1. Django and React Preparation

To set up a Django + React project, run Django on port 8000 and React on 3000. React's `package.json` should include a proxy to direct API calls to Django. For production, compile React with `npm run build`, then serve its static files through Django using tools like WhiteNoise or collectstatic. Use React Router for navigation; Django must return `index.html` for unknown routes to enable client-side routing.

2. Filtering and Pagination with Django and React

Use Django's `ModelViewSet` and configure pagination with `PageNumberPagination` or `LimitOffsetPagination`. Filtering is added via `DjangoFilterBackend` and fields are exposed using `filterset_fields`. React handles filter inputs and pagination links, fetching data via Axios with query params like `?page=2&category=books`. The backend ensures clean data slicing, reducing payload and improving performance.

3. Forms with React and Django

Django REST framework uses `ModelSerializer` to create serializers that map model fields into JSON-ready formats. On the frontend, React uses controlled forms with state, while Axios handles form submission. Django performs backend validation (e.g., unique, min/max), and returns errors in JSON, which React catches and renders as user-friendly messages. This split secures logic and ensures consistent validation.

4. Refactoring React with Hooks and Higher-Order Components

Older React code uses Higher-Order Components (HOCs) to share logic. Modern React favors hooks like `useState`, `useEffect`, and `useCallback` to encapsulate stateful logic inside functional components. You can refactor HOCs into custom hooks, reducing boilerplate and improving readability. For example, timers or event listeners can be managed using `useEffect` with cleanup callbacks.

5. Testing React and Django

Testing in React is done using `@testing-library/react` and `jest`. Focus on rendering, props, user interactions, and API integration using mocking libraries like `msw`. For Django, use `APITestCase` or `APIClient` from `rest_framework.test`. Validate CRUD operations, auth permissions, and field-level errors. Always test both frontend form behavior and backend API reliability for production-ready code.

6. Django and Angular Preparation

Angular and Django are run separately during development. After building Angular (`ng build`), copy the output from `dist/` into Django's static files or configure static serving in production. Django APIs remain the backend source. Angular interacts with APIs via `HttpClientModule`, which must be configured to handle Django's CSRF or token auth properly. Make sure CORS headers are set up correctly.

7. Forms with Angular and Django

Angular uses Reactive Forms with `FormGroup` and `FormControl` to manage and validate inputs. Submit actions use Angular's `HttpClient` to POST data to Django REST endpoints. On failure, Django returns validation errors, which Angular then maps to specific form fields. This ensures a smooth user experience and maintains secure server-side validation.

8. Front-End Design and Layout with Angular

Use Angular Material to design UI components like data tables, dialogs, and forms. Components like `mat-form-field`, `mat-table`, and `mat-dialog` improve UX and align with Material Design principles. Angular templates allow conditional rendering and real-time feedback. This helps present data dynamically and efficiently within the component structure.

9. Authentication with Django and Angular

Use Django OAuth Toolkit to implement OAuth2 with scopes. Angular must manage token storage securely (e.g., in memory or session storage) and attach it in request headers using `HttpInterceptor`. Login and permission checks are performed with Django's token validation, and user info is returned for role-based rendering on the frontend. This ensures secure, scalable user auth.

10. Filtering and Pagination with Django and Angular

Filtering is done in Django using query parameters and FilterSet classes. Angular passes those parameters via `HttpClient`. Pagination can be handled with Angular Material Paginator or custom logic. Always map backend pagination fields like `next`, `previous`, and `results` to the frontend UI for seamless UX.

11. Testing Angular

Angular testing uses Jasmine and Karma. Unit tests check isolated component logic, services, and pipes. Use mocks for HttpClient and simulate user actions to verify form and data handling. End-to-end tests use Protractor or Cypress to simulate user journeys. Key focus: form inputs, API calls, DOM rendering, and navigation.

