10-Month Backend Developer Roadmap

A practical learning plan for a junior backend developer working with Django, APIs, data platforms, async jobs, and observability.

Priority levels

Core skills you must become strong in

- Python
- Django
- Django Ninja / DRF
- REST API
- PostgreSQL
- Docker
- Linux
- Git
- Redis
- Celery

Production/backend architecture skills

- Message brokers: Kafka, RabbitMQ
- ELK
- OpenTelemetry
- gRPC
- GraphQL
- FastAPI

Domain/platform-specific skills

- DataHub
- Kafka deeper internals
- OpenTelemetry deeper internals

Honest criticism: do not try to learn Django, DRF, Django Ninja, FastAPI, GraphQL, gRPC, Kafka, Celery, ELK, OpenTelemetry, and DataHub all at the same depth at the same time. You will feel busy but not become strong.

```
Python -> Django -> REST/API design -> PostgreSQL -> Docker/Linux/Git -> Celery/Redis ->
    Kafka -> Observability -> advanced APIs/architecture
```

10-month roadmap

This plan assumes about 10-15 hours of study and practice per week. With 20+ hours per week, you can move faster.

Month 1 - Python for backend, not beginner Python

Goal: stop writing script-style Python and start writing production backend Python.

What to learn

- functions
- args / kwargs
- decorators
- context managers
- typing
- dataclasses
- enums
- exceptions
- generators and iterators
- list/dict/set comprehensions
- modules/packages
- virtualenv
- pip
- pyproject.toml
- requirements.txt
- logging basics

Special focus

```
typing.Optional
typing.Any
dict[str, Any]
list[Model]
Callable
Protocol
Literal
```

Exception handling

```
try:
    ...
except SpecificException as e:
    ...
except Exception as e:
    ...
finally:
    ...
```

Bad:

```
try:
    do_everything()
except Exception:
    pass
```

Good:

```
try:
    payload = json.loads(raw_payload)
except json.JSONDecodeError as e:
    raise DataValidationException("Invalid payload") from e
```

Practice tasks

- Build a JSON validator for batch_id, file_path, and bucket_name.
- Write a mini service layer such as QualityPayloadValidator.validate(...).
- Create a package structure with exceptions.py, validators.py, services.py, and tests/.

Good level after Month 1

You should be able to read backend Python code without getting lost and write clean service/helper classes.

Month 2 - Django foundations deeply

Goal: become comfortable with Django's real backend structure.

Django project structure

```
settings.py
urls.py
apps.py
models.py
views.py
admin.py
migrations/
management/commands/
```

Models

- CharField
- TextField
- IntegerField
- BooleanField
- DateTimeField
- JSONField
- ForeignKey
- OneToOneField
- ManyToManyField
- choices
- db_index

- unique

- null vs blank

Very important: null=True affects the database; blank=True affects forms and validation.

ORM

```
Model.objects.create()
Model.objects.get()
Model.objects.filter()
Model.objects.exclude()
Model.objects.update()
Model.objects.delete()
```

```
select_related()
prefetch_related()
annotate()
aggregate()
Count()
Avg()
Q()
F()
transaction.atomic()
```

Practice project

Build a small Data Quality Management backend with Dataset, QualityRule, QualityExecution, and QualityResult models.

```
class Dataset(models.Model):
    name = models.CharField(max_length=255)
    urn = models.CharField(max_length=500, unique=True)
    created_at = models.DateTimeField(auto_now_add=True)
```

Management commands

```
python manage.py seed_quality_rules
```

Questions you should answer

- What is a migration?

- What happens when I edit a model?

- What is the difference between makemigrations and migrate?

- What causes conflicting migrations?

- How do I fix multiple leaf migration nodes?

- When should I use transaction.atomic?

Month 3 - REST API, Django Ninja, DRF

Goal: design clean APIs, not just make endpoints work.

REST API concepts

- HTTP methods

- GET, POST, PUT, PATCH, DELETE
- status codes
- query params
- path params
- request body and response body
- pagination
- filtering
- sorting
- authentication
- authorization
- idempotency
- error response format

Status codes to know

```
200 OK
201 Created
204 No Content
400 Bad Request
401 Unauthorized
403 Forbidden
404 Not Found
409 Conflict
422 Validation Error
500 Internal Server Error
```

Django Ninja

- Router
- Schema
- ModelSchema
- Query
- Path
- HttpError
- response models
- router tags
- auth

```
GET /datasets
GET /datasets/{id}
POST /datasets
PATCH /datasets/{id}
DELETE /datasets/{id}
GET /datasets/{id}/quality-report
```

DRF

- APIView
- ViewSet
- ModelViewSet
- Serializer
- ModelSerializer
- permissions
- authentication
- pagination
- filter_backends
- routers

Practice task

Build the same CRUD twice: once with Django Ninja and once with DRF. Compare schema style, validation, auth, error handling, pagination, and developer experience.

Advice: for your current project, Django Ninja is enough. Learn DRF because it is marketable, but do not rewrite your project in DRF.

