

Back End and Docker: Exam Revision

Oct 9, 2026 · @Vihaan

Built only from your five PDFs: Unit 2 notes, Unit 2 code reference, Unit 5 notes, Unit 5 coding reference and the Docker cheat sheet.

Unit 2 theory: Back End Development

Tags like [N2 4.4] mean section 4.4 of the Unit 2 Notes PDF. The unit has seven blocks that build on each other: Python core, networking and files, Django MVT, auth and sessions, database connectivity, Django REST Framework (DRF), and machine learning integration. [N2 1]

1. Python core [N2 2]

args** and *kwargs** [N2 2.1]

```
def order_pizza(size, *toppings, **extras): ...
order_pizza("L", "olive", "corn", cheese=True)
```

Parameter	Role	Value in the call
size	Normal positional parameter	"L"
*toppings	Extra positional arguments collected into a tuple	("olive", "corn")
**extras	Extra keyword arguments collected into a dict	{"cheese": True}

- At the call site, ***** and ****** work in reverse: they unpack a list or dict into separate arguments. Django uses this to forward form data and URL keyword arguments.

Exception handling [N2 2.2]

Block	Runs when
try	Always, the risky code is attempted
except	Only if an error matched inside try
else	Only if try finished with no error
finally	Always, error or not

- Catch a specific error, e.g. `except ValueError as e:`. Never use a bare `except:` because it hides mistakes like a `NameError` typo.
- Several `except` blocks are checked top to bottom and only the first match runs.
- Django raises `Http404` when a row is not found and turns it into a proper 404 page.

Classes [N2 2.3]

- A class is a blueprint, each call makes an independent instance with its own state.
- `__init__` is the constructor and runs automatically at creation. `self` is the specific instance and is the first parameter of every ordinary method but is never passed explicitly.
- Attributes set with `self.x` in `__init__` belong to one instance. An attribute written in the class body (outside methods) is shared across all instances unless overridden.

Inheritance [N2 2.4]

```
class Animal:
    def speak(self): ...
class Dog(Animal):
    def speak(self): return "Woof"
```

- A child gets everything the parent has. Redefining an inherited method is **method overriding**. `super()` calls the parent's version, e.g. `super().speak()`. In Django you often override one method of a big built-in class and call `super()` to keep the framework's own logic.

Magic (dunder) methods [N2 2.5]

Method	Powers	Called as
<code>__init__</code>	Runs when an object is created	<code>obj = Dog()</code>
<code>__str__</code>	What <code>print()</code> shows	<code>print(obj)</code>
<code>__len__</code>	The <code>len()</code> built-in	<code>len(obj)</code>
<code>__eq__</code>	What <code>==</code> means	<code>a == b</code>
<code>__repr__</code>	Developer-facing representation	<code>repr(obj)</code>

- `__str__` is readable for people. `__repr__` is unambiguous for debugging, ideally enough to recreate the object. If only `__repr__` is defined, Python falls back to it wherever `__str__` is needed.

Four pillars of OOP [N2 2.6]

Pillar	Meaning	Example
Encapsulation	Data and methods bundled in one class, internals kept out of reach	name and bark() live together inside Dog
Abstraction	Caller uses a simple interface without knowing the inside	Calling bark() without knowing how the sentence is built
Inheritance	Child reuses and extends a parent	Dog(Animal), Cat(Animal)
Polymorphism	Same call behaves differently per object	speak() returns "Woof" or "Meow"

- Python has no truly private attributes. `self._balance` (single underscore) means "internal, do not touch". `self.__balance` (double underscore) triggers **name mangling**.

2. Networking and files [N2 3]

- A **socket** is the endpoint of a connection, identified by an IP address plus a port. Django's dev server default port is 8000. Port 80 is default for ordinary HTTP in production. [N2 3.1]
- Client calls: `socket()` create, `connect()` reach a server, `send()` / `recv()` exchange bytes, `close()` release. `AF_INET` = normal IP address. `SOCK_STREAM` = TCP, a reliable ordered byte stream. HTTP is a text protocol spoken over such a socket. [N2 3.2]
- Server calls: `bind()` attach to IP and port, `listen(5)` start waiting (queue up to 5), `accept()` wait for a client and return a **new** socket for that client plus its address. Sockets carry bytes (`b"..."`), so strings are encoded before sending. `runserver` is at its core this accept loop. [N2 3.5]

HTTP method	Intent
GET	Read a resource
POST	Create a resource
PUT	Replace a resource
DELETE	Remove a resource

- A view often checks `request.method` : GET shows a form or page, POST acts on submitted data. GET and PUT are expected to be idempotent (repeating has the same effect as once), which is why forms that create something use POST. [N2 3.3]

File mode	Behaviour
r	Read, file must exist
w	Write, overwrites completely
a	Append to the end
r+	Read and write, no truncation

- `with open(...)` as `f`: closes the file automatically even on error. Adding `b` (`rb`, `wb`) gives binary mode for images, PDFs and `.pk1` files. [N2 3.4]
- JSON: `json.dumps` (dict to string), `json.loads` (string to dict), `json.dump` (dict to file), `json.load` (file to dict). The `s` stands for string. Object becomes dict, array becomes list, `true/false/null` become `True/False/None`. [N2 3.6]

