FREE OPEN-SOURCE DEVELOPER GUIDE

GitBook • Leanpub • Packt Free Learning

DJANGO NINJA MADE SIMPLE

A Beginner-Friendly Guide to Building Modern Python APIs

23 chapters | **3 real projects** | **free forever**

FEATURING THREE PRACTICAL PROJECTS • STEP-BY-STEP DEPLOYMENT GUIDE

SYLVESTER BENJAMIN

Software Engineer • Cybersecurity Consultant

General Manager, Cyber Oracle

Django Ninja Made Simple

A Beginner-Friendly Guide to Building Modern APIs with Python Django Ninja 2nd Edition

Featuring Three Practical Projects §
Step-by-Step Deployment Guide

Preface

Hey there! You've just picked up Django Ninja Made Simple — the fastest, most practical way to get started with Django Ninja. If you're new to Django or just starting out as a developer, this book has your back. You don't need any prior experience with Django. I'll walk you through everything, step by step, from the very basics all the way to building a full, professional API system.

Along the way, you'll work with real examples, hands-on exercises, and some genuinely fun projects. By the time you reach the end, you'll have built a complete Banking API Capstone and a Hospital Management System. Let's dive in and start building!

What's New in This Edition

This edition has been enriched based on reader feedback. Every chapter still carries the same friendly voice and step-by-step approach you picked up this book for — only now it's sharper, cleaner, and more production-ready:

- **Visual diagrams — real flowcharts for the request lifecycle, authentication flow, and JWT process**
- Pinned requirements.txt — so you can follow along without mysterious version conflicts
- Better code blocks — monospace, clearly labelled, easy to read and copy
- New Chapter 23: Unit Testing — the one skill that separates good developers from great ones
- Each chapter now starts on its own fresh page

Let's dive in and start building!
—Sylvester Benjamin

Table of Contents

C H A P T E R 1

Introduction to Django Ninja

Building APIs the Easy Way

What Is Django Ninja?

Django Ninja is a tool that helps you build APIs (Application Programming Interfaces) using Python and Django — but faster, cleaner, and with less code.

It's called "Ninja" because it's:

- Fast: (built on top of FastAPI ideas)
- Simple: (you write fewer lines of code)
- Smart: (it checks your data automatically)

So instead of writing a lot of code to make a simple API, Django Ninja does most of the heavy lifting for you.

Why Should You Learn Django Ninja?

Because it makes API development easy — especially if you are a beginner in backend development or Django.

Here's what makes it special:

Feature	Django Ninja	Django REST Framework (DRF)
Speed	Very fast (built on ASGI)	Slower (WSGI)
Code size	Less code	More boilerplate
Validation	Uses Pydantic automatically	Needs manual setup
Docs	Auto-generates Swagger & ReDoc	Optional setup
Learning curve	Easy for beginners	Steeper learning

What Django Ninja Does for You

When you create an API, Django Ninja:

- Takes your incoming data (from a web or mobile app)
- Checks it automatically (Is it valid? Is it complete?)
- Passes it to your function to process
- Returns a clean JSON response
- And also creates interactive API documentation for you to test right
- All that --- from just a few lines of code!

Example: Your First Django Ninja API

Here's how easy it is to make your first API with Django Ninja:

```
/api.py

from ninja import NinjaAPI

api = NinjaAPI()

@api.get("/hello")
def say_hello(request):
    return {"message": "Hello, Django Ninja!"}
```

Let's Understand What's Happening:

NinjaAPI() → This is your main API app (like a container for your endpoints).

@api.get("/hello/") → This tells Django Ninja to make an API endpoint at /hello.

say_hello() → The function that runs when someone visits /hello.

It returns a dictionary, and Django Ninja automatically converts it to JSON.

Now, run your Django project and open:

http://127.0.0.1:8000/api/docs

You'll see a beautiful Swagger documentation page where you can test your /hello endpoint — no setup needed.

The "API Docs" Magic

The /api/docs page is a live testing playground that Django Ninja builds automatically for you.

It shows:

- Every endpoint in your project
- The required inputs
- Expected outputs
- And a "Try it out" button to test directly from your browser
- This saves hours of manual testing.

Beginner Tip

Django Ninja is not a replacement for Django, it works with Django. You still use Django models, ORM, and settings, but now your APIs are easier to manage.

Component	Purpose
NinjaAPI	The main API app
Routers	Group related endpoints together
Schemas (Pydantic Models)	Define input/output data

Endpoints	Functions that respond to API calls
Docs	Built-in testing & documentation (Swagger / ReDoc)

Key Components (You'll Learn These Later
Real-World Uses of Django Ninja

You can use Django Ninja to build:

- Backend for mobile apps (e.g., Flutter, React Native)
- APIs for web dashboards (e.g., Vue, React)
- Internal data services in organizations
- AI model endpoints (for ML predictions)
- Banking or E-commerce systems
- It's flexible and production-ready.

Quick Analogy

Think of Django Ninja like a friendly translator between your website/app and your backend logic.

User → (Request) → Django Ninja → (Response) → User

You just tell Django Ninja what to send and where, and it takes care of the rest.

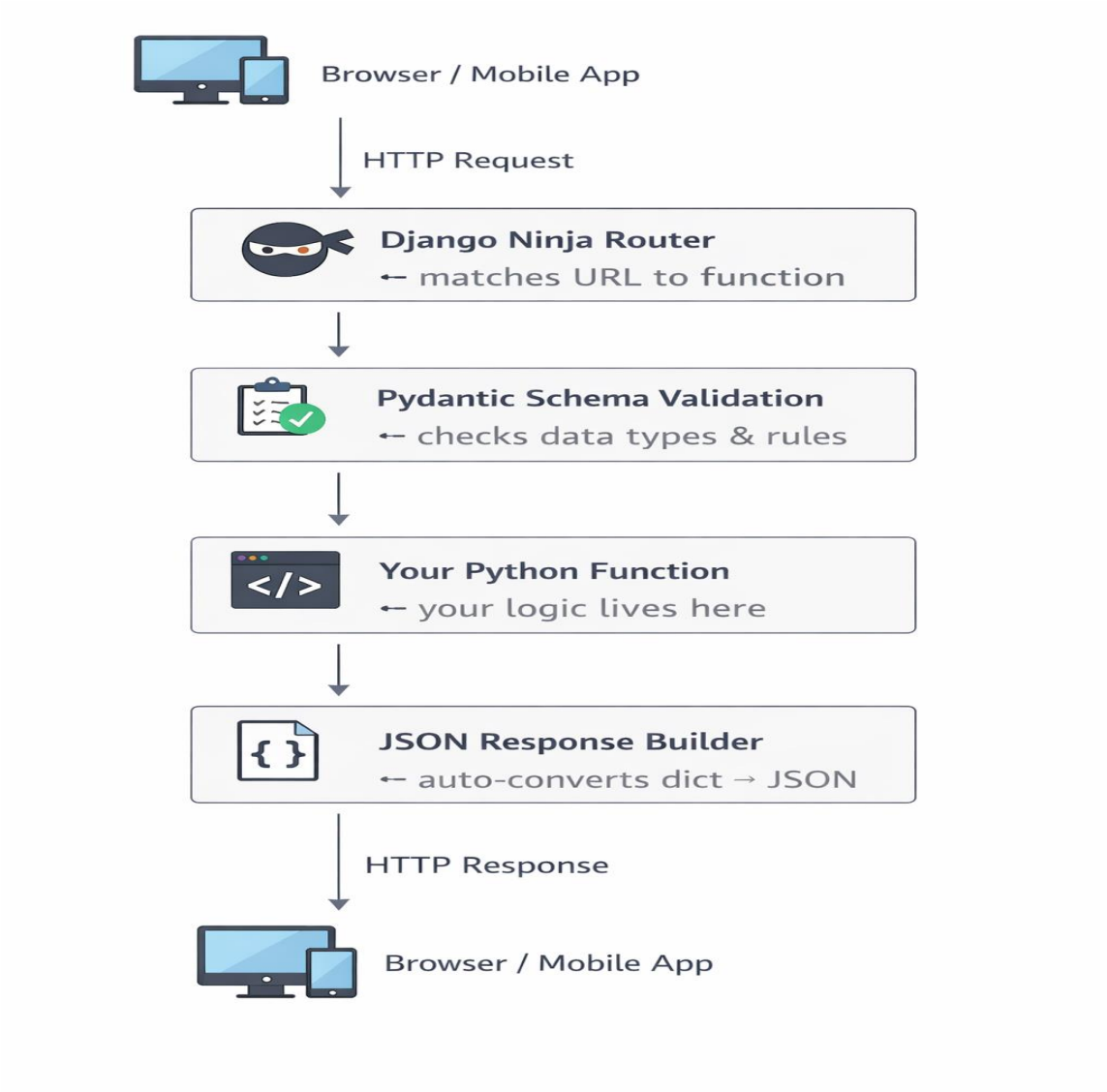
Common Terms You'll Hear Often

Term	Meaning
API	A way for software to communicate
Endpoint	A specific path (like /hello) in your API
Request	Data coming into your API

Response	Data going out from your API
JSON	The data format used to send and receive info
Swagger	A web tool that shows your API docs automatically

Request Life cycle Diagram

Here is how every request flows through a Django Ninja API:



Exercises

1. What is Django Ninja in your own words?
2. What's the difference between Django Ninja and Django REST
3. Create a simple API endpoint /greet that returns your name.
4. Open your /api/docs page and test it.

C H A P T E R 2

Setting Up Your Development Environment

Preparing Your System to Build Django Ninja APIs

The Goal of This Chapter

Before you start building APIs, you need a proper setup — your tools, editor, and Python environment.

In this chapter, you'll learn how to:

- Install Python and Django Ninja
- Create a virtual environment
- Set up your first Django project
- Verify that everything works correctly

Step 1: Check If Python Is Installed

Open your terminal (Command Prompt, PowerShell, or VS Code Terminal) and type:

- python \--version
- You should see something like:
- Python 3.10.11

If you don't have Python installed:

1. Go to <https://python.org/downloads>
2. Download the latest Python version for your OS.
3. During installation, check "Add Python to PATH" .

Step 2: Create a Project Folder

Choose a folder where you'll keep all your code projects.
For example:

- mkdir django_ninja_project
- cd django_ninja_project
- This will be your working directory.

Step 3: Create a Virtual Environment

A virtual environment keeps your project's Python packages separate

Create one with:

Then activate it:

On Windows:

- venv\\Scripts\\activate
- On Mac/Linux:
- source venv/bin/activate

Once activated, you'll see (venv) appear at the start of your terminal line — that means your virtual environment is active.

Beginner Tip:
Always activate your virtual environment before running or installing anything related to Django.

Step 4: Install Django and Django Ninja

Now install both Django and Django Ninja using pip:

```
Terminal

pip install django django-ninja
```

Wait a few seconds. Once done, you can verify the installations:

```
Terminal

python -m django --version
```

You should see something like:
5.1.2

and Django Ninja should now be available as part of your installed packages.

Recommended: Use a requirements.txt with Pinned Versions

One frustration beginners run into is following a tutorial and getting different errors because package versions changed. The fix is simple: pin your versions. Create a file called requirements.txt in your project folder and paste this:

```
Requirements.txt

 django ==1.3.0                 #the Django web framework
 django-ninja==1.3.0            #the Django Ninja API library

#Database
Psycopg2-binary==2.9.9          # PostgreSQL driver (for production)

# Authentication
 PyJWT==2.8.0                   #JSON Web Token library

#CORS
Django-cors-headers==4.4.0      #lets your frontend call the API

#Testing
Pytest==8.1.1                   #test runner

# Production
gunicorn==22.0.0                #production WSGI/ASGI Server

>Terminal – install everything in one go
pip install -r requirements.txt
```

💡 Why pin versions?

Without pinned versions, running pip install django-ninja today might install a different version than six months from now — and the API may have changed. Pinning means your code works the same way every time, on every machine.

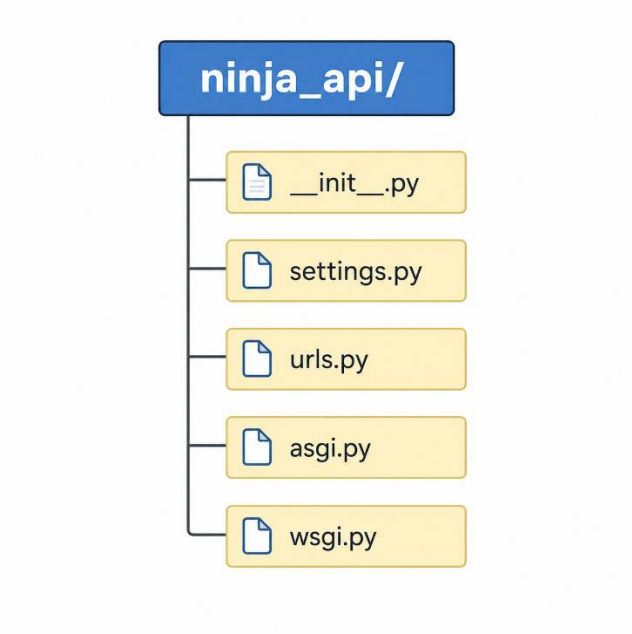
Step 5: Create a Django Project

Now, let's create your main Django project.

Run this:

```
Terminal
django-admin startproject ninja_api
```

That will create several files and folders:



These are the core files of any Django project.

You'll mainly work with settings.py, urls.py, and manage.py.

Step 6: Run the Development Server

Now test that your Django project runs correctly:

```
Terminal
python manage.py runserver
```

If everything is okay, you'll see something like: Starting development server at http://127.0.0.1:8000/
Now open that link in your web browser.

You should see the Django welcome page!

Step 7: Connect Django Ninja

Now, let's connect Django Ninja to your Django project.

Create a new file named api.py in the same folder as manage.py.
Inside it, paste this:

```
# api.py
from ninja import NinjaAPI

api = NinjaAPI()

@api.get("/hello")
def hello(request):
    return {"message": "Hello from Django Ninja!"}
```

Then, open your ninja_api/urls.py file and add this:

```
from django.contrib import admin
from django.urls import path
from api import api

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', api.urls),
]
```

Now, run your server again:

```
python manage.py runserver
```

Go to:

http://127.0.0.1:8000/api/docs

You'll see the **Django Ninja Swagger documentation** — ready to test your first endpoint

Beginner Tips

- Always activate your virtual environment before running commands.
- Keep your api.py clean and organized --- it's where your endpoints
- Visit /api/docs to test APIs without writing frontend code.
- If the server fails to start, check if api is imported correctly in

Troubleshooting

Issue	Fix
'django' is not recognized	Reinstall Python or check PATH
'ModuleNotFoundError: No module named ninja'	Run pip install django-ninja again
Server not loading	Make sure you are in the correct folder (manage.py directory)

Exercise
Try these steps:

1. Create your own API endpoint /welcome that returns your name.
2. Test it on the Swagger docs page.
3. Create a new endpoint /add that takes two numbers and returns their

Example:

```
@app.get("/add")
def add(a: int, b: int):
    return {"result": a + b}
```

CHAPTER 3

Your First Ninja API Project

Building Your First Real API Step by Step

The Goal of This Chapter

Now that you have Django and Django Ninja installed and running, it's time to build your first real API.
You'll learn:

- How API routes (endpoints) work
- How to organize your code
- How to accept user input
- How to return data as JSON

By the end of this chapter, you'll have a working mini API project.

What Is an Endpoint?

An endpoint is simply a URL path your API listens to.
For example:

Endpoint	Description
/hello	Returns a greeting
/add	Adds two numbers
/users	Returns a list of users

When someone visits your endpoint (via a web or mobile app), Django Ninja runs the Python function linked to that endpoint.

The Basic Structure of a Ninja API

You already created a simple api.py in Chapter 2.
Here's how a typical api.py looks:

```
\# api.py

From ninja import NinjaAPI

api = NinjaAPI()

@api.get(\"/hello\")
def hello(request):
    return {\"message\": \"Hello, Django Ninja!\"}
```

When you open http://127.0.0.1:8000/api/docs, Django Ninja automatically shows this /hello endpoint in the documentation — ready to test.

Creating Multiple Endpoints

You can create as many endpoints as you want, just by adding new decorators like \@api.get() or \@api.post().

Let's add a few examples:

```
# api.py
from ninja import NinjaAPI
api = NinjaAPI()

@api.get("/hello")
def hello(request):
    return {"message": "Hello, Django Ninja!"}

@api.get("/greet")
def greet_user(request, name: str):
    return {"greeting": f"Hello, {name}!"}

@api.get("/add")
def add_numbers(request, a: int, b: int):
    return {"result": a + b}
```

What's Happening Here?

- /greet?name=John → returns "Hello, John!"
- /add?a=10&b=20 → returns { \"result\": 30 }

Different Request Types (GET, POST, PUT, DELETE)

APIs use HTTP methods to define what action to perform:

Method	Action	Example
GET	Retrieve data	/users
POST	Create new data	/users
PUT	Update existing data	/users/1
DELETE	Remove data	/users/1

You'll use these to create CRUD operations (Create, Read, Update, Delete) later.

Example of a POST endpoint:

```
@api.post("/welcome")
def welcome_user(request, name: str):
    return {"message": f"Welcome, {name}!"}
```

When you test this from Swagger, it will send a POST request with name in the form data or JSON.