12. Testing Django

Django tests use `TestCase`, `APITestCase`, and `APIClient`. Test cases simulate requests to endpoints, validate responses, and assert permissions or data integrity. Common tests include model field validation, permissions, serializers, and views. Use fixtures or factories to populate test data. Use `setUp()` for reusable initialization.

✅ Use this fullstack guide to master Django + REST + React/Angular. Let me know if you want flashcards, quiz questions, deployment steps, or debugging patterns!

Serailization and eveyrhtin

Django + Django REST + React + Angular Fullstack Mastery Sheet

---

1. Serializing, Listing, Filtering, and Paginating Models

Django REST Framework uses **serializers** to convert model instances to JSON. `ModelSerializer` automatically generates fields based on a model. `ListAPIView` displays multiple records and can be connected to URLs using routers or manually with path functions. Use `filter_backends` to support URL-based filtering (e.g., `?category=books`) and `pagination_class` to chunk data into pages. Full-text search can be added using `SearchFilter` and fields defined in `search_fields`. You can also enable read replicas by adjusting the DB router logic to support scalable read-heavy workloads.

---

2. Create, Retrieve, Update, and Delete (CRUD) Operations for Models

Django REST offers generics like `CreateAPIView`, `UpdateAPIView`, `DestroyAPIView`, and `RetrieveAPIView` to handle CRUD actions for models. These classes simplify logic by encapsulating POST, GET, PUT, PATCH, and DELETE methods. Connect views to URLs using routers (`DefaultRouter.register()`) or explicitly with `path()`. For each view, define a corresponding serializer and optionally override methods like `perform_create()` to add custom logic (e.g., adding request user).

---

3. Managing Serializer Fields, Relations, and Validation

Serializers can include only selected fields using the `fields` attribute in `Meta`. To show relationships (e.g., FK or M2M), use `RelatedField`, `SlugRelatedField`, or nested serializers. Custom field validation can be implemented with `validate_<fieldname>()`, while global validation is done using `validate()` method. Handle complex types like JSON, lists, dicts using `ListField`, `DictField`, and `JSONField`. `ImageField` and `FileField` support file uploads with proper media configuration.

---

4. Testing API Views

Django's `rest_framework.test` provides `APITestCase` and `APIClient` for endpoint testing. Each test checks the behavior of endpoints (e.g., create, list, update) under various scenarios like valid input, invalid data, and permission rules. Test file/image upload with mock files, and validate expected response formats (status codes, JSON structure). Use `setUp()` for shared test data. Assertions include `assertEqual`, `assertContains`, and `assertIn`.

MAKING PAID WEBSITE

Here's a **comprehensive Django Developer Cheatsheet**, organized **heading-wise**, covering everything from core concepts to advanced deployment. This will serve as your go-to guide while taking LinkedIn Learning courses like *"Advance Your Skills as a Django Developer"* and building projects like paid membership sites.

🧠 Django Fundamentals

✅ What is Django?

- A high-level Python web framework that promotes rapid development and clean design using the **MVT (Model-View-Template)** architecture.

🏗️ Project Structure

- `manage.py`: Command-line utility
- `settings.py`: Configuration
- `urls.py`: URL routing
- `models.py`: Database schema
- `views.py`: Request handling
- `templates/`: HTML files
- `static/`: CSS, JS, images

⚙️ Django Commands

💡 Project & App

Shell
```
django-admin startproject projectname
python manage.py startapp appname
python manage.py runserver
```

🛠️ Migrations

Shell
```
python manage.py makemigrations
python manage.py migrate
python manage.py sqlmigrate appname 0001
```

👥 Superuser

Shell
```
python manage.py createsuperuser
```

🧩 Models

✍️ Defining a Model

Python
```
class Product(models.Model):
    name = models.CharField(max_length=255)
    price = models.DecimalField(max_digits=10, decimal_places=2)
```

🔁 Field Types

- `CharField`, `TextField`, `IntegerField`, `DateTimeField`, `BooleanField`
- `ForeignKey`, `OneToOneField`, `ManyToManyField`

