

**Types of APIs**

**1. REST APIs (Most Common)**

* Uses HTTP methods (GET, POST, PUT, DELETE)
* Stateless communication
* Resource-based URLs
* JSON data format

**Example REST API calls:**

GET /api/users/ # Get all users

GET /api/users/123/ # Get user with ID 123

POST /api/users/ # Create a new user

PUT /api/users/123/ # Update user 123

DELETE /api/users/123/ # Delete user 123

**2. GraphQL APIs**

* Query language for APIs
* Single endpoint
* Client specifies exactly what data it needs
* Reduces over-fetching and under-fetching

**Example GraphQL query:**

query {

user(id: 123) {

name

email

posts {

title

createdAt

}

}

}

**3. SOAP APIs (Legacy)**

* XML-based protocol
* More complex but highly structured
* Built-in error handling
* Used in enterprise systems

**4. RPC APIs**

* Remote Procedure Call
* Function-based approach
* Examples: gRPC, JSON-RPC

**JSON and Data Serialization**

**What is JSON?**

**JSON (JavaScript Object Notation)** is a lightweight, text-based data interchange format.

**JSON Structure:**

{

"user": {

"id": 123,

"name": "John Doe",

"email": "john@example.com",

"is\_active": true,

"age": 30,

"hobbies": ["reading", "coding", "gaming"],

"address": {

"street": "123 Main St",

"city": "New York",

"zipcode": "10001"

}

}

}

**JSON Data Types:**

* **String**: "Hello World"
* **Number**: 123, 45.67
* **Boolean**: true, false
* **Array**: [1, 2, 3]
* **Object**: {"key": "value"}
* **null**: null

**Serialization vs Deserialization:**

* **Serialization**: Converting Python objects to JSON
* **Deserialization**: Converting JSON to Python objects

import json

# Python dictionary

user\_data = {

"name": "John Doe",

"age": 30,

"is\_active": True

}

# Serialization (Python → JSON)

json\_string = json.dumps(user\_data)

print(json\_string) # {"name": "John Doe", "age": 30, "is\_active": true}

# Deserialization (JSON → Python)

parsed\_data = json.loads(json\_string)

print(parsed\_data["name"]) # John Doe

**1.2 REST Architecture**

**What is REST?**

**REST (Representational State Transfer)** is an architectural style for designing networked applications. It was introduced by Roy Fielding in 2000.

**REST Principles and Constraints**

**1. Client-Server Architecture**

* Clear separation between client and server
* Client handles user interface
* Server handles data storage and business logic
* They communicate through a uniform interface

**2. Stateless**

* Each request must contain all information needed to process it
* Server doesn't store client context between requests
* Every request is independent

**Stateless Example:**

❌ Stateful (Bad):

Request 1: "Login as John"

Request 2: "Get my profile" # Server remembers John from previous request

✅ Stateless (Good):

Request 1: "Login as John" → Returns auth token

Request 2: "Get profile for token ABC123" # All info in request

**3. Cacheable**

* Responses should be cacheable when appropriate
* Improves performance and scalability
* Reduces server load

**4. Uniform Interface**

* Consistent way to interact with resources
* Uses standard HTTP methods
* Resource identification through URLs

**5. Layered System**

* Architecture can have multiple layers (proxies, gateways, load balancers)
* Client doesn't need to know about intermediate layers

**6. Code on Demand (Optional)**

* Server can send executable code to client
* Rarely used in practice

**HTTP Methods in REST**

**GET - Retrieve Data**

* **Purpose**: Get resource(s)
* **Safe**: Yes (doesn't modify data)
* **Idempotent**: Yes (same result every time)

GET /api/users/

GET /api/users/123/

GET /api/users/123/posts/

**POST - Create New Resource**

* **Purpose**: Create new resource
* **Safe**: No
* **Idempotent**: No (creates new resource each time)

POST /api/users/

Content-Type: application/json

{

"name": "Jane Doe",

"email": "jane@example.com"

}

**PUT - Update/Replace Resource**

* **Purpose**: Update entire resource
* **Safe**: No
* **Idempotent**: Yes

PUT /api/users/123/

Content-Type: application/json

{

"id": 123,

"name": "John Smith",

"email": "johnsmith@example.com"

}

**PATCH - Partial Update**

* **Purpose**: Update part of resource
* **Safe**: No
* **Idempotent**: Yes

PATCH /api/users/123/

Content-Type: application/json

{

"email": "newemail@example.com"