Handling URL Path Parameters

Sometimes you want to include a value inside the URL, not as a

query.

Example:

```
@api.get("/user/{user_id}")
def get_user(request, user_id: int):
    return {"user_id": user_id, "status": "Found"}
```

If you visit /api/user/10, Django Ninja passes 10 to the function automatically.

Beginner Tip

Query parameters (?a=1&b=2) are best for filtering or optional data.
Path parameters (/user/10) are for identifying specific items.

Organizing Your Endpoints (Routers)

As your project grows, your api.py file can get big.
That's where Routers come in — they help organize related endpoints.

Example:

```
# users.py
from ninja import Router

router = Router()

@router.get("/list")
def list_users(request):
    return {"users": ["Alice", "Bob", "Charlie"]}
Now in api.py:
```

```
from ninja import NinjaAPI
from users import router as users_router

api = NinjaAPI()
api.add_router("/users", users_router)
```

Now your docs will show:

- /users/list → returns a list of users.

You can create routers for different modules (e.g., users, products, orders) to keep code clean.

Testing It All

Start your server:

```
Terminal:
python manage.py runserver
```

Visit:

http://127.0.0.1:8000/api/docs

You'll see all endpoints listed — /hello, /greet, /add, /users/list, etc.

Click on any one, hit "Try it out", and you'll get instant results!

Common Mistakes (and Fixes)

Mistake	Fix
Forgot to add api.urls in urls.py	Add path("api/", api.urls)
Missing type hints (a: int, b: int)	Always include data types
Browser shows "Not Found"	Check the endpoint path carefully
Error: "import not found"	Make sure the file name and import path are correct

Exercises

1. Create an endpoint /multiply that takes x and y and returns their

2. Add a new router for products with a /list endpoint that returns a

3. Create a /welcome/{name} endpoint that returns \"Welcome, \<name>!\".

CHAPTER 4

Request & Response Models (Schemas)

Teaching Your API How to Speak in Organized Data

The Goal of This Chapter

You've built simple endpoints using strings and numbers.
Now it's time to teach your API to handle real-world data like user info, orders, or posts — in a structured, reliable way.

We'll do that using Schemas, which are like data blueprints.

By the end of this chapter, you'll know:

- What schemas are
- How to send and receive structured data
- How Django Ninja automatically validates and documents your data

What Is a Schema?

A Schema is a Python class that describes the shape of your data.

For example, if you're building a user registration API, your data might look like this:

```
{
  "username": "JohnDoe",
  "email": "john@example.com",
  "age": 25
}
```

To represent that data in Django Ninja, you create a Schema using Pydantic (a powerful data validation library built into Django Ninja).

Creating a Schema

Here's how to define one:

```
from ninja import Schema

class UserSchema(Schema):
    username: str
    email: str
    age: int
```

That's it!

This class tells Django Ninja what to expect — and it will automatically:

- Validate incoming data
- Convert data to JSON
- Generate API documentation fields

Using Schema for Input (Request Body)

Now, let's accept user data in an endpoint:

```
from ninja import NinjaAPI, Schema

api = NinjaAPI()

class UserSchema(Schema):
    username: str
    email: str
    age: int

@api.post("/register")
def register_user(request, data: UserSchema):
    return {"message": f"User {data.username} registered successfully!"}
```

How It Works:

- When a user sends JSON data (username, email, age),
- Django Ninja automatically converts it into a UserSchema object

(data),

- You can then access data.username, data.email, etc.

Try it in Swagger!
It automatically shows the form fields and checks the data types for you

Sending Structured Data Back (Response Model)

Now let's return a schema as the response too.

```
class UserResponse(Schema):
    id: int
    username: str
    email: str


@api.post("/create", response=UserResponse)
def create_user(request, data: UserSchema):
    return {"id": 1, "username": data.username, "email": data.email}
```

Now, Django Ninja automatically formats and documents the response structure.

Behind the Scenes

When you add response=UserResponse, Ninja:

- Validates your output
- Converts Python objects to JSON
- Documents the result in Swagger

This makes your API predictable and consistent.

Example: Full Create and Get User Flow

```
from ninja import NinjaAPI, Schema

api = NinjaAPI()

class UserIn(Schema):
    username: str
    email: str

class UserOut(Schema):
    id: int
    username: str
    email: str

users = []  # pretend database

@api.post("/users", response=UserOut)
def create_user(request, data: UserIn):
    user_id = len(users) + 1
    new_user = {"id": user_id, **data.dict()}
    users.append(new_user)
    return new_user

@api.get("/users", response=list[UserOut])
def list_users(request):
    return users
```

What's Happening:

- /users (POST) → Creates a user
- /users (GET) → Lists all created users

Try it in Swagger — you'll see both routes, full form fields, and automatic documentation.

Data Validation Magic

For example, if your schema says:

Django Ninja automatically checks input data for you.

For example, if your schema says:

```
class Product(Schema):
    name: str
    price: float
Then sending this:
{ "name": "Laptop", "price": "abc" }
will return:
{ "detail": [{"loc": ["body", "price"], "msg": "value is not a valid float"}] }
```

No manual validation needed!

Default Values and Optional Fields

You can set defaults and make some fields optional.

title: str

author: Optional [str] = "Unknown" year: int = 2024

If a user omits author, it'll automatically become "Unknown".

Quick Tips

- Use Input Schema for data coming in (like UserIn)
- Use Output Schema for data going out (like UserOut)
- Always use type hints (str, int, float, bool, etc.)
- Swagger auto-generates docs for all fields

Try It Yourself

1. Create a schema called ProductIn with fields: name (str), price (float), stock (int)
2. Create ProductOut with an extra field id (int)
3. Build:

- /products (POST) → adds a product
- /products (GET) → lists all products

C H A P T E R 5

Connecting Django Ninja with a Database (Models + ORM)

Teaching Your API to Remember Things

The Goal of This Chapter

So far, your APIs can accept and send data — but every time you restart the server, all that data disappears.
It's time to give your API a memory by connecting it to a database using Django's ORM (Object Relational Mapper).

By the end of this chapter, you'll be able to:

- Connect Django Ninja to a database
- Create and use models
- Save, retrieve, and list data using Django ORM

What is an ORM?

ORM stands for Object Relational Mapper.
It allows you to interact with your database using Python objects instead of raw SQL.

For example:

```
# Instead of SQL like this
INSERT INTO users (username, email) VALUES ('John', 'john@example.com');

# You can do this
user = User(username="John", email="john@example.com")
user.save()
```

ORMs make your code **cleaner**, **safer**, and **database-independent**.

Step 1: Setting Up Your Database

If you open your project folder, you'll see this line in settings.py:

```
DATABASES = {
```

```
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / "db.sqlite3",
    }
}
```

You can keep this as is for now.

Step 2: Creating a Model

A model defines a table in your database.
Let's create one for our users.

Create a file inside your Django app (e.g. users/models.py):

```
from django.db import models

class User(models.Model):
    username = models.CharField(max_length=50)
    email = models.EmailField(unique=True)
    age = models.IntegerField()

    def __str__(self):
        return self.username
```

Each field represents a column in the database table.

Step 3: Run Migrations

Once the model is ready, tell Django to create the actual table in the database:

```
python manage.py makemigrations
python manage.py migrate
```

Done!

You now have a users_user table in your database.

Step 4: Connecting the Model to Ninja API

Now, let's create endpoints that use our User model.

Inside api.py (or your main views file):

```
from ninja import NinjaAPI, Schema
from .models import User

api = NinjaAPI()

class UserIn(Schema):
    username: str
    email: str
    age: int

class UserOut(Schema):
    id: int
    username: str
    email: str
    age: int

@api.post("/users", response=UserOut)
def create_user(request, data: UserIn):
    user = User.objects.create(**data.dict())
    return user

@api.get("/users", response=list[UserOut])
def list_users(request):
    return list(User.objects.all())
```

What's Happening Here

- User.objects.create(\\data.dict()) creates and saves a new record.
- User.objects.all() fetches all users from the database.
- The schemas make sure the input and output are properly structured.

Step 5: Viewing Your Data

After creating a few users in Swagger, check your database.

You can open the Django shell:

```
python manage.py shell
Then run:
from users.models import User
User.objects.all()
```

You'll see your data stored — persistence achieved!

Example Output

```
POST /users
{
  "username": "Alice",
  "email": "alice@example.com",
  "age": 28
}
Response
{
  "id": 1,
  "username": "Alice",
  "email": "alice@example.com",
  "age": 28
}
GET /users
```

```
[
  {
    "id": 1,
    "username": "Alice",
    "email": "alice@example.com",
    "age": 28
  }
]
```

Updating and Deleting Data (Preview)

We'll cover this properly in the next chapter (CRUD Operations),
but here's a sneak peek:

```
@api.put("/users/{user_id}", response=UserOut)
def update_user(request, user_id: int, data: UserIn):
    user = User.objects.get(id=user_id)
    for attr, value in data.dict().items():
        setattr(user, attr, value)
    user.save()
    return user

@api.delete("/users/{user_id}")
def delete_user(request, user_id: int):
    User.objects.filter(id=user_id).delete()
    return {"message": "User deleted successfully!"}
```

Try It Yourself

4. Create a model for Product with fields: name, price, and stock.
5. Create schemas (ProductIn, ProductOut) for API I/O.
6. Build /products (POST) and /products (GET) endpoints.
7. Test in Swagger UI and check the database.

C H A P T E R 6

CRUD Operations in Django Ninja

Create, Read, Update, Delete — the Four Pillars of Every API

The Goal of This Chapter

Every useful API needs to do four basic things:

1. Create data
2. Read data
3. Update data
4. Delete data

These are called CRUD operations, and Django Ninja makes them *super easy* to build.

By the end of this chapter, you'll have a fully functional CRUD API that works with a database and validates everything automatically.

What is CRUD?

CRUD stands for:

Operation	HTTP Method	Description
Create	POST	Add new data
Read	GET	Retrieve data
Update	PUT or PATCH	Modify existing data
Delete	DELETE	Remove data

Our Example: Managing Products

We'll build a simple Product Management API that can:

- Add new products
- View all products

- Update existing ones
- Delete unwanted ones

Step 1: Define the Model

In products/models.py:

```
from django.db import models
class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.FloatField()
    stock = models.IntegerField(default=0)


    def __str__(self):
        return self.name
```

Run your migrations to create the table:

```
python manage.py makemigrations
python manage.py migrate
```

Step 2: Create Schemas

In products/api.py:

```
from ninja import Schema
class ProductIn(Schema):
    name: str
    price: float
    stock: int

class ProductOut(Schema):
    id: int
    name: str
    price: float
    stock: int
```

Step 3: Create the API Endpoints

Now let's implement all CRUD routes using Django Ninja.

```
from ninja import NinjaAPI
from .models import Product
from .api import ProductIn, ProductOut
```

```
api = NinjaAPI()

# CREATE
@api.post("/products", response=ProductOut)
def create_product(request, data: ProductIn):
    product = Product.objects.create(**data.dict())
    return product

# READ ALL
@api.get("/products", response=list[ProductOut])
def list_products(request):
    return list(Product.objects.all())

# READ SINGLE
@api.get("/products/{product_id}", response=ProductOut)
def get_product(request, product_id: int):
    return Product.objects.get(id=product_id)

# UPDATE
@api.put("/products/{product_id}", response=ProductOut)
def update_product(request, product_id: int, data: ProductIn):
    product = Product.objects.get(id=product_id)
    for attr, value in data.dict().items():
        setattr(product, attr, value)
    product.save()
    return product

# DELETE
@api.delete("/products/{product_id}")
def delete_product(request, product_id: int):
```

```
    Product.objects.filter(id=product_id).delete()
    return {"message": "Product deleted successfully!"}
```

Understanding the Flow

You can test all these actions directly in Swagger UI (automatically generated at /api/docs).

Acton	Endpoint	Description
POST	/products	Create a new Product
GET	/products	View all product
GET	/products/{id}	View a single product
PUT	/products/{id}	Update product info
DELETE	/products/{id}	Delete product

You can test all these actions directly in swagger UI (automatically generated at /api/docs)

Example Interaction

```
POST /products
{
  "name": "Laptop",
  "price": 899.99,
  "stock": 10
}
Response
{
  "id": 1,
  "name": "Laptop",
  "price": 899.99,
  "stock": 10
}
```

```
GET /products
[
 {
  "id": 1,
  "name": "Laptop",
  "price": 899.99,
  "stock": 10
 }
]
```

Common Errors (and Fixes)

Error: Product.DoesNotExist
Happens when you try to access a product that doesn't exist.

Fix:

```
from django.shortcuts import get_object_or_404

@api.get("/products/{product_id}", response=ProductOut)
def get_product(request, product_id: int):
    product = get_object_or_404(Product, id=product_id)
    return product
```

Error: Missing field in POST

Django Ninja automatically checks for missing or invalid fields — and returns a helpful validation error.

Example:

```
{
```

```
  "detail": [
    {
      "loc": ["body", "price"],
      "msg": "field required"
    }
  ]
}
```

No manual checks needed — Ninja and Pydantic do the hard work!

Optional Improvement: PATCH for Partial Updates

Sometimes you don't want to update all fields — only one or two. That's what PATCH is for.

```
from typing import Optional
class ProductPatch(Schema):
    name: Optional[str]
    price: Optional[float]
    stock: Optional[int]

@api.patch("/products/{product_id}", response=ProductOut)
def partial_update(request, product_id: int, data: ProductPatch):
    product = Product.objects.get(id=product_id)
    for attr, value in data.dict(exclude_unset=True).items():
        setattr(product, attr, value)
    product.save()
    return product
```

Now you send only the fields you want to change.

CHAPTER 7

Authentication Basics

Keeping Your API Safe — Only the Right People Get In

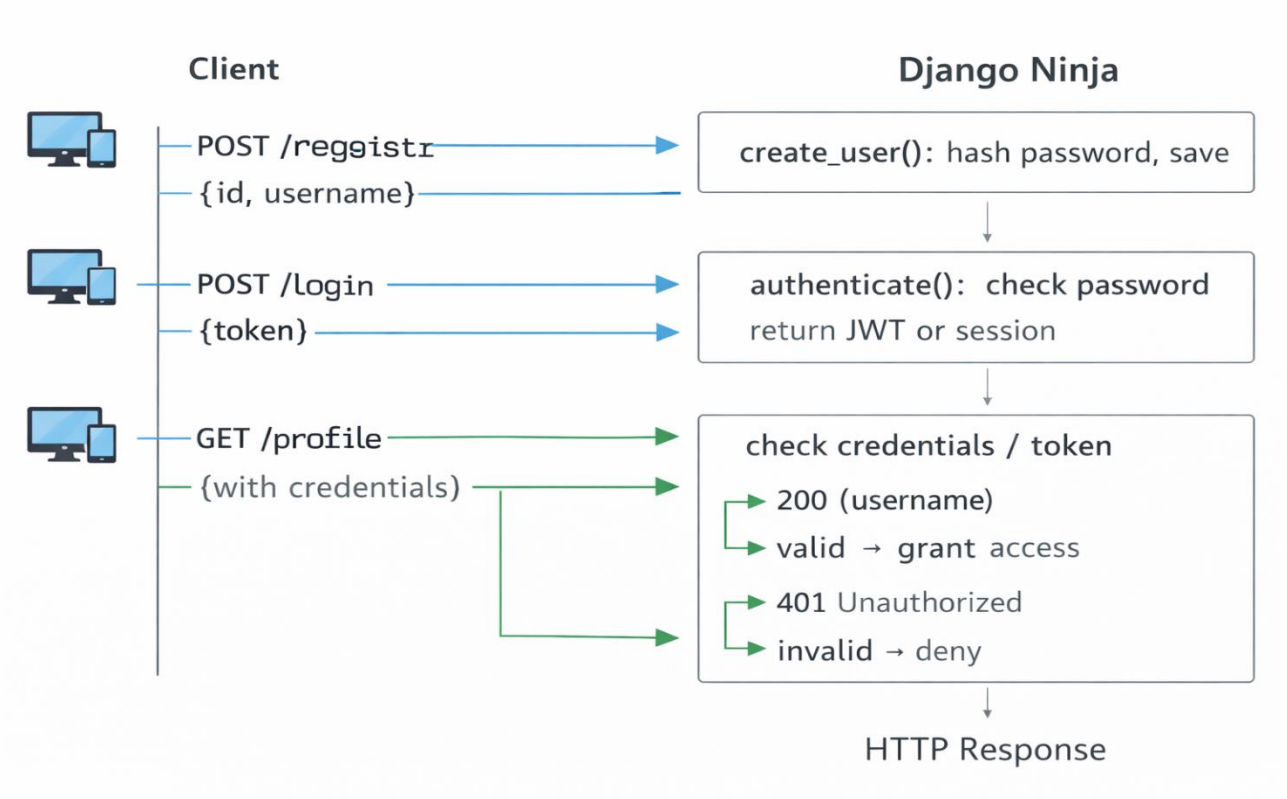


Figure 7.1 — Authentication flow in Django Ninja

The Goal of This Chapter

Up to now, your API accepts and returns data freely — which is great for testing, but terrible for security.

We need to make sure only authorized users can access certain endpoints.

This is where authentication comes in.