3. Django MVT [N2 4]

Part	File	Job
Model	<code>models.py</code>	The data
View	<code>views.py</code>	The logic
Template	<code>templates/</code>	The HTML

- `urls.py` routes a request to a view, the view asks a model for data and fills a template. MVT maps to MVC like this: Django's view is MVC's controller, and Django's template is MVC's view. [N2 4.1]
- **Project vs app**: project = one per site (`settings.py`, `root urls.py`, `wsgi.py`). App = one per feature (orders, accounts), reusable. An app only takes effect once listed in `INSTALLED_APPS`. `startapp` alone is not enough. [N2 4.2]
- **Anatomy of one request**: URL matched, view function runs, view may ask a model for data, template turns the result into HTML, response travels back. A view can also return a redirect or JSON. [N2 4.3]
- **Model to table**: write the model, `makemigrations` writes the migration file (does not touch the database), `migrate` applies it. Table name looks like `orders_order`. Migrations are version control for the schema and are committed to Git. [N2 4.4]
- **Templates**: `{{ variable }}` prints a value, `{% tag %}` adds logic (if, for, block). A template never runs raw Python. Filters like `|length`, `|date`, `|default` transform values. Inheritance: base template with `{% block content %}`, other templates start with `{% extends "base.html" %}`. [N2 4.5]

- **Fields:** CharField (max_length required), TextField, IntegerField/FloatField, DecimalField (exact, use for money), BooleanField, DateTimeField (`auto_now_add` sets it once on creation). Django adds the `id` primary key automatically. [N2 4.6]

Relationship field	Meaning	Example
ForeignKey	Many to one	Many orders belong to one user
OneToOneField	One to one	One profile per user
ManyToManyField	Many to many	Students and courses

- `on_delete` : CASCADE deletes the orders too, PROTECT blocks deleting the user, SET_NULL keeps orders with an empty customer. `__str__` labels rows in the admin site.
- **Views and URLs:** a view takes a request and returns a response. `render(request, template, context)` where context is a dict whose keys become template variable names. `get_object_or_404` shows a 404 page instead of crashing. `path("<int:pk>/", view, name=...)` uses a path converter that passes `pk` to the view as a keyword argument. Root urls use `include("orders.urls")`. `name=` lets templates use `{% url 'order-detail' order.id %}`. After a POST, return `redirect()` so refreshing does not submit again. [N2 4.7]

4. Auth and sessions [N2 5]

- `authenticate(request, username=u, password=p)` returns the user or `None`. `login(request, user)` attaches the user to the session. `logout(request)` clears it. [N2 5.1]
- The User model has hashed password plus flags `is_active`, `is_staff`, `is_superuser`. Create users with `User.objects.create_user(...)`, never `User(...)`, because only `create_user` hashes the password.
- Protect views with `@login_required` (redirects anonymous users) and `@permission_required` (needs a permission). `request.user.is_authenticated` and `request.user.has_perm(...)` work in code. Permissions are usually bundled into Groups. [N2 5.2]
- **Session:** only a session ID travels in the cookie (`sessionid`). The real data stays on the server in the `django_session` table (key, data, expiry). `SESSION_COOKIE_AGE` controls how long it lasts. Sessions also make CSRF protection possible: a one-time token tied to the session must come back with every POST form. [N2 5.3]

5. Database connectivity [N2 6]

	SQLite	PostgreSQL
Where it lives	One file in the project	Separate server process, port 5432
Setup	None, Django default	Running server plus settings
Many users writing	Weak, locks the whole file	Strong
Typical use	Learning, prototypes	Production

- Switching is a configuration change, not a rewrite, because Django talks to the database through the ORM. [N2 6.1]
- Install the driver `pip install psycopg2-binary` and set `DATABASES` with `ENGINE django.db.backends.postgresql`, `NAME`, `USER`, `PASSWORD`, `HOST`, `PORT`. Run `migrate` again because the new database is empty. Read the password from an environment variable, not `settings.py`. [N2 6.2]
- ORM: `Order.objects.all()`, `.get(id=7)` (exactly one row), `.filter(paid=False)`, `.filter(price__gte=50)`, `.filter() / all()` return a **lazy QuerySet**: SQL runs only when it is looped over, counted or displayed. `print(qs.query)` shows the SQL. [N2 6.3]
- Raw SQL: `cursor.execute("... WHERE id = %s", [user_id])`. Always use `%s` placeholders and pass values separately to prevent **SQL injection**. The ORM does this automatically. [N2 6.4]