Month 4 - PostgreSQL deeply

Goal: stop treating the database as just where Django saves data.

Learn

- tables
- columns
- primary keys
- foreign keys
- indexes
- unique constraints
- transactions
- joins
- views
- materialized views
- JSONB
- EXPLAIN
- EXPLAIN ANALYZE
- locks
- deadlocks
- isolation levels basics

Django + PostgreSQL

```
indexes = [
    models.Index(fields=["status"]),
    models.Index(fields=["created_at"]),
]
```

Add indexes for fields used in filters, joins, ordering, foreign keys, and unique lookups.

Practice tasks

- Create 100k fake rows and compare a slow query before and after adding an index.
- Practice JSONField queries such as summary__contains={"dataset_urn": dataset_urn}.
- Write create_execution_with_results() inside transaction.atomic().

Good level

- Why is this query slow?
- Should this field be indexed?
- Why did this migration lock the table?
- What happens if two workers update the same row?

Month 5 - Linux, Git, Docker

Goal: become comfortable in real development and deployment environments.

Linux

```
pwd
ls
cd
cat
less
grep
find
chmod
chown
ps
top
htop
kill
curl
wget
ssh
scp
systemctl
journalctl
```

Also learn environment variables, ports, processes, file permissions, logs, and services.

```
grep -RIn "handle_service_exception" apps
find . -name "*.pyc"
ps aux | grep python
curl -i http://localhost:8000/api/health
```

Git

```
git status
git diff
git log
git show
git branch
git checkout
git switch
git restore
git reset
git revert
git stash
git merge
git rebase basics
git cherry-pick
git reflog
```

```
git diff HEAD^1..HEAD
git diff HEAD^2..HEAD
git log --merges
git diff --name-status
git grep
```

Practice conflict resolution intentionally by creating two branches, editing the same file, merging, and resolving.

Docker

- Dockerfile
- image
- container
- volume
- network
- docker-compose
- environment variables
- ports
- build cache
- logs
- exec

```
docker build .
docker compose up
docker compose down
docker compose logs -f
docker compose exec backend bash
docker ps
docker images
docker volume ls
```

Practice project: Django + PostgreSQL + Redis in docker-compose.

Good level: you should be able to run your project from zero with docker compose up --build and debug why it fails.

Month 6 - Redis, Celery, background jobs

Goal: understand async processing.

Redis

- key-value store
- TTL
- cache
- pub/sub basics
- Redis as broker
- Redis as result backend

```
redis-cli
KEYS *
GET key
SET key value
TTL key
DEL key
```

Celery

- worker
- task
- broker
- result backend
- retry
- delay
- apply_async
- beat
- queues
- routing
- task id
- idempotency

```
@app.task(bind=True, max_retries=3)
def process_quality_job(self, batch_id):
    try:
        ...
    except Exception as exc:
        raise self.retry(exc=exc, countdown=10)
```

Practice

Move a slow endpoint into Celery. Bad: POST /run-quality-check waits 2 minutes. Good: POST creates a job and returns job_id; worker processes in background; GET /jobs/{job_id} returns status.

Important concepts

- at least once execution
- duplicate task problem

- idempotency
- retry danger
- dead letter queue concept

Month 7 - Kafka and message brokers

Goal: use message brokers correctly in backend systems, not become a Kafka admin.

General broker concepts

- producer
- consumer
- topic
- queue
- partition
- offset
- consumer group
- ack
- retry
- dead letter queue
- event-driven architecture

Kafka specifically

- topic
- partition
- offset
- consumer group
- retention
- rebalancing
- keyed messages
- ordering per partition
- schema/versioning

RabbitMQ = task/message queue. Kafka = event log / streaming platform.

Practice

Build a Django endpoint that produces a Kafka event, a Kafka consumer that receives it, validates payload, starts a background process or Celery task, and sends failed messages to a DLQ topic.

```
etl_quality
etl_quality_dlq
etl_ingestion_events
```

Use versioned event messages instead of random unversioned JSON.

```
{
  "event_id": "uuid",
  "event_type": "quality.requested",
  "version": 1,
  "timestamp": "...",
  "payload": {}
}
```

Month 8 - Observability: ELK and OpenTelemetry

Goal: become much more professional at debugging and production diagnosis.

Logging

```
{
  "service_name": "etl-qualitymanagement",
  "event_action": "quality.kafka.message.received",
  "event_outcome": "success",
  "trace_id": "...",
  "batch_id": "..."
}
```

ELK

- Elasticsearch
- Logstash
- Kibana
- index
- mapping
- query DSL basics
- dashboard
- alerting basics

Practice: send Django logs to Elasticsearch, create a Kibana dashboard, and search errors by batch_id and correlation_id.

OpenTelemetry

- trace
- span
- span attributes
- trace id
- span id
- context propagation
- instrumentation
- exporter
- collector
- OTLP

- Jaeger/Tempo

Practice tracing request from API to service to DB to Kafka producer, and tracing Kafka consumer to Spark runner spawn.