By the end of this chapter, you'll understand:

- What authentication is
- How to protect endpoints in Django Ninja
- How to use Django's built-in authentication system

What Is Authentication?

Authentication means confirming who someone is.

When you log in to a website or mobile app, you enter your credentials
— usually an email/username and password.
If correct, you're allowed to access protected resources.

Example:

- Anyone can view a public product list (no login).
- Only admins can add or delete products (login required).

Step 1: Using Django's Built-in Auth System

- Users
- Passwords (securely hashed)
- Sessions (for logged-in users)
- Permissions & Groups

We'll build on top of that using Django Ninja.

Step 2: Add Django's Auth App

Open your settings.py and ensure this line is present:

```
INSTALLED_APPS = [
```

```
    ...,
    'django.contrib.auth',
    'django.contrib.contenttypes',
]
```

Run migrations if you haven't already:

```
python manage.py migrate
```

This creates the user tables in your database.

Step 3: Create a User Registration API

Let's allow new users to sign up via the API.

```
from django.contrib.auth.models import User
from ninja import NinjaAPI, Schema

api = NinjaAPI()

class RegisterIn(Schema):
    username: str
    email: str
    password: str

class RegisterOut(Schema):
    id: int
    username: str
    email: str

@api.post("/register", response=RegisterOut)
def register(request, data: RegisterIn):
    user = User.objects.create_user(
        username=data.username,
        email=data.email,
        password=data.password
    )
    return use
```

This securely saves the user with a hashed password.

Step 4: Basic Authentication (Login Required)

Django Ninja supports Basic Authentication — a simple username + password check for endpoints.

Add this to api.py:

```
Add this to api.py:
from ninja.security import HttpBasicAuth

class BasicAuth(HttpBasicAuth):
    def authenticate(self, request, username, password):
        from django.contrib.auth import authenticate
        user = authenticate(username=username, password=password)
        if user:
            return user
Now use it to protect an endpoint:
auth = BasicAuth()

@api.get("/profile", auth=auth)
def get_profile(request):
    user = request.auth  # authenticated user object
    return {"message": f"Welcome {user.username}!"}
```

How It Works

1. When calling /profile, the client must include credentials (username & password).
2. Django verifies them using its built-in user system.
3. If valid → access granted.
4. If invalid → 401 Unauthorized.

Swagger UI even provides a small \"Authorize\" button so you can test authentication easily.

Step 5: Restricting Access to Admins Only

You can add simple permission logic:

```
@api.get("/admin-only", auth=auth)
def admin_area(request):
    user = request.auth
    if not user.is_staff:
```

```
    return api.create_response(request, {"error": "Access denied"}, status=403)
  return {"message": f"Welcome, admin {user.username}!"}
```

This ensures only staff members (users with is_staff=True) can access this route.

Step 6: Testing the Authentication Flow

1. Use /register to create a new user.
2. Go to /profile in Swagger.
3. Click "Authorize" → enter your username and password.
4. Test again --- you should see your welcome message.

Success! Your first protected API endpoint works.

Bonus: Using Django's Built-in Login System

You can also use Django's built-in login/logout logic if you want to integrate with web sessions:

```
from django.contrib.auth import login, logout

@api.post("/login")
def login_user(request, data: RegisterIn):
  from django.contrib.auth import authenticate
  user = authenticate(username=data.username, password=data.password)
  if user:
    login(request, user)
    return {"message": "Login successful"}
  return {"error": "Invalid credentials"}

@api.post("/logout")
```

```
def logout_user(request):
    logout(request)
    return {"message": "Logged out successfully"}
```

This works with Django session-based authentication (great for web apps).

Try It Yourself

1. Create /register and /profile endpoints.
2. Use BasicAuth to protect /profile.
3. Test logging in with valid and invalid credentials.
4. Add a new /admin-only route for staff users.

C H A P T E R 8

JWT Authentication

Modern, Token-Based Security for Your Django Ninja API

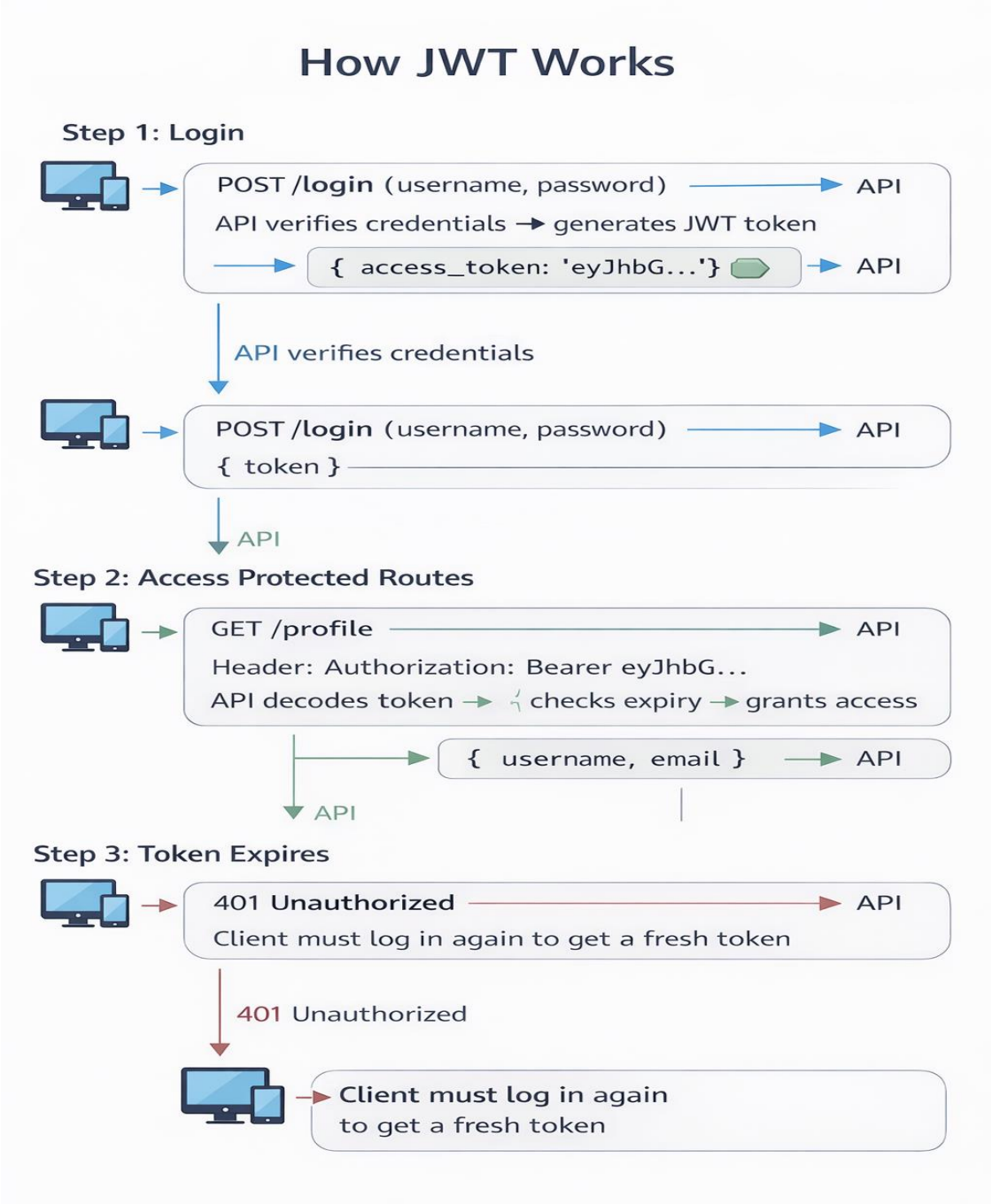


Figure 8.1 — How JWT authentication works

The Goal of This Chapter

Basic Authentication is simple, but it's not ideal for production apps — credentials are sent with every request.

Modern systems use JWT (JSON Web Tokens) — short-lived, secure tokens that clients can store and send automatically.

By the end of this chapter, you'll know:

- What JWT is and why it's used
- How to set up JWT authentication in Django Ninja
- How to protect routes using tokens

What Is JWT?

A JWT (JSON Web Token) is a secure digital token that identifies a user.

Instead of sending a username/password every time, a user logs in once and gets a token.

That token is then sent with every request.

Typical flow:

1. User logs in → API gives a JWT token.
2. User includes token in headers → API checks it → Grants access.

JWTs are stateless — they don't require Django sessions or cookies.
They're ideal for mobile and modern web apps.

Step 1: Install the JWT Library

Django Ninja doesn't include JWT by default, but it works perfectly with the popular

library **PyJWT**.

Install it:

```
Terminal
pip install PyJWT
```

Step 2: Create a JWT Utility File

In your app folder, create auth_utils.py:

```
import jwt
from datetime import datetime, timedelta
from django.conf import settings

# Secret key from settings.py
JWT_SECRET = settings.SECRET_KEY
JWT_ALGORITHM = "HS256"

def create_jwt_token(user):
    payload = {
        "user_id": user.id,
        "username": user.username,
        "exp": datetime.utcnow() + timedelta(hours=1)  # expires in 1 hour
    }
    return jwt.encode(payload, JWT_SECRET, algorithm=JWT_ALGORITHM)

def decode_jwt_token(token):
    try:
        payload = jwt.decode(token, JWT_SECRET, algorithms=[JWT_ALGORITHM])
        return payload
    except jwt.ExpiredSignatureError:
        return None
    except jwt.InvalidTokenError:
        return None
```

This file creates and validates JWTs.

Step 3: Create Login API That Returns JWT

```
from django.contrib.auth import authenticate
from ninja import NinjaAPI, Schema
from .auth_utils import create_jwt_token

api = NinjaAPI()
class LoginIn(Schema):
    username: str
    password: str
class LoginOut(Schema):
    access_token: str
    token_type: str = "Bearer"

@api.post("/login", response=LoginOut)
def login(request, data: LoginIn):
    user = authenticate(username=data.username, password=data.password)
    if not user:
        return api.create_response(request, {"error": "Invalid credentials"}, status=401)

    token = create_jwt_token(user)
    return {"access_token": token}
```

When a user logs in, they receive a token like:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

Step 4: Create JWT Authentication Class

In api.py:

```
from ninja.security import HttpBearer
from .auth_utils import decode_jwt_token
from django.contrib.auth.models import User

class JWTAuth(HttpBearer):
    def authenticate(self, request, token):
```

```
    payload = decode_jwt_token(token)
    if payload:
        try:
            user = User.objects.get(id=payload["user_id"])
            return user
        except User.DoesNotExist:
            return None
```

Step 5: Protect an Endpoint

```
auth = JWTAuth()


@api.get("/profile", auth=auth)
def get_profile(request):
    user = request.auth
    return {"username": user.username, "email": user.email}
```

Now, to access /profile, the user must send a valid JWT token.
Swagger UI automatically provides a **"Authorize"** button.

Enter your token as:
Bearer <your_token_here>

Example Flow

Login

POST /login

```
{
  "username": "alice",
  "password": "password123"
}
```

Response

```
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "token_type": "Bearer"
}
```

Access Protected Route

GET /profile

Header:

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

Response

```
{
  "username": "alice",
  "email": "alice@example.com"
}
```

Step 6: Handling Expired Tokens

If the token expires (after 1 hour in our example), the API will respond:

```
{"detail": "Unauthorized"}
```

You can refresh tokens by simply re-logging in or building a /refresh-token route.

Optional: Token Expiry and Refresh

If you want users to get a new token before the old one expires, you can add:

```
def refresh_jwt_token(old_token):
    payload = decode_jwt_token(old_token)
    if payload:
        user_id = payload["user_id"]
        username = payload["username"]
        new_payload = {
            "user_id": user_id,
            "username": username,
            "exp": datetime.utcnow() + timedelta(hours=1)
        }
        return jwt.encode(new_payload, JWT_SECRET, algorithm=JWT_ALGORITHM)
    return None
```

Try It Yourself

1. Register a user.
2. Log in to get a JWT token.
3. Copy the token and paste it in Swagger's Authorize section.
4. Try accessing /profile (it will work only if token is valid).

CHAPTER 9

Role-Based Permissions

Controlling What Users Can Access in Your API

The Goal of This Chapter

Now that we've learned how to authenticate users with JWT, the next step is authorization — deciding what a user can do after they login.

By the end of this chapter, you'll learn:

- The difference between authentication and authorization
- How to assign user roles (e.g., admin, editor, viewer)
- How to restrict routes based on user permissions
- How to make your API smarter and more secure

Authentication vs. Authorization

Concept	Description	Example
Authentication	Confirms *who* the user is	Logging in with username/password
Authorization	Determines *what* the user can do	Admin can delete users; viewer cannot

They work together:

1. Authentication → Validates the user (using JWT).
2. Authorization → Checks their permissions (using roles or

Step 1: Add Roles to Users

There are two common ways to manage roles:

3. Use Django's built-in Groups and Permissions
4. Create a custom field in your user model (simpler for small

Let's use the custom field method for clarity.

Update models.py

```
from django.contrib.auth.models import AbstractUser
from django.db import models
class User(AbstractUser):
    ROLE_CHOICES = (
        ("admin", "Admin"),
        ("editor", "Editor"),
        ("viewer", "Viewer"),
    )
    role = models.CharField(max_length=10, choices=ROLE_CHOICES, default="viewer")

    def __str__(self):
        return f"{self.username} ({self.role})"
```

Then run:

```
python manage.py makemigrations
python manage.py migrate
```

Step 2: Create Sample Users with Roles

You can create users using Django Admin or Python shell:

Or directly in shell:

```
python manage.py createsuperuser
```

Or directly in shell:

```
from app.models import User
User.objects.create_user(username="alice", password="1234", role="admin")
```

```
User.objects.create_user(username="bob", password="1234", role="editor")
User.objects.create_user(username="eve", password="1234", role="viewer")
```

Step 3: Create Role-Based Permission Decorator

We'll create a simple helper function to check if a user has the right role.

Create a new file: permissions.py

```
from ninja.errors import HttpError

def role_required(allowed_roles):
    def decorator(func):
        def wrapper(request, *args, **kwargs):
            user = request.auth
            if user.role not in allowed_roles:
                raise HttpError(403, f"Permission denied: {user.role} cannot access this resource.")
            return func(request, *args, **kwargs)
        return wrapper
    return decorator
```

Now you can easily protect routes by specifying roles.

Step 4: Use Role-Based Decorators on Routes

In your api.py:

```
from .permissions import role_required
from .auth_utils import JWTAuth


auth = JWTAuth()

@api.get("/admin/users", auth=auth)
@role_required(["admin"])
def list_users(request):
    from django.contrib.auth.models import User
    users = User.objects.all().values("id", "username", "email", "role")
    return list(users)

@api.post("/edit/article", auth=auth)
@role_required(["admin", "editor"])
def edit_article(request):
    return {"message": f"{request.auth.username} edited the article!"}
```

```
@api.get("/read/article", auth=auth)
@role_required(["admin", "editor", "viewer"])
def read_article(request):
    return {"message": "You can view this article"}
```

Example Flow:

- alice (admin) → can list users, edit, and read.
- bob (editor) → can edit and read.
- eve (viewer) → can only read.

Step 5: Test It in Swagger

1. Log in as bob (editor).
2. Copy your JWT token.
3. Click Authorize in Swagger → Paste Bearer \<token\>.
4. Try to call /admin/users → You'll get a 403 Forbidden.
5. Try /edit/article → Works
6. Try /read/article → Works

This proves your permission system works correctly.

Optional: Use Django's Built-in Groups

If you're building a larger system, you can use:
Each group (like "Admin", "Editor", etc.) can be linked to permissions dynamically.
But for learning and small apps, our custom field method is simpler and easier to understand.

Exercise

1. Add a new role called \"moderator\".
2. Give moderators the ability to read and edit, but not delete.
3. Update the decorator to allow this new role in /edit/article.
4. Test it by creating a moderator user and checking your API routes.

CHAPTER 10

API Versioning and Documentation Mastery

Keeping Your APIs Organized, Compatible, and Well-Explained

The Goal of This Chapter

As your API grows, new versions may be needed — for bug fixes, feature upgrades, or compatibility reasons. You'll also want clear, automatic documentation for users and developers.

By the end of this chapter, you'll learn:

- How to organize and version your APIs (v1, v2, etc.)
- How to customize Django Ninja's Swagger / ReDoc docs
- How to provide rich documentation with examples and tags

Step 1: What Is API Versioning?

Imagine your mobile app calls your API. You update it and change some routes — suddenly, old apps stop working

Versioning solves this.
It lets you publish new versions while keeping old ones running.

Example:

/api/v1/users
/api/v2/users

Both can exist at the same time, and clients choose which version to use.

Step 2: Implementing Versioning in Django Ninja

In Django Ninja, versioning is super easy — just create multiple

NinjaAPI instances.

Example:

```
from ninja import NinjaAPI
api_v1 = NinjaAPI(version="1.0.0", title="Banking API - v1")
api_v2 = NinjaAPI(version="2.0.0", title="Banking API - v2")
```

You can register both in your project's urls.py:

```
from django.urls import path
from .api_v1 import api as api_v1
from .api_v2 import api as api_v2

urlpatterns = [
    path("api/v1/", api_v1.urls),
    path("api/v2/", api_v2.urls),
]
```

Now, you can update your routes independently without breaking older clients.