}

**DELETE - Remove Resource**

* **Purpose**: Delete resource
* **Safe**: No
* **Idempotent**: Yes

DELETE /api/users/123/

**HTTP Status Codes**

**1xx - Informational**

* 100 Continue - Continue with request

**2xx - Success**

* 200 OK - Request successful
* 201 Created - Resource created successfully
* 202 Accepted - Request accepted for processing
* 204 No Content - Success, but no content to return

**3xx - Redirection**

* 301 Moved Permanently - Resource moved to new URL
* 302 Found - Temporary redirect
* 304 Not Modified - Resource not changed (caching)

**4xx - Client Errors**

* 400 Bad Request - Invalid request format
* 401 Unauthorized - Authentication required
* 403 Forbidden - Access denied
* 404 Not Found - Resource doesn't exist
* 405 Method Not Allowed - HTTP method not supported
* 422 Unprocessable Entity - Validation errors

**5xx - Server Errors**

* 500 Internal Server Error - Generic server error
* 502 Bad Gateway - Invalid response from upstream server
* 503 Service Unavailable - Server temporarily unavailable

**Resource-Based URLs**

**Good RESTful URL Design:**

✅ Good Examples:

GET /api/users/ # Get all users

GET /api/users/123/ # Get user 123

POST /api/users/ # Create new user

GET /api/users/123/posts/ # Get posts by user 123

POST /api/users/123/posts/ # Create post for user 123

GET /api/posts/456/comments/ # Get comments for post 456

**Poor URL Design:**

❌ Bad Examples:

GET /api/getAllUsers/ # Verb in URL

POST /api/createUser/ # Verb in URL

GET /api/user?action=delete&id=123 # Action in query param

GET /api/posts/456/getComments/ # Verb in URL

**URL Design Best Practices:**

1. **Use Nouns, Not Verbs**
   * ✅ /api/users/
   * ❌ /api/getUsers/
2. **Use Plural Nouns**
   * ✅ /api/users/123/
   * ❌ /api/user/123/
3. **Hierarchical Structure**
   * ✅ /api/users/123/posts/456/comments/
   * ❌ /api/comments/?user=123&post=456
4. **Consistent Naming**
   * Use lowercase
   * Use hyphens for multiple words: /api/blog-posts/

**Stateless Communication**

**What Does Stateless Mean?**

Each HTTP request from client to server must contain all information necessary to understand the request. The server cannot take advantage of any stored context on the server.

**Example of Stateless Design:**

# ❌ Stateful approach (violates REST)

# Server remembers user after login

def login(request):

user = authenticate(request.data['username'], request.data['password'])

request.session['current\_user'] = user # Server stores state

return Response({'message': 'Logged in'})

def get\_profile(request):

user = request.session['current\_user'] # Relies on stored state

return Response({'profile': user.profile})

# ✅ Stateless approach (RESTful)

def login(request):

user = authenticate(request.data['username'], request.data['password'])

token = generate\_token(user) # Generate token

return Response({'token': token})

def get\_profile(request):

token = request.headers.get('Authorization') # Client sends token

user = validate\_token(token) # Validate token each time

return Response({'profile': user.profile})

**Benefits of Stateless Design:**

* **Scalability**: Requests can be handled by any server
* **Reliability**: No shared state to corrupt or lose
* **Simplicity**: Each request is independent
* **Caching**: Easier to cache responses

**1.3 Introduction to Django REST Framework**

**What is Django REST Framework?**

**Django REST Framework (DRF)** is a powerful and flexible toolkit for building Web APIs in Django. It's built on top of Django and provides:

* Serialization for both ORM and non-ORM data sources
* Authentication policies including OAuth1a and OAuth2
* Permissions and throttling policies
* A browsable API which is a huge usability win for your developers
* Extensive documentation and great community support

**Why Use DRF?**

**1. Built on Django**

* Leverages Django's ORM, authentication, and admin interface
* Familiar if you already know Django
* Uses Django's URL routing and middleware

**2. Batteries Included**

* Authentication and permissions
* Serialization and validation
* ViewSets and routers
* Pagination, filtering, and searching
* API documentation

**3. Flexible and Customizable**

* Override any component
* Custom serializers, views, and permissions
* Pluggable authentication backends