🔍 Model Meta & Methods

Python
```python
class Meta:
    ordering = ['-created_at']
def __str__(self):
    return self.name
```

📚 ORM (Object-Relational Mapper)

📌 Queries

Python
```python
Product.objects.all()
Product.objects.filter(price__gte=100)
Product.objects.get(id=1)
Product.objects.create(name='X', price=99.99)
```

📦 Views

🧠 Function-Based View (FBV)

Python
```python
def homepage(request):
    return render(request, 'home.html')
```

💡 Class-Based View (CBV)

Python
```python
from django.views import View
class HomeView(View):
    def get(self, request):
        return render(request, 'home.html')
```

🔗 URL Routing

🔄 Include in `urls.py`

Python
```python
from django.urls import path
from . import views
urlpatterns = [
    path('', views.homepage, name='home'),
]
```

🎨 Templates

📋 Basics

HTML
```html
<h1>{{ product.name }}</h1>
{% for product in products %}
    <p>{{ product.price }}</p>
{% endfor %}
```

🔄 Template Inheritance

HTML
```html
{% extends "base.html" %}
{% block content %}{% endblock %}
```

📥 Forms

🔧 Basic Form

```python
from django import forms
class ProductForm(forms.Form):
    name = forms.CharField()
    price = forms.DecimalField()
```

🛡️ CSRF Protection

```html
<form method="POST">
  {% csrf_token %}
  {{ form.as_p }}
</form>
```

🧾 Admin Interface

⚙️ Register Models

```python
from django.contrib import admin
from .models import Product
admin.site.register(Product)
```

🛒 Paid Membership / Stripe Integration

🏁 Flow

1. Setup Stripe account & API keys
2. Create product and pricing plans (monthly, yearly)
3. Use Stripe Checkout Session (via Stripe API)
4. Handle webhooks for success/cancel
5. Store subscription status in UserProfile

🔐 Example

Python
```python
checkout_session = stripe.checkout.Session.create(
    payment_method_types=['card'],
    line_items=[{...}],
    mode='subscription',
    success_url=YOUR_DOMAIN + '/success/',
    cancel_url=YOUR_DOMAIN + '/cancel/',
)
```

👥 Authentication & Authorization

✅ User Auth

Python
```python
from django.contrib.auth import authenticate, login, logout
```

🔐 Decorators

Python
```python
@login_required
@permission_required('app.add_product')
```

🧪 Testing

🔍 Unit Test Example

Python
```python
from django.test import TestCase
class ProductTestCase(TestCase):
    def test_product_creation(self):
        product = Product.objects.create(name="Test", price=10)
        self.assertEqual(product.name, "Test")
```

📬 Email / Notifications

```python
from django.core.mail import send_mail
send_mail("Subject", "Message", "from@example.com",
["to@example.com"])
```

🚀 Deployment

🛠️ Production Settings

- `DEBUG = False`
- `ALLOWED_HOSTS = ['yourdomain.com']`
- Use `gunicorn` or `uwsgi` with `nginx`
- Use PostgreSQL for DB

☁️ Hosting Platforms

- Heroku (PaaS)
- AWS EC2 (IaaS)
- DigitalOcean

🔐 Security Best Practices

- Keep `SECRET_KEY` secret
- Disable `DEBUG` in prod
- Use HTTPS
- Validate all user input
- Use Django’s built-in protection: CSRF, XSS, SQLi safe ORM

🧰 Tools & Packages

- **Stripe**: Payments
- **Celery + Redis**: Background tasks
- **Django REST Framework**: APIs

- **WhiteNoise**: Serve static files in production
- **dotenv / environs**: Env vars management

Would you like me to turn this into a printable PDF or Notion page for easy reference while doing the course?

SECURING THE DJANGO PATH

Great! You've now reached the **"Securing Django Applications"** course on LinkedIn Learning. Here's a **Django Security Cheatsheet**, neatly organized by headings to help you **master all security topics** covered in the course — from permissions and throttling to encryption, 2FA, CSRF, and more.