Step 3: Organizing Endpoints by Version

Let's imagine v1 returns basic user info, and v2 adds account balance.

api_v1.py

```
from ninja import NinjaAPI

api = NinjaAPI(title="Banking API v1")

@api.get("/users")
def list_users(request):
    return [{"id": 1, "name": "Alice"}]
```

api_v2.py

```
from ninja import NinjaAPI

api = NinjaAPI(title="Banking API v2")

@api.get("/users")
def list_users(request):
    return [{"id": 1, "name": "Alice", "balance": 2000}]
```

Now, visiting:

- /api/v1/docs → Version 1 docs
- /api/v2/docs → Version 2 docs

Both versions coexist safely.

Step 4: Auto Documentation (Swagger & ReDoc)

- Swagger UI (interactive testing)
- ReDoc UI (clean reference-style documentation)

Visit:

http://127.0.0.1:8000/api/v1/docs

or

http://127.0.0.1:8000/api/v1/redoc

You'll see all routes, models, and example requests.
No extra setup required!

Step 5: Customizing Documentation

You can enhance the docs using parameters inside route decorators.

Example:

The docs will show:

- Tag: "User Management"
- Summary: Short title
- Description: Full explanation

This makes your docs look professional and easy to navigate.

Step 6: Adding Examples and Metadata

You can show request/response examples in Swagger for better clarity.

```
class CreateUserIn(Schema):
    name: str
    email: str

class CreateUserOut(Schema):
    id: int
    name: str
    email: str

@router.post(
    "/users",
    response=CreateUserOut,
    summary="Create new user",
    description="Registers a new user and returns their details",
    examples={
        "input": {"name": "John", "email": "john@example.com"},
        "output": {"id": 1, "name": "John", "email": "john@example.com"}
    }
)
def create_user(request, data: CreateUserIn):
    return {"id": 1, **data.dict()}
```

Swagger will show "Try it out" examples — perfect for beginners testing your API.

Step 7: Customizing Global API Info

When creating your NinjaAPI instance, you can personalize the documentation details:

```
api = NinjaAPI(
    title="Cyber Oracle Banking API",
    version="2.0.0",
    description="A secure and high-performance banking backend powered by Django Ninja.",
    docs_url="/docs",
    openapi_url="/openapi.json"
)
```

Your docs will now show your title, version, and description beautifully.

Exercise

5. Create a v1 and v2 version of your "Banking API."

6. In v2, add a new field called balance to the User model output.

7. Add Swagger tags and a summary to each route.

8. Visit both /v1/docs and /v2/docs --- compare the results.

CHAPTER 11

Error Handling & Validation

Making Your API Smart, Reliable, and Foolproof

The Goal of This Chapter

Your API should never crash or confuse users when bad data or errors occur.

This chapter teaches you how to gracefully handle such issues.

By the end, you'll learn:

- How Django Ninja automatically validates input
- How to define custom error messages
- How to handle server and client errors
- How to send clean, structured responses for all situations

Step 1: Built-in Validation with Pydantic

Example:

```
from ninja import Schema, NinjaAPI
api = NinjaAPI()

class UserIn(Schema):
    name: str
    age: int

@api.post("/users")
def create_user(request, data: UserIn):
    return {"message": f"User {data.name} ({data.age}) created successfully"}
```

Now, if a client sends:

```
{"name": "Alice", "age": "twenty"}
Ninja instantly returns:
{
  "detail": [
    {
      "type": "int_parsing",
      "loc": ["body", "age"],
      "msg": "Input should be a valid integer"
    }
  ]
```

```
}
```

No manual validation needed — Ninja handles it for you!

Step 2: Adding Default Values and Constraints

You can add validation rules and defaults inside your schemas.

```
from pydantic import Field
class RegisterUser(Schema):
    username: str = Field(..., min_length=3, max_length=20)
    password: str = Field(..., min_length=6)
    email: str = Field(..., regex=r"^[\w\.-]+@[\w\.-]+\.\w+$")
```

Now, Ninja will:

- Reject usernames shorter than 3 characters
- Reject invalid emails
- Require all fields (\... means required)

Step 3: Handling Common Errors Gracefully

If something goes wrong in your route, you can raise a HttpError.

```
from ninja.errors import HttpError
@api.get("/divide")
def divide(request, a: int, b: int):
    if b == 0:
        raise HttpError(400, "Division by zero is not allowed.")
    return {"result": a / b}
```

Example:

GET /divide?a=10&b=0

Response:

```
{
```

```
  "detail": "Division by zero is not allowed."
}
```

Step 4: Custom Error Responses

You can also define your own error format using exception handlers.

```
from ninja.errors import HttpError
from django.http import JsonResponse
@api.exception_handler(HttpError)
def custom_http_error(request, exc):
    return JsonResponse({
        "success": False,
        "error": str(exc),
        "path": request.path
    }, status=exc.status_code)
Now all errors will return in a neat, uniform JSON structure:
{
  "success": false,
  "error": "Division by zero is not allowed.",
  "path": "/divide"
}
```

Step 5: Global Exception Handling

Sometimes an unexpected server error might occur.

You can catch all exceptions using Exception in a handler.

```
@api.exception_handler(Exception)
def global_exception_handler(request, exc):
    return JsonResponse({
        "success": False,
        "error": "Unexpected server error. Please try again later."
    }, status=500)
```

This ensures users never see ugly error traces.

Step 6: Validation for Query & Path Parameters

You can also validate query parameters using type hints:

```
@api.get("/search")
def search_users(request, keyword: str, limit: int = 10):
    if limit > 100:
        raise HttpError(400, "Limit cannot exceed 100.")
    return {"message": f"Searching for {keyword}, limit {limit}"}
```

- keyword is required
- limit defaults to 10
- Raises an error if limit \> 100

Step 7: Custom Validation with Pydantic Validators

If you want to apply complex rules, use \@validator.

```
from pydantic import validator

class ProductIn(Schema):
    name: str
    price: float

    @validator("price")
    def price_must_be_positive(cls, v):
        if v <= 0:
            raise ValueError("Price must be greater than zero")
        return v
```

Invalid Input:

```
{"name": "Laptop", "price": -100}
Response:
{
  "detail": [
    {
      "loc": ["body", "price"],
      "msg": "Price must be greater than zero",
      "type": "value_error"
```

```
  }
 ]
}
```

Exercise

9. Add validation for user registration (name, email, password).
10. Add a /calculate route that validates positive numbers only.
11. Add a global handler for all ValueError exceptions.
12. Test it using Swagger's "Try It Out" feature.

CHAPTER 12

Pagination & Filtering

Keeping API Responses Fast, Clean, and Easy to Navigate

The Goal of This Chapter

As your app grows, you'll likely have **hundreds or thousands of records** — users, products, transactions, etc.

Returning them all at once would be slow and wasteful.
That's where pagination and filtering come in.

By the end of this chapter, you'll learn:

- How to use Django Ninja's built-in pagination
- How to create custom pagination logic
- How to filter and search through your database
- How to combine both effectively

Step 1: Why Pagination?

Without pagination:

GET /users

→ returns 10,000 users

With pagination:

GET /users?limit=10&offset=20

→ returns 10 users (from position 20)

Much faster. Much cleaner.

Step 2: Django Ninja's Built-in Pagination

Example:

```
from ninja import NinjaAPI, Schema
```

```
from ninja.pagination import paginate, PageNumberPagination

api = NinjaAPI()

class UserOut(Schema):
    id: int
    name: str
    email: str

users_db = [
    {"id": i, "name": f"User {i}", "email": f"user{i}@example.com"}
    for i in range(1, 101)
]
@api.get("/users", response=list[UserOut])
@paginate(PageNumberPagination)
def list_users(request):
    return users_db
```

Now when you visit /users?page=1, /users?page=2, etc.,
you'll see 10 results per page (default limit = 10).

Step 3: Customize Pagination Settings

You can create your own pagination class:

```
from ninja.pagination import PageNumberPagination
class CustomPagination(PageNumberPagination):
    page_size = 5
    max_page_size = 50
Then use it:
@api.get("/users", response=list[UserOut])
@paginate(CustomPagination)
def list_users(request):
    return users_db
```

Now each page shows only 5 users.

Step 4: Offset Pagination

Offset pagination is another style:

Instead of pages, it uses numeric offsets.

```
from ninja.pagination import LimitOffsetPagination

@api.get("/users-offset", response=list[UserOut])
@paginate(LimitOffsetPagination)
def list_users_offset(request):
    return users_db
```

Now you can query:

/users-offset?limit=5&offset=10

Step 5: Adding Filtering to Your API

Filtering lets users search or narrow down results.

Example:

```
from typing import Optional
@api.get("/search-users", response=list[UserOut])
def search_users(request, name: Optional[str] = None):
    if name:
        return [u for u in users_db if name.lower() in u["name"].lower()]
    return users_db
```

Now:

```
/search-users?name=User 3
returns:
[{"id": 3, "name": "User 3", "email": "user3@example.com"}]
```

Step 6: Combine Pagination + Filtering

Both can work together perfectly!

```
@api.get("/users/filter", response=list[UserOut])
@paginate(PageNumberPagination)
def filter_users(request, name: Optional[str] = None):
```

```
    data = users_db
    if name:
        data = [u for u in users_db if name.lower() in u["name"].lower()]
    return data
```

Example:

/users/filter?name=User&page=2

returns paginated, filtered results!Step 7: Database Filtering (ORM Way)

When using Django models, filtering becomes even easier.

```
from app.models import User
@api.get("/users-db", response=list[UserOut])
@paginate(PageNumberPagination)
def list_users_from_db(request, name: Optional[str] = None):
    users = User.objects.all()
    if name:
        users = users.filter(name__icontains=name)
    return users
```

Exercise

1. Create an endpoint /transactions that lists all transactions with
2. Add a query parameter type (e.g., deposit, withdrawal) for
3. Combine both pagination and filtering.
4. Test it in Swagger and try different page numbers.

CHAPTER 13

Middleware & CORS

Controlling and Securing Every Request to Your API

The Goal of This Chapter

Every request to your API goes through several steps — preprocessing, authentication, logging, etc.
Middleware gives you full control over this process.

By the end of the chapter, you'll know:

- What middleware is and how it works in Django Ninja
- How to write custom middleware
- How to enable CORS to allow safe access from the frontend/mobile
- How to use third-party CORS libraries

Step 1: What is Middleware?

Middleware is a layer that processes requests before they reach your API endpoints — and responses before they go back to the client.

Think of it as a security guard or filter at the gate.

In Django:

- Middleware is global (applies to all routes)
- You add middleware in the settings file

Step 2: Creating Your First Custom Middleware

Let's track how long each API request takes.

Step 1: Create a file (e.g. middleware.py) in your app:

```
import time

class RequestTimingMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        start_time = time.time()
        response = self.get_response(request)
        duration = time.time() - start_time
        print(f"Request to {request.path} took {duration:.4f}s")
        return response
```

Step 2: Add it to your settings

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    # ...
    'your_app.middleware.RequestTimingMiddleware',
]
```

Step 3: Adding Authorization Globally

You can reject certain requests before they hit any view.

Example: Block all requests from unauthorized IPs.

```
class BlockSpecificIPMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        self.blocked_ips = ["192.168.0.100"]
```

```
    def __call__(self, request):
        if request.META.get('REMOTE_ADDR') in self.blocked_ips:
            from django.http import JsonResponse
            return JsonResponse({"error": "Not allowed"}, status=403)
        return self.get_response(request)
```

Step 4: What is CORS & Why You Need It

CORS = Cross-Origin Resource Sharing

By default, browsers block requests from websites hosted at different domains or ports.

Example:
Frontend: http://localhost:3000
Backend API: http://localhost:8000

Requests will get blocked unless CORS is enabled.

Step 5: Install & Enable Django CORS Headers

Install:

```
pip install django-cors-headers
```

Add to settings.py:

```
INSTALLED_APPS = [
    ...
    'corsheaders',
    ...
]
MIDDLEWARE = [
    'corsheaders.middleware.CorsMiddleware',
    ...
]
```

Allow all origins (for development):

```
CORS_ALLOW_ALL_ORIGINS = True
```

Step 6: Allow Only Specific Origins (Recommended in Production)

```
CORS_ALLOWED_ORIGINS = [
    "https://yourfrontend.com",
    "https://another-allowed-client.com",
]
```

Step 7: Allow Custom Headers & Methods

```
CORS_ALLOW_HEADERS = [
    "content-type",
    "authorization",
    "x-custom-header",
]
CORS_ALLOW_METHODS = [
    "GET",
    "POST",
    "PUT",
    "PATCH",
    "DELETE"
]
```

Now only those methods & headers are allowed.

Exercise

5. Add a middleware to log every API request path & method
6. Enable CORS and allow only your frontend app
7. Try making frontend requests before and after adding CORS

CHAPTER 14

Background Tasks

Running Time-Consuming Work Without Making Users Wait

The Goal of This Chapter

Not all tasks should run while the user waits for a response.

Imagine API tasks like:

- Sending emails
- Processing payment
- Generating reports
- Sending push notifications

These can take time — and can slow down your API responses.

This is where background tasks come in. They let you process work after sending a response, so your API stays fast.

Step 1: What Are Background Tasks?

A background task is a **function that runs after your main API response is returned

The user doesn't wait for it to finish.
Django Ninja supports background tasks through its integration with Starlette.Step 2: Adding a Basic Background Task

Step 1: Import BackgroundTasks:

```
from ninja import NinjaAPI, BackgroundTasks
api = NinjaAPI()
```

Step 2: Define a background task:

```
def send_welcome_email(email: str):
    print(f'Sending welcome email to {email}...")
    # Simulate delay
    import time
    time.sleep(3)
    print("Email sent!")
```

Step 3: Add it to your endpoint:

```
@api.post("/register")
def register_user(request, email: str, background_tasks: BackgroundTasks):
    background_tasks.add_task(send_welcome_email, email)
    return {"message": f"Registration successful for {email}"}
```

What Happens?

When someone calls:

```
POST /register?email=test@example.com
```

The response returns **immediately**:

```
{
  "message": "Registration successful for test@example.com"
}
```

Meanwhile, the email is sent **in the background**.

No lag for the user.

Step 3: Background Tasks With Arguments

You can pass multiple arguments:

```
def generate_report(user_id: int, include_pdf: bool = False):
    print(f'Generating report for user {user_id}...")
    if include_pdf:
        print("PDF version included.")
    print("Report ready!")
Use it like this:
@api.post("/report")
```

```
def generate_user_report(request, user_id: int, background_tasks: BackgroundTasks):
    background_tasks.add_task(generate_report, user_id, True)
    return {"message": "Report generation started"}
```

Step 4: Background Tasks with Database

Yes, you can even use Django ORM inside background tasks.