**4. Production Ready**

* Used by major companies (Mozilla, Red Hat, Heroku)
* Well-tested and maintained
* Great performance characteristics

**DRF vs Vanilla Django Views**

**Vanilla Django View:**

from django.http import JsonResponse

from django.views.decorators.csrf import csrf\_exempt

from django.views.decorators.http import require\_http\_methods

import json

@csrf\_exempt

@require\_http\_methods(["GET", "POST"])

def user\_list(request):

if request.method == 'GET':

users = User.objects.all()

data = []

for user in users:

data.append({

'id': user.id,

'username': user.username,

'email': user.email,

})

return JsonResponse({'users': data})

elif request.method == 'POST':

try:

data = json.loads(request.body)

user = User.objects.create(

username=data['username'],

email=data['email']

)

return JsonResponse({

'id': user.id,

'username': user.username,

'email': user.email

}, status=201)

except (KeyError, json.JSONDecodeError):

return JsonResponse({'error': 'Invalid data'}, status=400)

**DRF Equivalent:**

from rest\_framework import generics

from rest\_framework.response import Response

from .models import User

from .serializers import UserSerializer

class UserListView(generics.ListCreateAPIView):

queryset = User.objects.all()

serializer\_class = UserSerializer

**Benefits of DRF Approach:**

* **Less Code**: Much more concise
* **Automatic Serialization**: Handles JSON conversion
* **Validation**: Built-in data validation
* **Error Handling**: Proper HTTP status codes
* **Content Negotiation**: Supports multiple formats (JSON, XML, etc.)
* **Authentication**: Easy to add authentication
* **Permissions**: Built-in permission system

**Installation and Setup**

**1. Install Django REST Framework**

pip install django

pip install djangorestframework

**2. Add to Django Settings**

# 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 this

'your\_app',

]

# DRF Settings

REST\_FRAMEWORK = {

'DEFAULT\_PERMISSION\_CLASSES': [

'rest\_framework.permissions.IsAuthenticated',

],

'DEFAULT\_AUTHENTICATION\_CLASSES': [

'rest\_framework.authentication.SessionAuthentication',

'rest\_framework.authentication.TokenAuthentication',

],

'DEFAULT\_RENDERER\_CLASSES': [

'rest\_framework.renderers.JSONRenderer',

'rest\_framework.renderers.BrowsableAPIRenderer',

],

'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.PageNumberPagination',

'PAGE\_SIZE': 20

}

**3. Add URLs**

# urls.py

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path('admin/', admin.site.urls),

path('api/', include('your\_app.urls')),

path('api-auth/', include('rest\_framework.urls')), # DRF login/logout

]

**Project Structure Best Practices**

**Recommended Project Structure:**

myproject/

├── manage.py

├── myproject/

│ ├── \_\_init\_\_.py

│ ├── settings/

│ │ ├── \_\_init\_\_.py

│ │ ├── base.py

│ │ ├── development.py

│ │ ├── production.py

│ │ └── testing.py

│ ├── urls.py

│ └── wsgi.py

├── apps/

│ ├── \_\_init\_\_.py

│ ├── users/

│ │ ├── \_\_init\_\_.py

│ │ ├── models.py

│ │ ├── serializers.py

│ │ ├── views.py

│ │ ├── urls.py

│ │ ├── permissions.py

│ │ ├── tests.py

│ │ └── admin.py

│ └── blog/

│ ├── \_\_init\_\_.py

│ ├── models.py

│ ├── serializers.py

│ ├── views.py

│ ├── urls.py

│ └── tests.py

├── requirements/

│ ├── base.txt

│ ├── development.txt

│ └─- production.txt

└── docs/

└── api.md

**App-Level Organization:**

# apps/users/models.py

from django.contrib.auth.models import AbstractUser

from django.db import models

class User(AbstractUser):

email = models.EmailField(unique=True)

bio = models.TextField(max\_length=500, blank=True)

birth\_date = models.DateField(null=True, blank=True)

# apps/users/serializers.py

from rest\_framework import serializers

from .models import User

class UserSerializer(serializers.ModelSerializer):

class Meta:

model = User

fields = ['id', 'username', 'email', 'bio', 'birth\_date']

extra\_kwargs = {'password': {'write\_only': True}}

# apps/users/views.py

from rest\_framework import generics

from .models import User

from .serializers import UserSerializer

class UserListView(generics.ListCreateAPIView):

queryset = User.objects.all()

serializer\_class = UserSerializer

# apps/users/urls.py

from django.urls import path

from . import views

app\_name = 'users'

urlpatterns = [

path('users/', views.UserListView.as\_view(), name='user-list'),

]