Important idea

```
Logs answer: What happened?
Metrics answer: How many / how fast / how often?
Traces answer: Where did time/failure happen across services?
```

Month 9 - FastAPI, gRPC, GraphQL

This month is for API expansion. Do not let it distract you before you are good at Django.

FastAPI

- Path operation
- Pydantic models
- Depends
- dependency injection
- async basics
- middleware
- exception handlers
- OpenAPI

Build the same quality job API in FastAPI and compare it with Django Ninja.

gRPC

- .proto files
- service
- message
- unary call
- server streaming basics
- client stub
- protobuf

Use gRPC for service-to-service communication, high performance internal APIs, strict schema, or streaming. Do not use it just because it sounds advanced.

GraphQL

- schema
- query
- mutation
- resolver
- N+1 problem

- DataLoader
- pagination

Use GraphQL when frontend needs flexible querying. For admin dashboards and internal panels, REST is often simpler.

Practice

Build one small service with REST endpoint, GraphQL query, and gRPC method for the same data.

Month 10 - Capstone: production-level ETL backend module

Build one complete backend module similar to your real project: ETL Quality Orchestration Service.

Features

- Django / Django Ninja API
- PostgreSQL models
- Redis cache
- Celery worker
- Kafka producer/consumer
- Docker compose
- structured logs
- OpenTelemetry traces
- ELK dashboard
- tests
- management commands
- permissions/auth basics

Required flows

- POST /api/quality/requests creates DB record and publishes Kafka event.
- Consumer receives batch_id, dataset_urn, file_path, and bucket_name.
- Consumer validates payload.
- Starts Celery task or subprocess.
- Stores status, score, summary, error_message, started_at, and finished_at.
- Logs quality.request.created, quality.kafka.message.received, quality.job.started, quality.job.success, quality.job.failed.
- Every exception uses your service exceptions.
- GET /api/quality/reports/{batch_id} returns report.

Weekly routine

3 days learning

Each day: 60 minutes reading/watching, 90 minutes coding, 30 minutes notes/refactor.

2 days project practice

Work only on your mini project or real ETL project.

1 day review

```
What did I learn?
What did I build?
What broke?
What confused me?
Can I explain it simply?
```

1 day rest/light review

Read code. Do not burn out.

Daily practice template

```
1. Learn one concept
2. Write code using it
3. Break it intentionally
4. Debug it
5. Write notes
6. Refactor
7. Commit with a clean message
```

```
git checkout -b practice/celery-retry
# code
pytest
git add .
git commit -m "practice celery retry handling"
```

What you should be able to do after 10 months

- Design Django models correctly
- Write clean service-layer code
- Build REST APIs with Django Ninja and DRF
- Use PostgreSQL indexes and transactions
- Use Docker for local development
- Debug Linux server issues
- Resolve Git conflicts safely
- Use Redis and Celery for background jobs
- Consume and produce Kafka messages
- Design event payloads
- Add structured logs
- Use OpenTelemetry traces
- Search logs in ELK
- Understand when to use FastAPI, GraphQL, and gRPC

- Review backend code with confidence

What to remove or reduce from focus

Do not deeply learn both DRF and Django Ninja at the same time

For your current work: Primary: Django Ninja. Secondary: DRF.

Do not deeply learn GraphQL now

GraphQL is useful, but not urgent for your backend level. Learn it after you are comfortable with REST.

Do not deeply learn gRPC now

Useful for microservices, but not needed before you understand REST, Docker, DB, and async jobs well.

Do not treat DataHub as a general backend skill

DataHub is domain-specific. Learn it because your project needs metadata/lineage, but do not let it replace core backend learning.

Your strongest path based on your current project

Because you are already working on ETL, RCA, traceability, governance, logs, quality, Kafka, and observability, your best specialization is:

```
Backend engineer for data platforms / observability-heavy systems
```

That is better than becoming a generic CRUD Django developer.

```
Django + PostgreSQL + Kafka + Celery + Docker + OpenTelemetry + ELK
```

Month-by-month summary

Month	Main focus	Output
1	Python backend foundations	Clean Python package with validators/services/tests
2	Django core	Models, migrations, ORM, management commands
3	REST, Ninja, DRF	CRUD APIs with validation and errors
4	PostgreSQL	Optimized queries, indexes, transactions
5	Linux, Git, Docker	Dockerized Django/Postgres/Redis app
6	Redis, Celery	Async job system with retries/status
7	Kafka/message brokers	Event-driven quality pipeline
8	ELK/OpenTelemetry	Logs, traces, metrics, dashboards
9	FastAPI, gRPC, GraphQL	Compare API styles
10	Capstone	Production-like ETL backend module

The biggest mistake to avoid

Do not study like this:

```
Today Python
Tomorrow Docker
Next day Kafka
Next day GraphQL
Next day OpenTelemetry
```

That creates shallow knowledge. Study like this instead:

```
Build one backend system.
Add one skill to it every month.
Keep improving the same system.
```

That is how you become good.