```
def notify_user(user_id: int):
    from app.models import User
    user = User.objects.get(id=user_id)
    print(f"Notifying {user.email}")
```

Use normally: background_tasks.add_task(notify_user, new_user.id)

Warning: Background Tasks Are Not Persistent

By default, background tasks run in-memory.

If you restart your server, pending tasks will be lost. For serious workloads (like sending massive emails), use **Celery** + Redis.

Optional: Using Celery for Advanced Background Processing

If your application needs distributed, persistent job handling, use Celery.
(Main workflow involves a task queue, worker processes, and a broker like Redis).

We'll cover Celery deeper in the Appendix.

Exercise

1. Add a background task to:
   - Send a welcome SMS
   - Generate a user statistics file
2. Simulate delay using time.sleep()
3. Watch the task run asynchronously in the console while the response

CHAPTER 15

WebSockets Support

Adding Real-Time, Two-Way Communication to Your Ninja API

The Goal of This Chapter

Some applications need instant, live updates — think chat apps, notifications, live dashboards, or multiplayer games.

That's where WebSockets come in — they allow two-way communication between your server and client in real-time.

By the end of this chapter, you'll know:

- How to enable WebSockets support in Django Ninja
- How to manage connections
- How to send and receive messages
- Real-world use cases for live APIs

Step 1: What are WebSockets?

Unlike normal HTTP requests (one-way: client → server), WebSockets allow continuous two-way communication.

Good for:

- Live chat apps
- Live feeds or notifications
- Multiplayer games
- IoT updates
- Stock or crypto price trackers

Step 2: WebSocket Support in Django Ninja

Django Ninja supports WebSockets via **NinjaAPI.ws_route**.

And because it's built on ASGI, it's super-fast for real-time apps.

Step 3: Your First WebSocket Route

```
from ninja import NinjaAPI
api = NinjaAPI()
@api.ws_route("/ws/echo")
async def echo(request, ws):
    await ws.accept()
    while True:
        message = await ws.receive_text()
        await ws.send_text(f"Echo: {message}")
```

What this does:

- Client connects to /ws/echo
- Server accepts connection
- Any message sent by the client is echoed back

Step 4: Testing Your WebSocket

You can test this using:

- A frontend JS WebSocket client
- Browser dev console
- Tools like Postman (WebSocket tab)
- VS Code REST client

Example (Quick Test in Browser Console):

```
let ws = new WebSocket('ws://localhost:8000/ws/echo')
ws.onopen = () => ws.send('Hello Server')
```

```
ws.onmessage = (msg) => console.log(msg.data)
```

Expected Output:

Echo: Hello Server

Step 5: Sending JSON Instead of Text

```
@api.ws_route("/ws/json")
async def json_test(request, ws):
    await ws.accept()
    await ws.send_json({"message": "Connected!"})
    while True:
        data = await ws.receive_json()
        await ws.send_json({"you_sent": data})
```

When client sends:

```
{"name": "Alice"}
```

Response is:

```
{"you_sent": {"name": "Alice"}}
```

Step 6: Broadcasting to Multiple Clients

To build something like a live chat, you can store connections:

```
connections = []

@api.ws_route("/ws/broadcast")
async def broadcast(request, ws):
    await ws.accept()
    connections.append(ws)
    try:
        while True:
            message = await ws.receive_text()
            for conn in connections:
                await conn.send_text(f"Broadcast: {message}")
    except:
        connections.remove(ws)
```

Result:

- All connected clients receive the same message
- Works like a room chat

Important Notes

- WebSockets require async functions
- They stay open until the client/server closes it
- Use proper cleanup to avoid memory leaks

Real-World Use Cases

Use Case	How WebSockets Help
Chat App	Live message updates
Notifications	Instant alerts on UI
Live Dashboard	Push real-time numbers
IoT	Device status updates
Games	Live moves across players

Exercise

4. Create a WebSocket endpoint that counts the number of connected
5. Build a simple \"real-time notification\" WebSocket that sends a
6. Test using browser console or Postman

CHAPTER 16

API Documentation & OpenAPI

Automatically Document Your API Like a Pro — No Extra Work!

The Goal of This Chapter

- API What API documentation is and why it matters
- How Django Ninja generates interactive docs
- How to customize the docs with metadata
- How you can share and version the docs easily

documentation helps both developers and clients understand how to consume your API — and Django Ninja automatically provides this out of the box using OpenAPI and Swagger UI.

By the end of this chapter, you'll understand:

Step 1: What is OpenAPI?

The OpenAPI Specification is a standard format for describing REST APIs.

- Machine-readable API specs
- Human-readable interactive docs (via Swagger UI or Redoc

Step 2: Automatic Documentation

As soon as you define your API using Ninja:

You AUTOMATICALLY get API documentation at:

- /docs
- /openapi.json
- /redoc

Example:

http://localhost:8000/api/docs

Try it now — you\'ll see your hello endpoint clearly displayed!

Step 3: Swagger UI (Interactive Docs)

The /docs route uses Swagger UI, which allows you to:

- See every endpoint
- Try requests live
- View request/response models
- Explore the API structure

It's interactive, easy to explore, and client-friendly.

Step 4: Customizing Your API Metadata

You can customize the title, version, and description of your API:

```
api = NinjaAPI(
    title="My Awesome API",
    version="1.0.0",
    description="This API does amazing things using Django Ninja!"
)
```

It shows up at the top of the API docs

Step 5: Adding Authentication to Docs

If you're using authentication, Django Ninja includes it automatically in the docs.

Example using Bearer Tokens:

```
from ninja.security import HttpBearer

class Auth(HttpBearer):
    def authenticate(self, request, token):
        if token == "supersecret":
            return token

api = NinjaAPI(auth=Auth())
```

The docs will now show an Authorize button for adding the token.

Step 6: Dependency Documentation

If you\'re using dependencies or security, Django Ninja includes them in the docs:

```
def get_token(request):
    token = request.headers.get("X-TOKEN")
    if token != "secret":
        raise Exception("Invalid token")
    return token
@api.get("/secure-data")
def secure_data(request, token=Depends(get_token)):
    return {"data": "Top secret"}
```

The doc will show:

- Required headers
- Security dependencies

Step 7: Exporting OpenAPI Spec

You can share the full API documentation spec as JSON:

- Visit: /openapi.json

This helps with:

- API client generation
- Documentation syncing
- Sharing specs with other teams

Bonus: ReDoc UI

Want a different look for your API docs?
It's more static and suitable for public-facing docs.

Exercise

7. Add a custom title and description to your API
8. Add a secure endpoint and test how it shows in the docs
9. Export your OpenAPI spec and view as JSON
10. Try /redoc and compare with /docs

CHAPTER 17

CORS, Deployment & Production Setup

Turn your Django Ninja API from Local Project to Live, Secure Service

The Goal of This Chapter

After development and testing, the next step is Deployment — taking your API live so others can use it. In this chapter, you'll learn:

- What CORS is and how to enable it
- How to prepare your Django project for production
- Best practices for environment variables and secret keys
- How to deploy to common platforms like Render, Railway, or AWS
- Tips for debugging and maintaining your deployed API

Step 1: What is CORS and Why It Matters

CORS (Cross-Origin Resource Sharing) controls which websites are allowed to make requests to your API.

Example:

- Your backend is at api.myapp.com
- Your frontend is at myfrontend.com

Without CORS configured, browsers will block requests.

Install CORS Headers for Django

```
pip install django-cors-headers
```

Add to settings.py

```
INSTALLED_APPS = [
```

```
    ...
    "corsheaders",
]

MIDDLEWARE = [
    "corsheaders.middleware.CorsMiddleware",
    ...
]

CORS_ALLOWED_ORIGINS = [
    "https://example-frontend.com",
    "http://localhost:3000",
]
```

You can also allow all for testing but never do this in production:

CORS_ALLOW_ALL_ORIGINS = True

Step 2: Security Checklist for Production

Before going live, make sure these are in place:

1. SECRET_KEY

Never hardcode your SECRET_KEY. Use environment variables:

SECRET_KEY = os.getenv("SECRET_KEY") Use

a .env file or your hosting platform's settings.

2. DEBUG Mode

Turn off debug in production:

DEBUG = False

3. Allowed Hosts

```
ALLOWED_HOSTS = ["yourdomain.com", "api.yourdomain.com"]
```

4. HTTPS

Make sure all requests are encrypted with HTTPS.

5. Database Configuration

Use a production-grade database like PostgreSQL.

Example for DATABASES:

```
DATABASES = {
  'default': {
    'ENGINE': 'django.db.backends.postgresql',
    'NAME': os.getenv("DB_NAME"),
    'USER': os.getenv("DB_USER"),
    'PASSWORD': os.getenv("DB_PASSWORD"),
    'HOST': os.getenv("DB_HOST"),
    'PORT': 5432,
  }
```

Step 3: Deployment Options

Here's a quick overview of where you can deploy:

Render.com (Beginner-friendly)

- Supports Django directly
- Automatic deploys from GitHub
- Free tier available (can sleep after inactivity)

Steps:

11. Push your code to GitHub
12. Create a Django web service in Render
13. Add your environment variables
14. Deploy

Railway.app (Easy CI/CD)

- Similar to Render

- Supports PostgreSQL built-in

AWS Elastic Beanstalk (Advanced Level)

- Highly scalable
- Requires extra config for networking, load balancing

Docker + VPS (For professionals)

- Build Docker image
- Use servers like DigitalOcean, Linode
- Full control

Step 4: Deploy with Render

Example: render.yaml

```
services:
 - type: web
   name: django-ninja-api
   env: python
   buildCommand: pip install -r requirements.txt
   startCommand: gunicorn project_name.wsgi:application
   envVars:
    - key: SECRET_KEY
      sync: false
```

Step 5: Final Deployment Checklist

15. CSS & static files collected: python manage.py collectstatic
16. Database migrated: python manage.py migrate
17. Logs available for monitoring
18. Test API with external tools like Postman

Exercise

19. Install and configure django-cors-headers
20. Set up a .env file with your SECRET_KEY
21. Try deploying your API on Render.com
22. Inspect your live /docs endpoint

CHAPTER 18

Bonus Tools, Tips & Tricks for Django Ninja

Accelerate Your Workflow and Master Django Ninja Like a Pro

The Goal of This Chapter

Go beyond the basics! In this chapter, we\'ll explore:

- Tools to speed up API development
- Auto-generating client SDKs
- Structuring large projects
- API versioning
- Reusability and modular code techniques
- Useful third-party tools for Django Ninja

1. API Versioning (Maintain Multiple Versions)

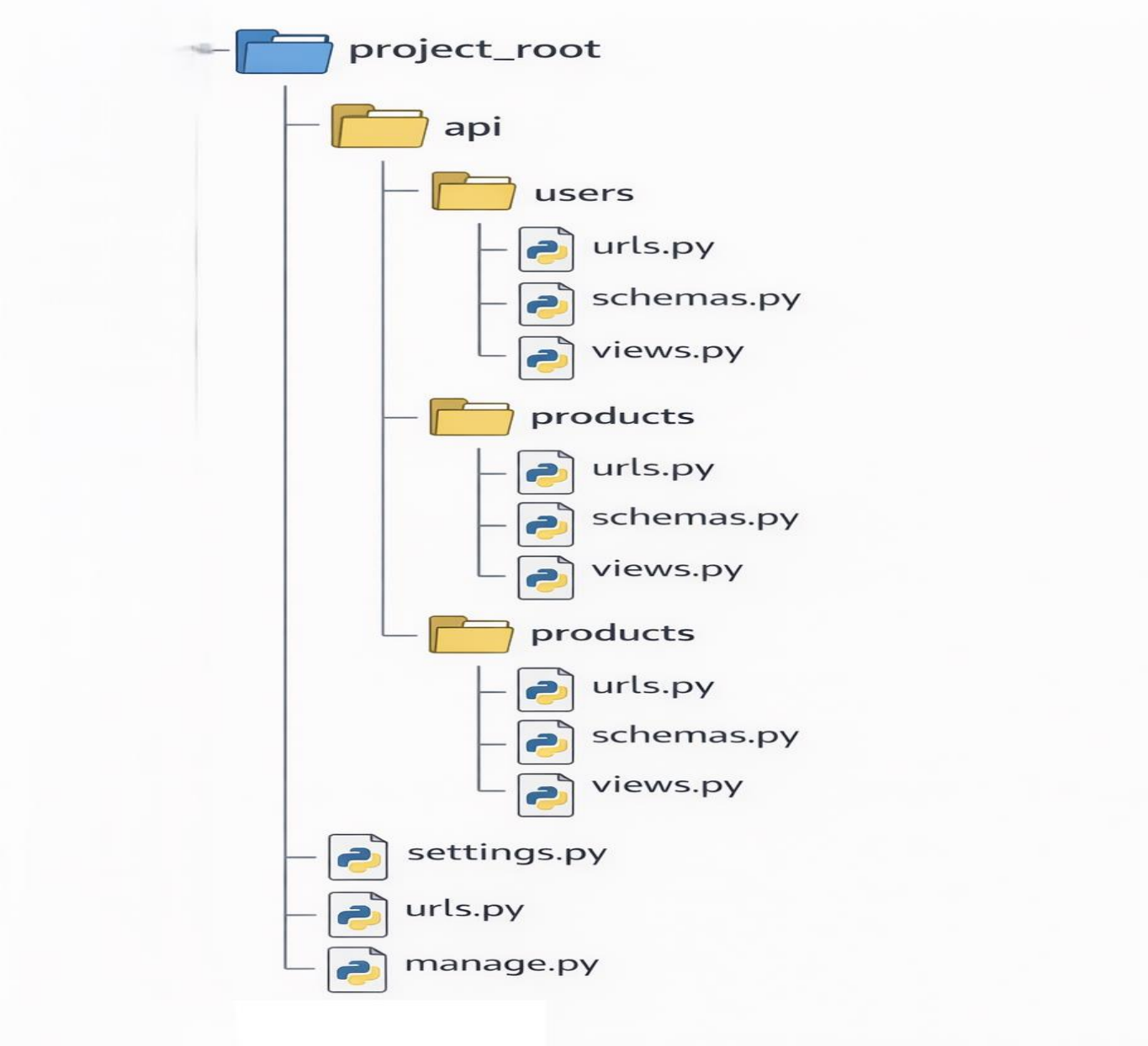
Versioning allows you to run multiple APIs (v1, v2, etc.) side-by-side for backward compatibility.

```
from ninja import NinjaAPI
api_v1 = NinjaAPI(version="1.0.0", title="My API v1")
api_v2 = NinjaAPI(version="2.0.0", title="My API v2")
Your URLs might look like:
/api/v1/docs
```

2. Structuring Large Projects

When your API grows, it\'s important to modularize the code:

Recommended folder structure:



3. Live API Documentation with ReDoc & Swagger

Ninja automatically provides documentation:

- Swagger UI → http://localhost:8000/api/docs
- ReDoc → http://localhost:8000/api/redoc

4. Auto-generate API Client SDKs

You can generate client code (for JavaScript, Android, etc.) with tools like:

- OpenAPI Generator
- Swagger Codegen

openapi-generator-cli generate -i http://localhost:8000/api/docs.json -g

This helps frontend/mobile teams use your API faster.

5. Useful Third-party Packages

These tools integrate beautifully with Django Ninja:

- django-ninja-extra: class-based views, DI
- django-ninja-jwt: Simple JWT support
- drf-yasg: advanced doc generation
- django-cors-headers: CORS management
- django-stubs: type hinting support

6. Use .env + Settings Management (Tip!)

Use a library like python-decouple to manage settings cleanly:

```
pip install python-decouple
from decouple import config
SECRET_KEY = config("SECRET_KEY")
DEBUG = config("DEBUG", default=False, cast=bool)
```

Exercise

23. Add a second version of your API using .add_router
24. Generate a client SDK using OpenAPI Generator
25. Try reorganizing your app to follow the modular pattern
26. Install and configure django-ninja-extra or django-ninja-jwt

C H A P T E R 1 9

Capstone Project -- Banking API System

Build a Complete Real-World API Using Django Ninja

The Chapter Goal

In this capstone project, you'll build a fully functional API for a basic banking system. This will combine everything we\'ve learned:

- Models & ORM
- CRUD operations
- Authentication
- Permissions & Roles
- Transactions
- Validation & Error handling
- Testing
- Documentation

System Overview

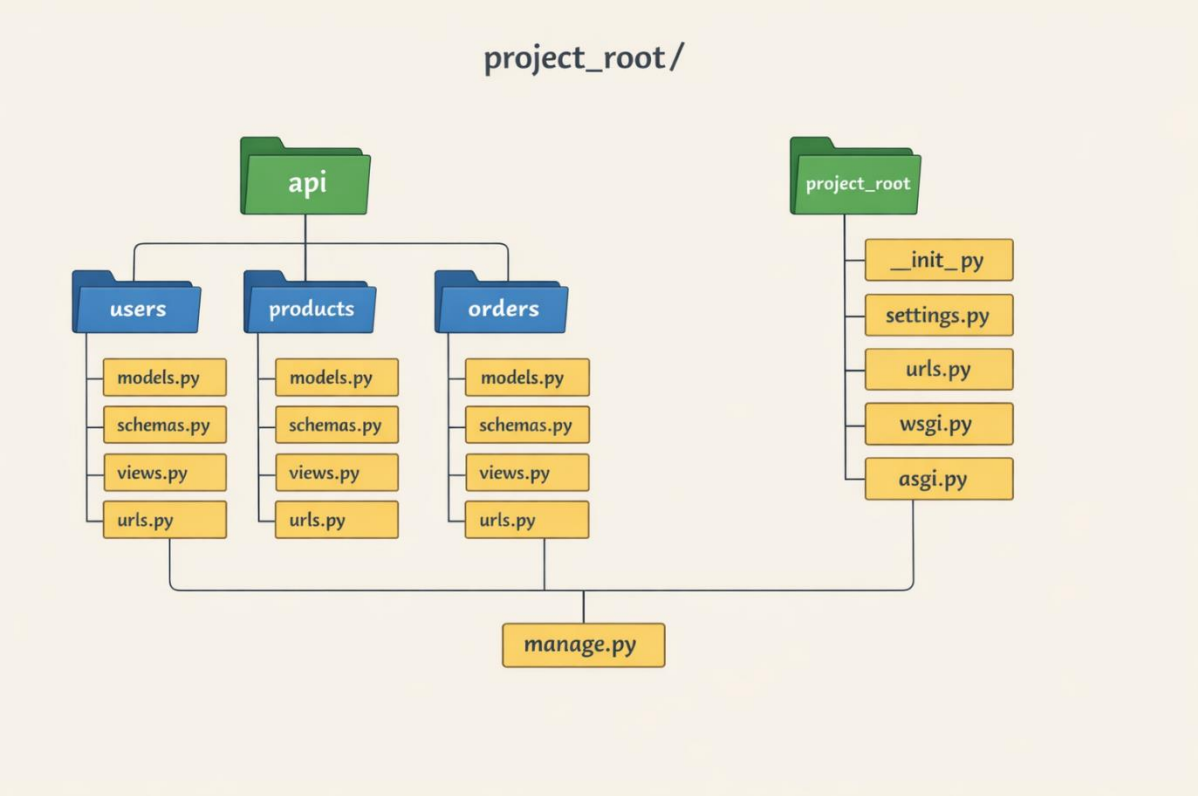
The Banking API will support:

Feature	Description
User Registration	Users can register and login
Account Management	Users can create bank accounts
Deposit & Withdrawal	Transactions affecting account balance
Balance Check	Check account balance anytime
Transaction History	List recent activities

Role-Based Access	Admin vs user permissions

Folder Structure

Here's the folder structure (matching your new graphic):



Step 1: Create Django Project and Apps

Add sub-apps manually:

- accounts

- authentication
- transactions

Step 2: Define Models (accounts/models.py)