**Hands-on Project: Setting Up Your First DRF Project**

**Step 1: Create Django Project**

# Create virtual environment

python -m venv drf\_env

source drf\_env/bin/activate # On Windows: drf\_env\Scripts\activate

# Install packages

pip install django djangorestframework

# Create project

django-admin startproject blog\_api

cd blog\_api

# Create app

python manage.py startapp blog

**Step 2: Configure Settings**

# blog\_api/settings.py

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'rest\_framework',

'blog',

]

REST\_FRAMEWORK = {

'DEFAULT\_PERMISSION\_CLASSES': [

'rest\_framework.permissions.AllowAny', # For development

],

'DEFAULT\_RENDERER\_CLASSES': [

'rest\_framework.renderers.JSONRenderer',

'rest\_framework.renderers.BrowsableAPIRenderer',

],

}

**Step 3: Create Models**

# blog/models.py

from django.db import models

from django.contrib.auth.models import User

class Post(models.Model):

title = models.CharField(max\_length=200)

content = models.TextField()

author = models.ForeignKey(User, on\_delete=models.CASCADE)

created\_at = models.DateTimeField(auto\_now\_add=True)

updated\_at = models.DateTimeField(auto\_now=True)

def \_\_str\_\_(self):

return self.title

**Step 4: Create Serializer**

# blog/serializers.py

from rest\_framework import serializers

from .models import Post

class PostSerializer(serializers.ModelSerializer):

author\_username = serializers.CharField(source='author.username', read\_only=True)

class Meta:

model = Post

fields = ['id', 'title', 'content', 'author', 'author\_username', 'created\_at', 'updated\_at']

extra\_kwargs = {'author': {'write\_only': True}}

**Step 5: Create Views**

# blog/views.py

from rest\_framework import generics

from .models import Post

from .serializers import PostSerializer

class PostListCreateView(generics.ListCreateAPIView):

queryset = Post.objects.all()

serializer\_class = PostSerializer

def perform\_create(self, serializer):

serializer.save(author=self.request.user)

class PostDetailView(generics.RetrieveUpdateDestroyAPIView):

queryset = Post.objects.all()

serializer\_class = PostSerializer

**Step 6: Configure URLs**

# blog/urls.py

from django.urls import path

from . import views

urlpatterns = [

path('posts/', views.PostListCreateView.as\_view(), name='post-list'),

path('posts/<int:pk>/', views.PostDetailView.as\_view(), name='post-detail'),

]

# blog\_api/urls.py

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path('admin/', admin.site.urls),

path('api/', include('blog.urls')),

path('api-auth/', include('rest\_framework.urls')),

]

**Step 7: Run Migrations and Test**

python manage.py makemigrations

python manage.py migrate

python manage.py createsuperuser

python manage.py runserver

**Step 8: Test Your API**

Visit http://127.0.0.1:8000/api/posts/ in your browser to see the DRF browsable API!

You can also test with curl:

# Get all posts

curl -X GET http://127.0.0.1:8000/api/posts/

# Create a new post (you'll need to handle authentication)

curl -X POST http://127.0.0.1:8000/api/posts/ \

-H "Content-Type: application/json" \

-d '{"title": "My First Post", "content": "Hello, DRF!"}'

**Summary**

In Module 1, you've learned:

1. **APIs Fundamentals**: What they are, why they're important, and different types
2. **REST Architecture**: The principles that guide RESTful API design
3. **HTTP Methods and Status Codes**: How to use them correctly in REST APIs
4. **JSON**: The standard format for API data exchange
5. **Django REST Framework**: Why it's powerful and how it simplifies API development
6. **Project Setup**: How to structure and configure a DRF project

**Key Takeaways:**

* APIs enable communication between different software systems
* REST is an architectural style with specific constraints and principles
* DRF provides powerful tools to build APIs quickly and correctly
* Proper project structure and configuration are important for maintainability

**Next Steps:**

In Module 2, we'll dive deep into **Serializers** - the heart of DRF that handles data conversion and validation. You'll learn how to create serializers, validate data, and handle complex relationships.

Are you ready to move on to Module 2, or do you have any questions about the concepts covered in Module 1?