🛡️ Django Security Cheatsheet

1. 🔐 Permissions, Access Controls, and Audit Logs

✅ Object-Level Permissions

- Use `django-guardian` or custom `has_object_permission` logic in DRF.

```python
def has_object_permission(self, request, view, obj):
    return obj.owner == request.user
```

🧑‍🤝‍🧑 Group-Based Access

- Use Django's built-in `Group` and `Permission` models.

```python
from django.contrib.auth.models import Group, Permission
group = Group.objects.create(name='Editors')
group.permissions.add(Permission.objects.get(codename='change_post'))
```

🧾 Activity Logs

- Custom middleware or model to log:
    - Who accessed what

    ○ When and why

```python
class ActivityLog(models.Model):
    user = models.ForeignKey(User)
    action = models.CharField()
    timestamp = models.DateTimeField(auto_now_add=True)
```

🚮 Secure Delete

- Never hard delete sensitive data. Use soft-deletion flags like:

```python
class MyModel(models.Model):
    is_deleted = models.BooleanField(default=False)
```

2. 🧃 Throttling and Flood Protection

📶 ApacheBench Testing

```shell
ab -n 1000 -c 10 http://localhost:8000/api/test/
```

🔁 Idempotent Operations

- Use unique transaction IDs or `once` flags to ensure operations happen once.

```python
if not obj.processed:
    obj.processed = True
    obj.save()
```

⏳ Background Queues

- Use `Celery` with `Redis` or `RQ` for deferring tasks.

3. 🛡️ Data Protection & Privacy

🔐 Per-field Encryption

- Use `cryptography.fernet` or `django-encrypted-model-fields`.

```python
from cryptography.fernet import Fernet
key = Fernet.generate_key()
cipher = Fernet(key)
encrypted = cipher.encrypt(b"secret")
```

🔒 Zero-Knowledge Encryption

- Encrypt on the frontend, decrypt only client-side.
- Server stores ciphertext only, not the decryption key.

🧪 Unit Test Encrypted Fields

```python
self.assertNotIn("plaintext", response.content)
```

📦 GDPR Data Packaging

- Provide full export of user data (e.g. JSON zip file) for compliance

4. 📲 Two-Factor Authentication (2FA)

📤 SMS Code via Twilio

- Send OTP using Twilio API.

```python
client.messages.create(
    to=user.phone,
    from_="YourApp",
    body=f"Your 2FA code is {otp}"
)
```

✅ Confirming OTP

- Match code and set a `2fa_verified` flag before allowing access.

🔐 Validate Before Actions

```python
@user_passes_test(lambda u: u.profile.two_fa_verified)
def sensitive_action(request):
    ...
```

🧪 Unit Testing 2FA

- Mock OTP verification and assert access only after validation.

5. 🛡️ CSRF Protection

🔒 CSRF in Forms

```html
<form method="POST">
  {% csrf_token %}
</form>
```

📋 CSRF in APIs (If using cookies)

- Use `X-CSRFToken` header or `csrf_exempt` (with caution).

```python
from django.views.decorators.csrf import csrf_exempt
@csrf_exempt
def my_api(request):
    ...
```

🧪 Unit Testing CSRF

```python
self.client.post('/endpoint/', {}, follow=True)
```

🧰 Additional Django Security Tips

Category	Recommendation
Settings.py	Never expose `SECRET_KEY`, use `.env`
HTTPS	Use `SECURE_SSL_REDIRECT = True`
Headers	Set `X-Frame-Options`, `Content-Security-Policy`
Rate Limiting	Use `drf-extensions` or middleware
Third-Party Auth	Prefer social auth or JWT with expiry
Error Handling	Avoid leaking sensitive info in stack traces

Would you like this turned into a **Notion doc**, **PDF**, or **Django security checklist spreadsheet** for review as you implement it in your project?