```
from django.db import models
from django.contrib.auth.models import User

class Account(models.Model):
    owner = models.OneToOneField(User, on_delete=models.CASCADE)
    account_number = models.CharField(max_length=10, unique=True)
    balance = models.DecimalField(max_digits=12, decimal_places=2, default=0.00)
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"{self.owner.username}'s account"
```

Step 3: Schemas (accounts/schemas.py)

```
from ninja import Schema

class AccountCreate(Schema):
    account_number: str

class AccountOut(Schema):
    account_number: str
    balance: float
```

Step 4: Views & Endpoints (accounts/views.py)

```
from ninja import Router
from .models import Account
from .schemas import AccountCreate, AccountOut
from django.contrib.auth.models import User
from django.shortcuts import get_object_or_404
```

```
router = Router()

@router.post("/create", response=AccountOut)
def create_account(request, payload: AccountCreate):
    user = request.user
    new_account = Account.objects.create(
        owner=user,
        account_number=payload.account_number,
        balance=0.00
    )
    return new_account

@router.get("/balance", response=AccountOut)
def get_balance(request):
    account = get_object_or_404(Account, owner=request.user)
    return account
```

Step 5: Add Transaction Logic

Create separate app /transactions that handles:

- Deposit
- Withdrawal
- Listing transactions

Example model (transactions/models.py):

```
class Transaction(models.Model):
    account = models.ForeignKey(Account, on_delete=models.CASCADE)
    amount = models.DecimalField(max_digits=12, decimal_places=2)
    type = models.CharField(max_length=10, choices=[("deposit", "Deposit"), ("withdraw", "Withdraw")])
    timestamp = models.DateTimeField(auto_now_add=True)
```

Step 6: Authentication & Permissions

Use django-ninja-jwt or your simple session-based approach from earlier chapters to protect endpoints.

Example (protect all account endpoints):

```
from ninja.security import django_auth
router = Router(auth=django_auth)
```

Step 7: Test With Swagger UI

Once you wire everything up in urls.py and start the server, visit:

http://127.0.0.1:8000/api/docs
You'll interact with your API visually from your browser.

Advanced Features (Optional Challenges)

- Add email notifications on transactions
- Add parallel transfer support
- Use Celery for background tasks
- Add permissions (e.g. staff-only deposits)
- Implement pagination for transaction history

Capstone Exercises

1. Create your own API endpoint to transfer funds between accounts
2. Add admin-only access to an endpoint that lists all users' accounts
3. Write a test script using the Django test client
4. Package and deploy your Banking API on a cloud platform

CHAPTER 20

Project -- Hospital Management System API

A full-featured healthcare backend with Django Ninja

Project Objectives

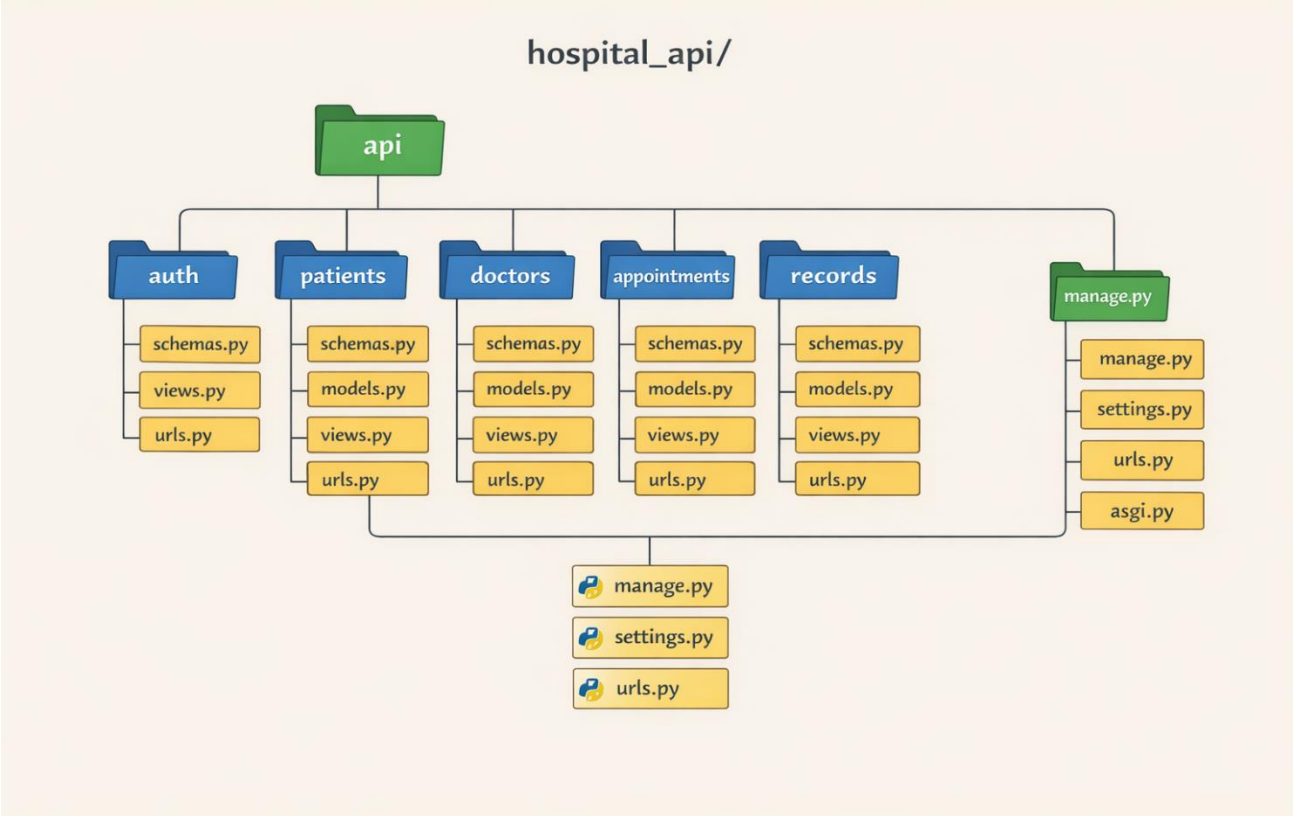
By the end of this project, you will:

- Build and organize a multi-module API
- Integrate user roles and permissions
- Handle nested relationships (e.g., doctor → appointments → patients)
- Implement CRUD with auth and filtering
- Enable a clean API layer ready for mobile/IoT integration

Features Overview

Module	Features Included
Authentication	Patient vs. Staff vs. Admin roles
Patient Management	Register, update, view medical history
Doctor Management	Create doctor profiles, assign patients
Appointments	Book, update, cancel, check availability
Medical Records	Create, view, and manage a patient's record
Prescriptions	Doctors can prescribe medication to patients
(Optional) Pharmacy	Track drug prescriptions and availability
(Optional) Billing	Generate invoices and track payments

Proposed Folder Structure



Step-by-Step Workflow

Step 1: Create the Project & Apps

```
django-admin startproject hospital_api
cd hospital_api
python manage.py startapp api
```

Inside api/, we'll create the sub-apps: patients, doctors, appointments, records.

Step 2: Models Example

Let's create a few simplified models to start with.

api/patients/models.py

```
from django.db import models
from django.contrib.auth.models import User
```

```
class Patient(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    dob = models.DateField()
    address = models.TextField()
    phone = models.CharField(max_length=20)
    created_at = models.DateTimeField(auto_now_add=True)
```

api/doctors/models.py

```
class Doctor(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    specialization = models.CharField(max_length=100)
    room_number = models.CharField(max_length=10)
```

api/appointments/models.py

```
class Appointment(models.Model):
    patient = models.ForeignKey("patients.Patient", on_delete=models.CASCADE)
    doctor = models.ForeignKey("doctors.Doctor", on_delete=models.CASCADE)
    date = models.DateTimeField()
    status = models.CharField(
        max_length=10,
        choices=[("scheduled", "Scheduled"), ("completed", "Completed"), ("cancelled",
"Cancelled")],
        default="scheduled"
    )
    notes = models.TextField(blank=True)
```

APIs With Schemas (Example)

api/appointments/schemas.py

```
from ninja import Schema
from datetime import datetime
from typing import Optional

class AppointmentIn(Schema):
```

```
    doctor_id: int
    date: datetime
    notes: Optional[str] = None

class AppointmentOut(Schema):
    id: int
    patient_id: int
    doctor_id: int
    date: datetime
    status: str
    notes: Optional[str]
```

Example Endpoint: Booking an Appointment

api/appointments/views.py

```
from ninja import Router
from .models import Appointment
from .schemas import AppointmentIn, AppointmentOut
from django.shortcuts import get_object_or_404
from api.patients.models import Patient
from api.doctors.models import Doctor

router = Router(tags=["appointments"])

@router.post("/", response=AppointmentOut)
def create_appointment(request, payload: AppointmentIn):
    patient = get_object_or_404(Patient, user=request.user)
    doctor = get_object_or_404(Doctor, id=payload.doctor_id)
    appointment = Appointment.objects.create(
        patient=patient,
        doctor=doctor,
        date=payload.date,
        notes=payload.notes
```

```
    )
    return appointment
```

Security & Roles

You should extend User for both Patient and Doctor and use

- Patient role: Limited to managing their own records and appointments
- Doctor role: View assigned patients, medical records, create
- Admin role: Full CRUD over all data

Example using Ninja\'s auth:

```
from ninja.security import django_auth
router = Router(auth=django_auth)
```

Stretch Features to Try

- Add an admin dashboard endpoint to see system metrics
- Create a prescription model and attach to patient records
- Add a calendar view for doctor schedules
- Implement email or SMS notifications for appointments
- Add JWT-based login system
- Add pagination and filtering

Deployment Tips

- Use PostgreSQL for real deployment
- Configure CORS for front-end (React, Angular, mobile apps)
- Set up Nginx + Gunicorn for hosting
- Use Swagger UI (/api/docs) as a client testing layer

CHAPTER 21

Personal Task Manager API

Build & Manage Tasks with CRUD Operations, Authentication, and Filtering

The Chapter Goal

In this chapter, you will build a fully functional Task Manager API, where users can:

- Create accounts
- Login and manage their tasks
- Add, update, delete tasks
- Mark tasks as complete
- Filter tasks (by date, status, or priority)

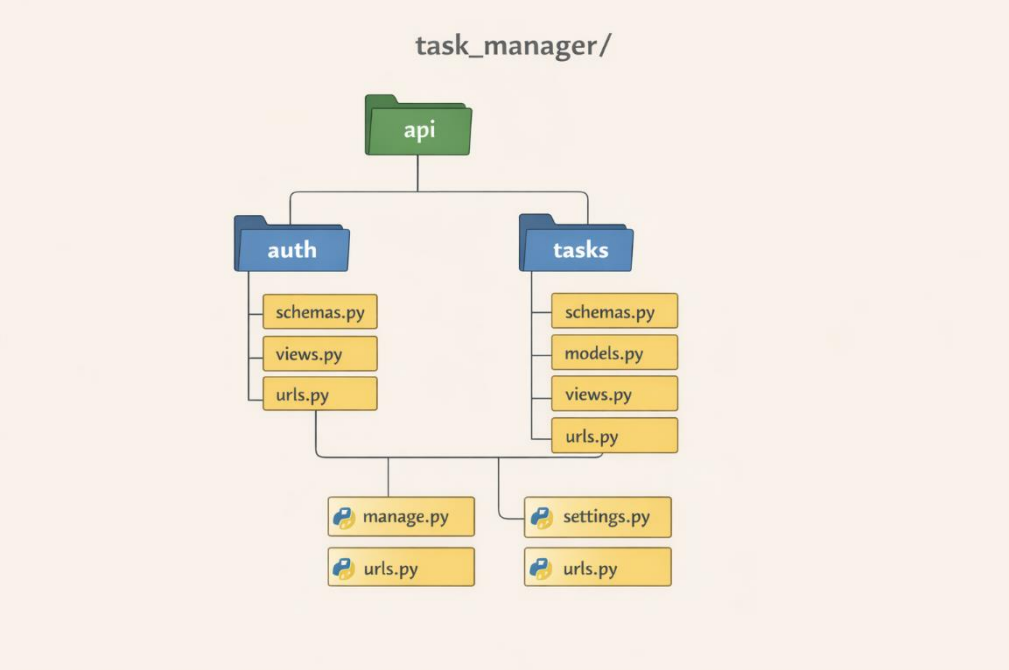
You'll use everything you\'ve learned so far, including schemas, views, routers, authentication, pagination, and filtering. Let\'s get started!

API Features Overview

Feature	Description
User Registration	Signup and login to the system
Task CRUD	Create, read, update, delete personal tasks
Status & Priority	Mark tasks as done & assign priority
Filtering	Filter by status, priority, or date
Pagination	List tasks in a paginated format
Documentation	Fully documented via Swagger UI
Security	Token or session authentication

Project Structure

We'll reuse the same modular folder structure as before:



Step 1: Create Django Project & App

```
django-admin startproject task_manager
cd task_manager
python manage.py startapp api
```

Step 2: Task Model (api/tasks/models.py)

```
from django.db import models
from django.contrib.auth.models import User

class Task(models.Model):
    owner = models.ForeignKey(User, on_delete=models.CASCADE)
    title = models.CharField(max_length=120)
    description = models.TextField(blank=True)
    priority = models.CharField(
        max_length=10,
        choices=[("low", "Low"), ("medium", "Medium"), ("high", "High")],
        default="medium"
```

```
    )
    status = models.CharField(
        max_length=10,
        choices=[("pending", "Pending"), ("done", "Done")],
        default="pending"
    )
    created_at = models.DateTimeField(auto_now_add=True)
    due_date = models.DateTimeField(null=True, blank=True)

    def __str__(self):
        return self.title
```

Step 3: Schemas for Task API (api/tasks/schemas.py)

```
from ninja import Schema
from datetime import datetime
from typing import Optional


class TaskIn(Schema):
    title: str
    description: Optional[str]
    priority: str = "medium"
    status: str = "pending"
    due_date: Optional[datetime]


class TaskOut(Schema):
    id: int
    title: str
    status: str
    priority: str
    created_at: datetime
    due_date: Optional[datetime]
```

Step 4: Task Views (api/tasks/views.py)

```
from ninja import Router
from .models import Task
```

```
from .schemas import TaskIn, TaskOut
from django.shortcuts import get_object_or_404

router = Router(tags=["tasks"])

@router.post("/", response=TaskOut)
def create_task(request, payload: TaskIn):
    task = Task.objects.create(owner=request.user, **payload.dict())
    return task

@router.get("/", response=list[TaskOut])
def list_tasks(request, status: str = None, priority: str = None):
    tasks = Task.objects.filter(owner=request.user)
    if status:
        tasks = tasks.filter(status=status)
    if priority:
        tasks = tasks.filter(priority=priority)
    return tasks

@router.get("/{task_id}", response=TaskOut)
def get_task(request, task_id: int):
    task = get_object_or_404(Task, owner=request.user, id=task_id)
    return task

@router.put("/{task_id}", response=TaskOut)
def update_task(request, task_id: int, payload: TaskIn):
    task = get_object_or_404(Task, owner=request.user, id=task_id)
    for attr, value in payload.dict().items():
        setattr(task, attr, value)
    task.save()
    return task
```

```
@router.delete("/{task_id}")
def delete_task(request, task_id: int):
    task = get_object_or_404(Task, owner=request.user, id=task_id)
    task.delete()
    return {"success": True}
```

Step 5: Add Authentication

You can reuse your JWT or session-based setup:

```
from ninja.security import django_auth
router = Router(auth=django_auth)
```

Step 6: Add URL Routing (in main urls.py)

```
from django.contrib import admin
from django.urls import path
from ninja import NinjaAPI
from api.tasks.views import router as tasks_router
from api.auth.views import router as auth_router

api = NinjaAPI()
api.add_router("/tasks", tasks_router)
api.add_router("/auth", auth_router)

urlpatterns = [
    path("admin/", admin.site.urls),
    path("api/", api.urls),
]
```

Testing With Swagger UI

Once the server is running, test your API with all endpoints under:

http://127.0.0.1:8000/api/docs

Check:

- Create a task

- List all your tasks
- Update one task
- Delete a task
- Filter by priority or status

Optional Stretch Goals

- Add task reminders via email
- Notify when a task is overdue
- Add search by keyword
- Add user avatars and profiles
- Assign tasks to teammates

Capstone Exercise

1. Build the TaskManager API by following the steps
2. Add at least one filtering and one pagination feature
3. Secure all task endpoints with authentication
4. Deploy your app and share the live link + Swagger docs
5. Push your code to GitHub as a project portfolio piece

C H A P T E R 2 2

Deployment & Scaling Django Ninja APIs

Step-by-step to go from local dev → production, and scale safely

What you'll get in this chapter

- Production checklist (quick-read)
- Two primary deployment methods (server + Docker) with full config
- Database (Postgres) setup and connection details
- Serving static files (collectstatic) and media files strategy
- Process management (Gunicorn + systemd) + Nginx reverse proxy
- Docker Compose example (Postgres + Redis + Celery + Nginx +
- Kubernetes notes (basic manifests for scaling)
- Background tasks (Celery + Redis) and how to run them in production
- Caching (Redis) & connection pooling hints
- Zero-downtime deploys, migrations, health checks
- Security, secrets, logging, backups, monitoring
- Troubleshooting checklist & exercises

Production Checklist (quick)

Before you deploy, ensure:

1. DEBUG = False in production.
2. SECRET_KEY stored in environment variable (never in repo).
3. ALLOWED_HOSTS configured.
4. Database (Postgres) provisioned and reachable.
5. Static files handled (collectstatic) and served by Nginx/CDN.
6. Gunicorn (or ASGI server) configured to run the app.

7. Reverse proxy (Nginx) in front to serve static files & TLS.
8. HTTPS (Let\'s Encrypt) configured.
9. Logging & monitoring in place.
10. Backups & migrations strategy defined.
11. Health checks and readiness endpoints.
12. Celery workers (if used) and Redis configured.
13. Minimal required ports open, firewall rules enforced.

1. Prepare your Django project for production

1.1 Settings best practices (settings.py)

1.1 Settings best practices (settings.py)

Move sensitive values to environment variables. Example pattern:

```
# settings.py (fragment)
import os
from pathlib import Path

BASE_DIR = Path(__file__).resolve().parent.parent

SECRET_KEY = os.environ.get("DJANGO_SECRET_KEY")
DEBUG = os.environ.get("DJANGO_DEBUG", "False") == "True"

ALLOWED_HOSTS = os.environ.get("DJANGO_ALLOWED_HOSTS",
"localhost").split(",")

# Database via env (Postgres)
DATABASES = {
   "default": {
      "ENGINE": "django.db.backends.postgresql",
      "NAME": os.environ.get("POSTGRES_DB"),
      "USER": os.environ.get("POSTGRES_USER"),
      "PASSWORD": os.environ.get("POSTGRES_PASSWORD"),
```

```
        "HOST": os.environ.get("POSTGRES_HOST", "localhost"),
        "PORT": os.environ.get("POSTGRES_PORT", "5432"),
    }
}

# Static files
STATIC_ROOT = BASE_DIR / "staticfiles"
MEDIA_ROOT = BASE_DIR / "mediafiles"

# Security
SECURE_PROXY_SSL_HEADER = ("HTTP_X_FORWARDED_PROTO", "https")
SESSION_COOKIE_SECURE = True
CSRF_COOKIE_SECURE = True
```

Important: Do not commit secret keys or production DB credentials.

2. Database: Postgres setup & config

2.1 Provision a Postgres DB

Use a managed DB (recommended): AWS RDS, DigitalOcean Managed DB, Render DB, Railway Postgres. Or provision on your server with:

Ubuntu example (not for large production)

```
sudo apt update
sudo apt install postgresql postgresql-contrib
sudo -u postgres createuser --interactive   # create user
sudo -u postgres createdb mydb
```

Create a database and a role, then set a strong password. Example SQL:

CREATE ROLE ninja_user WITH LOGIN PASSWORD 'verystrongpassword';

CREATE DATABASE ninja_db OWNER ninja_user;

GRANT ALL PRIVILEGES ON DATABASE ninja_db TO ninja_user;

2.2 Connection string / DATABASE_URL (recommended)

Many platforms use a single DATABASE_URL env var:

postgres://USER:PASSWORD@HOST:PORT/DBNAME

You can parse this in Django using dj-database-url:

```
# pip install dj-database-url
import dj_database_url
DATABASES['default'] =
dj_database_url.config(default=os.environ.get("DATABASE_URL"))
```

3. Static files and media

Run collectstatic during deploy:

```
python manage.py collectstatic --noinput
```

Serve STATIC_ROOT and MEDIA_ROOT with Nginx (not Django). Optionally push static files to a CDN or S3 for scale.

Nginx snippet (later full example) will map location /static/ to STATIC_ROOT.

4. Option A: Deploy to a single VPS (Gunicorn + Nginx + systemd)

This is classic, understandable, and good for small teams.

4.1 Install system dependencies

(Ubuntu example)

```
sudo apt update
sudo apt install python3-pip python3-venv nginx
```

4.2 Create virtualenv & install requirements

```
cd /srv
sudo mkdir django_ninja_app
sudo chown $USER:$USER django_ninja_app
cd django_ninja_app

python3 -m venv venv
```

```
source venv/bin/activate
pip install -U pip
pip install -r requirements.txt
```

4.3 Gunicorn systemd service

```
Create /etc/systemd/system/gunicorn.service:
[Unit]
Description=gunicorn daemon for django ninja
After=network.target

[Service]
User=www-data
Group=www-data
WorkingDirectory=/srv/django_ninja_app
Environment="DJANGO_SETTINGS_MODULE=project.settings"
Environment="PYTHONUNBUFFERED=1"
Environment="DJANGO_SECRET_KEY=... (set via systemd or env file)"
ExecStart=/srv/django_ninja_app/venv/bin/gunicorn \
  project.wsgi:application \
  --name ninja_app \
  --bind unix:/run/gunicorn.sock \
  --workers 3 \
  --threads 2 \
  --timeout 120

[Install]
WantedBy=multi-user.target
Reload & start:
sudo systemctl daemon-reload
sudo systemctl start gunicorn
sudo systemctl enable gunicorn
```

4.4 Nginx reverse proxy

```
Create /etc/nginx/sites-available/django_ninja:
```

```
server {
    listen 80;
    server_name yourdomain.com www.yourdomain.com;

    location /static/ {
        alias /srv/django_ninja_app/staticfiles/;
    }

    location /media/ {
        alias /srv/django_ninja_app/mediafiles/;
    }

    location / {
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        proxy_redirect off;
        proxy_pass http://unix:/run/gunicorn.sock;
    }
}
```

Enable site and reload:

```
sudo ln -s /etc/nginx/sites-available/django_ninja /etc/nginx/sites-enabled
sudo nginx -t
sudo systemctl restart nginx
```

4.5 HTTPS with Let's Encrypt

Install certbot and request cert:

```
sudo apt install certbot python3-certbot-nginx
sudo certbot --nginx -d yourdomain.com -d www.yourdomain.com
```

Certbot will update your Nginx config to redirect HTTP → HTTPS and manage renewals.

5. Option B: Docker Compose (recommended for reproducible deploys)

Here’s a full docker-compose.yml (production-ish, small-scale) that includes Postgres, Redis, web (Gunicorn), worker (Celery), and Nginx.

```
version: '3.8'
services:
 db:
  image: postgres:15
  environment:
   POSTGRES_DB: ninja_db
   POSTGRES_USER: ninja_user
   POSTGRES_PASSWORD: supersecret
  volumes:
   - postgres_data:/var/lib/postgresql/data
  networks:
   - ninja-net

 redis:
  image: redis:7
  command: redis-server --appendonly yes
  volumes:
   - redis_data:/data
  networks:
   - ninja-net
 web:
  build: .
  command: gunicorn project.wsgi:application -w 4 -k gthread -b 0.0.0.0:8000
  volumes:
   - .:/app
  env_file:
   - .env.production
  depends_on:
   - db
   - redis
```

```
    networks:
      - ninja-net
  worker:
    build: .
    command: celery -A project worker -l info
    env_file:
      - .env.production
    depends_on:
      - db
      - redis
    networks:
      - ninja-net

  nginx:
    image: nginx:stable
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./deploy/nginx/conf.d:/etc/nginx/conf.d
      - ./staticfiles:/staticfiles
      - ./mediafiles:/mediafiles
    depends_on:
      - web
    networks:
      - ninja-net

volumes:
  postgres_data:
  redis_data:

networks:
```

```
  ninja-net:
```

Dockerfile (simple):

```
FROM python:3.11-slim
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

WORKDIR /app
COPY requirements.txt /app/
RUN pip install --upgrade pip && pip install -r requirements.txt

COPY . /app
RUN adduser --disabled-password djuser
USER djuser

CMD ["gunicorn", "project.wsgi:application", "-w", "4", "-b", "0.0.0.0:8000"]
```

Important files:

- .env.production should contain all env vars (DATABASE_URL, SECRET,
- deploy/nginx/conf.d/default.conf should proxy to web:8000 and serve

/static from mounted ./staticfiles.

Commands:

```
# Build & start
docker-compose up -d --build

# Run migrations
docker-compose exec web python manage.py migrate
docker-compose exec web python manage.py collectstatic --noinput

# Create superuser
docker-compose exec web python manage.py createsuperuser
```

docker-compose exec web python manage.py createsuperuser

This approach is reproducible, easy to scale with multiple web replicas, and portable.

5. Background tasks: Celery + Redis

5.1 Install & configure Celery

```
pip install celery redis
project/celery.py:
import os
from celery import Celery
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "project.settings")
app = Celery("project")
app.config_from_object("django.conf:settings", namespace="CELERY")
app.autodiscover_tasks()
project/__init__.py:
from .celery import app as celery_app
__all__ = ("celery_app",)
Django task example app/tasks.py:
from celery import shared_task

@shared_task
def add(a, b):
   return a + b
```

Start worker with celery -A project worker -l info (Docker Compose example above uses this).

5.2 Beat (periodic tasks)

If you need scheduled jobs:

celery -A project beat -l info Or use flower for monitoring.

6 Scaling strategies

Vertical vs Horizontal

- Vertical: bigger machine (more CPU / RAM). Easy, limited.
- Horizontal: multiple web worker instances behind a load balancer (recommended for scale).

Scale the web layer

- With Docker Compose, run multiple web containers behind Nginx or
- In Kubernetes, scale Deployments: kubectl scale deployment web --replicas=5.

Database scaling

- Use replica/read-replicas for heavy read workloads.
- Use connection pooling (pgbouncer) to reduce DB connection overhead.
- Use managed DB service for automated backups, failover.

Caching

- Use Redis for caching expensive queries, sessions, or rate limiting.
- Example: cache response for 60s:

```
@cache_page(60)
...
```

Horizontal autoscaling (cloud)

- Set autoscaling rules (CPU, request latency) on Kubernetes or cloud

7 Kubernetes (brief, for production-grade scaling)

```
Minimal Deployment (web) YAML (conceptual):
apiVersion: apps/v1
kind: Deployment metadata:
name: web spec: replicas: 3 selector: matchLabels:
app: ninja-web template: metadata: labels:
app: ninja-web spec: containers:
```

```
- name: web
image: yourrepo/ninja:latest ports:
- containerPort: 8000 envFrom:
- secretRef:

name: ninja-secrets readinessProbe:
httpGet:

path: /health port: 8000 livenessProbe:
httpGet:

path: /health port: 8000
```

Service + Ingress sits in front. Use HorizontalPodAutoscaler for CPU-based autoscaling. Use managed Kubernetes (EKS/GKE/AKS) for easier operations.

8 Migrations & zero-downtime deploys

Safe migration strategy

- Add new column with nullable default in code first, deploy, then
- Avoid breaking migrations that require heavy locks (e.g., ALTER

Zero-downtime tip

- Use rolling deployments (Kubernetes) or multiple app instances; apply DB migrations on a single maintenance pod/step before switching traffic to new pods; or use feature flags

9 Health checks & readiness endpoints

Add endpoints that return simple status:

```
from ninja import NinjaAPI
api = NinjaAPI()

@api.get("/health")
def health(request):
    return {"status": "ok"}
```

Nginx or LB can probe /health for readiness/liveness.

10 Security best practices

- DEBUG=False.
- Use HTTPS (Let\'s Encrypt / managed TLS).
- Keep SECRET_KEY secret.
- Use strong DB passwords and restrict IP access if possible.
- Use up-to-date packages (dependabot).
- Content Security Policy & rate limiting for public endpoints.
- Sanitize file uploads; store media in private buckets when needed.
- Use CSP, HSTS headers (Nginx config).
- Limit admin access (IP allowlist) and enable 2FA for admin accounts.

Nginx security headers example:

```
add_header X-Frame-Options \"SAMEORIGIN\"; add_header X-Content-Type-

Options \"nosniff\";

add_header  Referrer-Policy  \"no-referrer-when-downgrade\";
add_header Strict-Transport-Security \"max-age=31536000; includeSubDomains;
```

```
preload\";
```

Backups & Disaster Recovery

Postgres on VPS:

```
# Dump
pg_dump -U ninja_user -h localhost ninja_db > backup_$(date +%F).sql

# Restore
psql -U ninja_user -h localhost -d ninja_db < backup.sql
```

For large DBs use pg_dump -Fc (custom format) and pg_restore.

For managed DBs, use automated daily backups and snapshots.

Store backups offsite (S3) and test restores regularly.

11 Logging, Monitoring & Alerts

- Logs: Use structured logging (JSON) and ship to ELK stack or
- Metrics: Export metrics (Prometheus) and visualize in Grafana.
- Uptime: UptimeRobot or cloud provider health checks.
- Alerts: CPU, errors, high latency, DB connection issues should

```
LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "handlers": {
        "console": {"class": "logging.StreamHandler"},
    },
    "root": {"handlers": ["console"], "level": "INFO"},
}
```

Consider Sentry for error tracking (sentry-sdk).

12 Performance tuning & common pitfalls Pitfalls

- N+1 DB queries: use select_related / prefetch_related.
- Long-running requests: move to background tasks.
- Too many DB connections: use pgbouncer.
- Large static/media files served by Django: use Nginx or S3.
- Unmanaged secrets: store in Vault or cloud secret manager.

Tuning

- Increase Gunicorn workers depending on CPU: workers ≈ 2 × cores + 1 (but test).
- Use async workers (uvicorn) for async endpoints / websockets if
- Cache (Redis) for expensive queries.

13 Example Nginx + Gunicorn production setup (full)

Nginx site (HTTPS handled by certbot):

```
server {
    listen 80;
    server_name example.com;

    location /.well-known/acme-challenge/ {
        root /var/www/certbot;
    }

    location /static/ {
        alias /srv/django_ninja_app/staticfiles/;
    }

    location /media/ {
        alias /srv/django_ninja_app/mediafiles/;
    }

    location / {
        proxy_pass http://unix:/run/gunicorn.sock;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
```

```
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

Gunicorn socket path and systemd file were shown earlier.

14 CI/CD (recommended)

Use GitHub Actions / GitLab CI or cloud CI to:

- Run tests
- Lint code
- Build Docker image
- Push to registry
- Trigger deployment (SSH deploy, Render/GCP deploy, or update

Example GitHub Actions (sketch):

```
name: CI
on: [push]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-python@v4
        with: {python-version: '3.11'}
      - run: pip install -r requirements.txt
      - run: pytest
```

Add a deployment job after tests pass.

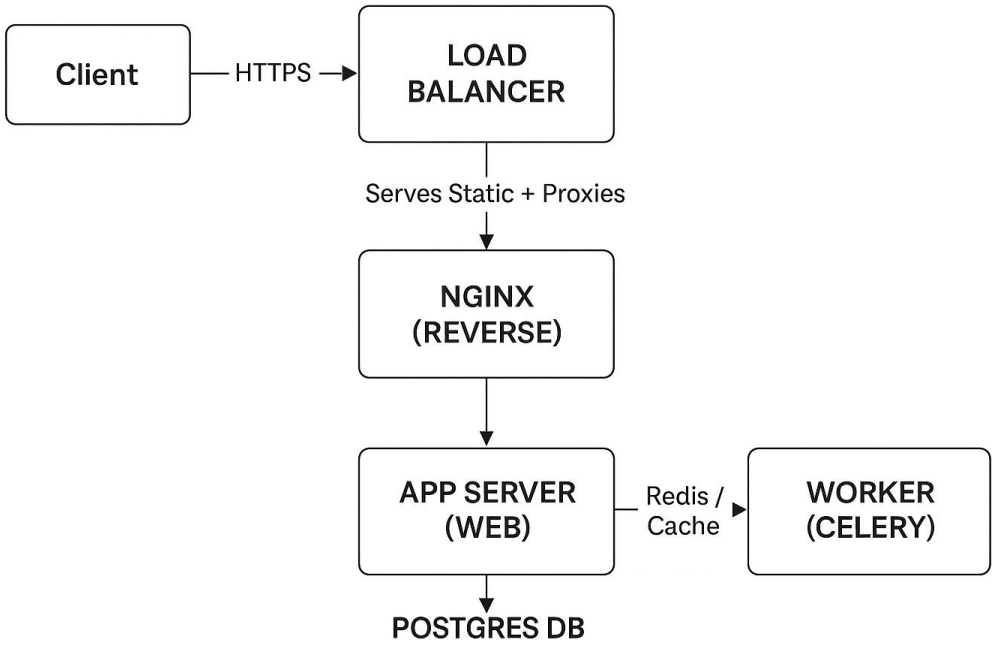
15 Health checks & readiness in Kubernetes / Docker

- Kubernetes uses readiness and liveness probes from earlier.
- For Docker Swarm or simple LB, set an HTTP health check hitting /health.

16 Example: Full deploy sequence (Docker Compose on VPS)

1. SSH to server and install Docker & Docker Compose.
2. Clone repo to /srv/app.
3. Create .env.production with env vars.
4. docker-compose up -d \--build
5. docker-compose exec web python manage.py migrate
6. docker-compose exec web python manage.py collectstatic \--noinput
7. Configure Nginx to proxy to the Docker Nginx or use the Docker Nginx
8. Add certbot for SSL or configure cloud TLS.
9. Run docker-compose ps to confirm services healthy.

17 Small Architecture diagram



20. Troubleshooting quick tips

- 502 Bad Gateway → Check Gunicorn socket path and that Gunicorn is
- 500 Internal Server Error → Check Gunicorn logs & Django logs,
- Static files 404 → Confirm collectstatic ran and Nginx static alias
- DB connection refused → Verify DB host, user, password, and that
- High latency → Profile DB queries for N+1 or missing indexes.

21. Checklist before hitting "deploy"

- DEBUG=False and YES in env variables
- Secrets in environment, not repo
- Database accessible and migrated
- Static files collected & served via Nginx/CDN/S3

- Gunicorn workers tuned and managed (systemd or containers)
- Nginx configured and TLS set up
- Celery workers and Redis if background tasks used
- Monitoring & logging configured
- Backups tested for DB
- Load test in staging (optional)
- Health endpoints for readiness/liveness

Exercises (hands-on)

21. Deploy with Docker Compose: Use the provided docker-compose.yml
22. Deploy to a small VPS: Follow the Gunicorn + Nginx + Certbot
23. Add Celery: Implement a background task (welcome email) and
24. Scale: Spin up two web replicas behind Nginx and confirm
25. Backup & Restore: Create a Postgres dump and restore it to a new

Quick reference commands

```
# Migrate and static
python manage.py migrate
python manage.py collectstatic --noinput

# Gunicorn run (local)
gunicorn project.wsgi:application -w 3 -b 0.0.0.0:8000

# Docker build & up
docker-compose up -d --build
```

```
# Create DB dump
pg_dump -U user -h host dbname > backup.sql

# Restore
psql -U user -h host dbname < backup.sql
```

Final words --- keep it safe, simple, proven

Deploying is the point where code meets reality. Start small, automate everything you can (CI/CD, infra as code), and monitor closely. Use managed services where they save time (managed Postgres, Load Balancers, Certs). Always test restores and rehearse deployments.

If you want, I can:

- Generate a ready-to-run Docker Compose repo scaffold for your project , or
- Produce Nginx + systemd config files customized to your project
- Create a Kubernetes manifest set and a GitHub Actions pipeline that builds/pushes Docker images and deploy.

CHAPTER 23

Unit Testing Your Django Ninja APIs

The professional habit that separates confident developers from nervous ones

Here's something nobody tells you until it's too late: the scariest moment in development is deploying code you're not sure about. Testing takes that fear away. It's not about following rules — it's about the freedom to change things without lying awake wondering what you broke.

This chapter shows you how to write fast, readable tests for every part of your Django Ninja API. By the end, you'll have a test suite that checks your endpoints, your authentication, and your business logic automatically — every single time you push code.

Why Testing Matters

Without Tests	With Tests
Bugs hide until a user finds them	You find bugs before users do
You avoid refactoring — too risky	Refactor freely — tests catch regressions
Manual Swagger testing every change	One command checks everything
Hard to onboard new developers	Tests document expected behaviour
Deploy with crossed fingers	Deploy with confidence

Step 1 — Install the Testing Tools

The good news: Django already ships with a test runner. We're just adding pytest on top for a nicer experience and better output.

requirements.txt (testing section):

```
pytest==8.1.1  # the test runner — easier to use than Django's default
pytest-django==4.8.0  # bridges pytest with Django's database and settings
```

pytest.ini (project root):

```
[pytest]
DJANGO_SETTINGS_MODULE = your_project.settings
python_files = tests.py test_*.py *_test.py

python_classes = Test*
python_functions = test_*
```

Step 2 — Your First Test

Let's start with the simplest possible test — does your /hello endpoint return 200 OK and the right message? This takes two minutes to write and will pay you back forever.

```
from django.test import TestCase, Client

class TestHelloEndpoint(TestCase):

    def setUp(self):
        self.client = Client()  # Django's built-in HTTP test client

    def test_hello_returns_200(self):
        response = self.client.get('/api/hello')
        self.assertEqual(response.status_code, 200)  # must be 200 OK

    def test_hello_message_content(self):
        response = self.client.get('/api/hello')
        data = response.json()
        self.assertIn('message', data)  # response has a 'message' key
```

Step 3 — Testing CRUD Endpoints

This is where testing earns its keep. You test each operation — create, read, update, delete — and you test both the happy path (valid data) and the sad path (bad data, missing records).

```
from django.test import TestCase, Client
import json
class TestProductAPI(TestCase):
    def setUp(self):
        self.client = Client()
        self.product = self._create_product('Laptop', 899.99, 10)

    def _create_product(self, name, price, stock):
        """Helper so we don't repeat POST logic in every test."""
        resp = self.client.post(
            '/api/products',
            data=json.dumps({'name': name, 'price': price, 'stock':
stock}),
            content_type='application/json'
        )
        return resp.json()

    # CREATE ——————————————————————
    def test_create_product_returns_id(self):
        self.assertIn('id', self.product)  # DB assigned an id
        self.assertEqual(self.product['name'], 'Laptop')

    def test_create_product_rejects_bad_price(self):
```

```
        resp = self.client.post(
            '/api/products',
            data=json.dumps({'name': 'X', 'price': 'not-a-number',
'stock': 1}),
            content_type='application/json'
        )
        self.assertEqual(resp.status_code, 422)  # Pydantic rejects it


    # READ ──────────────────────────────────────
    def test_get_product_by_id(self):
        pid = self.product['id']
        resp = self.client.get(f'/api/products/{pid}')
        self.assertEqual(resp.status_code, 200)
        self.assertEqual(resp.json()['name'], 'Laptop')


    def test_get_nonexistent_product_returns_404(self):
        resp = self.client.get('/api/products/99999')
        self.assertEqual(resp.status_code, 404)


    # DELETE ────────────────────────────
    def test_delete_product(self):
        pid = self.product['id']
        del_resp = self.client.delete(f'/api/products/{pid}')
```

```
        self.assertEqual(del_resp.status_code, 200)

        # Confirm it's gone
        get_resp = self.client.get(f'/api/products/{pid}')
        self.assertEqual(get_resp.status_code, 404)
```

Step 4 — Testing Authentication

Authentication bugs are the most dangerous kind — they either lock out your users or let the wrong people in. Test both scenarios every time.

```
from django.test import TestCase, Client
from django.contrib.auth.models import User
import json
class TestAuthentication(TestCase):
    def setUp(self):
        self.client = Client()
        User.objects.create_user(username='alice', password='pass123')

    def test_login_with_correct_credentials(self):
        resp = self.client.post(
            '/api/login',
            data=json.dumps({'username': 'alice', 'password': 'pass123'}),
            content_type='application/json'
        )
        self.assertEqual(resp.status_code, 200)
        self.assertIn('access_token', resp.json())
```

```
def test_login_with_wrong_password(self):
    resp = self.client.post(
        '/api/login',
        data=json.dumps({'username': 'alice', 'password': 'wrong'}),
        content_type='application/json'
    )
    self.assertEqual(resp.status_code, 401)  # Unauthorized

def test_protected_route_without_token(self):
    resp = self.client.get('/api/profile')  # no token
    self.assertEqual(resp.status_code, 401)

def test_protected_route_with_valid_token(self):
    # 1. Get a token
    login = self.client.post(
        '/api/login',
        data=json.dumps({'username': 'alice', 'password': 'pass123'}),
        content_type='application/json'
    )
    token = login.json()['access_token']

    # 2. Use it on a protected endpoint
    resp = self.client.get(
        '/api/profile',
        HTTP_AUTHORIZATION=f'Bearer {token}'
    )
```

```
        self.assertEqual(resp.status_code, 200)
        self.assertEqual(resp.json()['username'], 'alice')
```

Step 5 — Testing Business Logic (Banking Example)

The real value of tests shows up when you're testing things like 'the balance should increase by exactly the deposit amount' or 'you can't withdraw more than your balance'. These are the rules that matter most — and they're exactly the kind of thing that's easy to break accidentally.

```
from django.test import TestCase, Client
from django.contrib.auth.models import User
from api.accounts.models import Account
import json
class TestBankingAPI(TestCase):
    def setUp(self):
        self.client = Client()

        self.user = User.objects.create_user(
            username='bob',
            password='pass123'
        )
        self.account = Account.objects.create(
            owner=self.user,
            account_number='ACC001',
            balance=500.00
        )
```

```
    # Log in and store the token
    resp = self.client.post(
        '/api/login',
        data=json.dumps({'username': 'bob', 'password': 'pass123'}),
        content_type='application/json'
    )

    self.headers = {
        'HTTP_AUTHORIZATION': f"Bearer {resp.json()['access_token']}"
    }

def test_deposit_increases_balance(self):
    self.client.post(
        '/api/transactions/deposit',
        data=json.dumps({'amount': 200.00}),
        content_type='application/json',
        **self.headers
    )

    self.account.refresh_from_db()  # reload from DB
    self.assertEqual(float(self.account.balance), 700.00)  # 500 + 200

def test_cannot_overdraw_account(self):
    resp = self.client.post(
        '/api/transactions/withdraw',
        data=json.dumps({'amount': 9999.00}),  # more than balance
```

```
            content_type='application/json',
            **self.headers
        )
        self.assertEqual(resp.status_code, 400)  # should be rejected
        self.account.refresh_from_db()
        self.assertEqual(float(self.account.balance), 500.00)  # unchanged
```

Step 6 — Running Your Tests

Terminal — common pytest commands

```
# Run the full suite
pytest

# Run with verbose output (see every test name)
pytest -v

# Run only one file
pytest tests/test_auth.py

# Run only one specific test
pytest tests/test_auth.py::TestAuthentication::test_login_with_correct_credentials

# Run with coverage report
pytest --cov=api --cov-report=term-missing
```

Sample output — what passing tests look like

```
tests/test_hello.py::TestHelloEndpoint::test_hello_returns_200 PASSED
tests/test_products.py::TestProductAPI::test_create_product_returns_id PASSED
tests/test_products.py::TestProductAPI::test_get_nonexistent_product_returns_404
PASSED
tests/test_auth.py::TestAuthentication::test_login_with_correct_credentials PASSED
tests/test_auth.py::TestAuthentication::test_protected_route_with_valid_token
PASSED
```

Step 7 — Automatic Testing with GitHub Actions

The final step is making sure tests run automatically every time you push code to GitHub.

This way, nobody can accidentally merge broken code.

```
name: Run Tests

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - uses: actions/setup-python@v5
        with:
          python-version: '3.11'

      - run: pip install -r requirements.txt

      - run: pytest -v
# Every push to GitHub now runs your test suite automatically.
# A failing test blocks the pull request from merging.
```

Testing Best Practices

Write one test per scenario. Test the happy path (valid input) AND the sad path (bad data, wrong password, missing record). Use setUp() to avoid repeating code. Run tests before every deploy — not after something breaks.

Exercises

1. Write tests for all five CRUD endpoints of your Product API.
2. Add a test that confirms a viewer role cannot access /admin/users.
3. Set up GitHub Actions to run your tests on every push.
4. Run pytest --cov and see what percentage of your code is covered.

CHAPTER 24

Interview Questions and Best Practices

Stand Out as a Django Ninja and API Engineer

Overview

This chapter includes:

- Core interview questions (with sample answers)
- Conceptual and scenario-based questions
- Best practices for Django Ninja + API development
- Live coding challenges
- Soft skill & collaboration questions
- Tips to stand out as a candidate

21.1 : Django Ninja-Specific Questions

These questions focus on the framework and its usage in real-world REST API development.

Q1. What is Django Ninja and how is it different from Django REST Framework (DRF)?

A:
Django Ninja is a modern, fast web framework for building APIs with Django. Key differences from DRF include:

Django Ninja	Django REST Framework
Built on Pydantic for validation	Custom serializers
Faster performance (based on FastAPI principles)	Slower comparatively

Auto docs via OpenAPI + Swagger	Requires config for docs
Simpler routing and schema handling	More flexible but verbose

Q2. What are the key building blocks in a Django Ninja app?

- Router for grouping endpoints
- Schema for validating input and formatting output
- Pydantic models under the hood
- Response type to specify output
- HTTP methods (get, post, put, etc.)

Q3. How do you secure routes in Django Ninja?

A:
By using security backends like django_auth or custom strategies:

```
from ninja import Router
from ninja.security import django_auth
router = Router(auth=django_auth)
@router.get('/secure-data')
def secure_view(request):
    return {"message": "This is secured"}
```

21.2 High-Level Concepts and System Design Q4. How do you design REST APIs for scalability?

- Use pagination
- Optimize database queries (prefetch/select_related)
- Cache responses where appropriate
- Use async for high IO tasks (if necessary)
- Separate read/write services in large systems (CQRS)

- Document API with Swagger/OpenAPI
- Use load balancing and horizontal scaling

Q5. How would you structure a Django Ninja project for a large system?

Answer may include:

- Domain-driven design structure
- Broken into modules (/api/users, /api/payments, etc.)
- Use services/helpers for business logic outside views
- Use Django signals and middleware smartly
- Use settings for environment-based configs

21.3 : Behavioral / Soft Skill Questions

Q: Can you describe a challenge you faced while building a Django API?

Example Answer:
"I was working on a patient management API where some endpoints were very slow. I profiled the API and noticed N+1 query issues, so I used select_related and prefetch_related to optimize it, reducing the
response time by over 70%."

Section 21.4: Live Coding Questions

Turn these into a practice exercise before real interviews.

**Task: Create a simple Django Ninja view that filters products based on
category and price range.**

Expected Solution:

```
@router.get("/products")
def filter_products(request, category: str = None, min_price: float = None, max_price:
float = None):
    qs = Product.objects.all()
    if category:
        qs = qs.filter(category__icontains=category)
```

```
    if min_price:
        qs = qs.filter(price__gte=min_price)
    if max_price:
        qs = qs.filter(price__lte=max_price)
    return qs
```

21.5: Best Practices

Area	Best Practice
Schemas	Use explicit schemas for input/output
Error Handling	Use global exception handlers
Logging	Log all errors, use correlation IDs for debugging
Security	Use HTTPS, validate data, sanitize inputs
Performance	Avoid N+1, cache when possible
Documentation	Make your /api/docs intuitive and always up to date

21.6
21.8
21.9
21.10

21.6: Bonus -- What Hiring Managers Look For

- Clean code with clarity and minimalism
- Understanding of design patterns
- Ability to abstract logic and reuse code
- Communication ability
- Demonstrated side projects (like your hospital API)
- Eagerness to learn and collaborate

Final Tip

Always have a mini portfolio ready. A well-structured **GitHub repository** with a Django Ninja project (e.g., the hospital app) gives you an edge. Include a README.md with installation steps, diagrams, and sample API calls.

Acknowledgements

I would like to express my heartfelt appreciation to Vitaliy Kucheryaviy, the creator of Django Ninja. His vision, dedication, and contribution to the Python ecosystem made this book — and the clarity it brings to developers — truly possible.

My deepest gratitude goes to my mother, Happiness Emechebe, whose unwavering support, encouragement, and belief in my journey have been a constant source of strength.

I also wish to acknowledge all my mentees, whose continuous requests and anticipation for this book kept me motivated throughout the writing process. Your passion for learning and your trust in my guidance inspired me to bring this work to life.

Thank you all for being part of this journey.

References

Brady, A., Greenfeld, D., & Roy, D. (2017). Two Scoops of Django 1.11: Best Practices for the Django Web Framework. Two Scoops Press.

Django Software Foundation. (n.d.). Django documentation. https://docs.djangoproject.com/en/stable/

Docker Inc. (n.d.). Docker documentation. https://docs.docker.com/

Jones, B. (n.d.). Real Python Tutorials. https://realpython.com/

Linux Foundation. (n.d.). NGINX official documentation. https://nginx.org/en/docs/

Matsiev, E. (n.d.). Django Ninja documentation. https://django-ninja.dev/

OWASP Foundation. (2019). OWASP Top Ten API Security. https://owasp.org/www-project-api-security/

PostgreSQL Global Development Group. (n.d.). PostgreSQL documentation. https://www.postgresql.org/docs/

Redis Ltd. (n.d.). Redis documentation. https://redis.io/docs/

Uvicorn Authors. (n.d.). Uvicorn — ASGI server documentation. https://www.uvicorn.org/ Van Rossum, G., & Drake, F. L. (2009). Python programming language. Python Software Foundation. https://python.org

About the Author

Sylvester Benjamin is a Software Engineer, Cybersecurity Consultant, and Founder of Cyber Oracle and SentinelMesh Ltd.

He leads Cyber Oracle, where he works with organizations to design and deliver secure, scalable software systems, cybersecurity solutions, and resilient network infrastructure. His work is grounded in solving real business problems — building systems that don't just function, but hold up under pressure.

He is also the founder of SentinelMeshAI, an enterprise AI operating system designed to help businesses run smarter, automate intelligently, and make better decisions at scale. His focus here is on simplifying how organizations interact with AI — turning complexity into usable, practical systems that drive real outcomes.

Sylvester's approach sits at the intersection of engineering, security, and execution. He's not just focused on writing code, but on building systems, leading teams, and turning ideas into working products that people can rely on.

Beyond his work, he is committed to mentoring and supporting the next generation of developers and tech professionals, especially in areas like APIs, cloud systems, and building with a security-first mindset. Through his academy and community efforts, he shares what he's learned to help others grow faster and build better.

He believes that real innovation isn't just about moving fast — it's about building things that last. This book reflects that mindset, offering a practical bridge between rapid development and disciplined, secure engineering.