6. APIs with Django REST Framework [N2 7]

- An API returns JSON (not HTML) so any client can use it. DRF adds three ideas: **serializers**, **ViewSet**s, **routers**. Install `pip install djangorestframework`, add `"rest_framework"` to `INSTALLED_APPS`. REST: GET `/api/users/` lists, POST creates, GET `/api/users/5/` reads one, PUT replaces, DELETE removes. HTML views stay logged in with a session cookie, API views with a token. [N2 7.1]
- **Serializer** translates between model instance and JSON, both directions, and validates incoming data (bad data gives a 400 with an error message). `ModelSerializer` builds fields from the model. The `fields` list decides what is exposed, so the hashed password never leaks. [N2 7.2]
- **ViewSet**: `ModelViewSet` with `queryset` and `serializer_class` gives list, create, retrieve, update, delete (five views in two lines). **Router**: `DefaultRouter().register("users", UserViewSet)` creates `/api/users/` and `/api/users/<id>/`. For custom logic (like ML prediction) inherit from `APIView` and define one method per HTTP verb. [N2 7.3]

- **Token auth:** header `Authorization: Token <key>` (DRF uses the word Token, many APIs use Bearer). Needs `"rest_framework.authtoken"` in `INSTALLED_APPS` (only creates the table) and `DEFAULT_AUTHENTICATION_CLASSES` (TokenAuthentication plus SessionAuthentication) to actually read the header. Create a token with `python manage.py drf_create_token <username>`. [N2 7.4]

Permission class	Allows
AllowAny	Everyone (DRF default)
IsAuthenticated	Valid session or token only
IsAdminUser	Only <code>is_staff = True</code>
IsAuthenticatedOrReadOnly	Anyone reads, only logged-in users change data

- Authentication says who is asking. Permissions decide if they are allowed in. With no `permission_classes` a view is open to everyone even if tokens are configured. Adding `[IsAuthenticated]` turns a missing token into a 401 "Authentication credentials were not provided." [N2 7.5]

7. Machine learning model integration [N2 8]

- Example: Iris dataset, 4 measurements in, one of 3 species out. **Train once in a separate script**, save with `joblib.dump(clf, "iris_model.pkl")` (binary file). The server never trains, because training takes minutes and a request must answer in milliseconds. [N2 8.1]
- **Load once, not per request:** `model = joblib.load(MODEL_PATH)` at module top level. A module's top-level code runs only on first import, so it is read once when the server starts. `pathlib` builds the path relative to the file. [N2 8.2]
- Endpoint: plain `Serializer` (no database model) validates four `FloatFields`, `APIView` with `post()`. `permission_classes = [IsAuthenticated]` stops anyone using the model. `is_valid(raise_exception=True)` returns 400 on bad input. `[...]` because scikit-learn expects a batch of rows. `int(p)` because a NumPy integer cannot be turned into JSON. Valid token gives `{"prediction": 0}` (setosa), no token gives 401. [N2 8.3]
- From a browser page: use `fetch` with `async/await`. The session cookie goes automatically, so no token pasted. CSRF applies: read the `csrftoken` cookie and send it in the `X-CSRFToken` header (JavaScript equivalent of `{% csrf_token %}`). [N2 8.4]

Unit 2 code: what to write in the exam

Tags like [C2 6.3] mean section 6.3 of the Unit 2 Code Reference Manual. Learn each block below well enough to write it from memory.

Python core [C21]

```
# *args and **kwargs
def order_pizza(size, *toppings, **extras):
    print(size, toppings, extras)    # L ('olive', 'corn') {'cheese': True}

order_pizza("L", "olive", "corn", cheese=True)
order_pizza("L", *["olive", "corn"], **{"cheese": True})    # unpack at the
call
```

```
# exceptions
try:
    result = 10 / int(input("n: "))
except ZeroDivisionError:
    print("cannot divide by zero")
except ValueError as e:
    print("bad input:", e)
else:
    print("ok", result)    # only if try had no error
finally:
    print("always runs")

raise ValueError("age cannot be negative")    # raise your own
```

```
# classes, inheritance, super, dunder methods
class Animal:
    species = "unknown"    # class attribute, shared by all instances
    def __init__(self, name):    # constructor
        self.name = name    # instance attribute
    def speak(self):
        return "..."
    def __str__(self):    # print(obj)
        return f"{self.name}: {self.speak()}"
    def __repr__(self):    # repr(obj), shell output
        return f"{type(self).__name__}({self.name!r})"
    def __eq__(self, other):    # a == b
        return self.name == other.name
    def __len__(self):    # len(obj)
        return len(self.name)
```

```

class Dog(Animal):
    def speak(self):                # override
        return "Woof"

class Puppy(Dog):
    def __init__(self, name, toy):
        super().__init__(name)      # run the parent constructor
        self.toy = toy
    def speak(self):
        return super().speak() + "!" # reuse the parent version

print(Dog("Rex"))                   # Rex: Woof
print(Puppy("Bit", "ball"))         # Bit: Woof!
print(Dog("Rex") == Dog("Rex"))    # True

```

```

# the four pillars in a few lines
class Account:
    def __init__(self, balance):
        self._balance = balance    # encapsulation: internal by convention
        self.__pin = 1234          # name mangled to _Account__pin
    def deposit(self, amount):      # abstraction: callers do not see the
checks
        if amount <= 0:
            raise ValueError("amount must be positive")
        self._balance += amount

for pet in [Dog("Rex"), Animal("Tom")]:
    print(pet.speak())              # polymorphism: same call, different
result
# inheritance: class Dog(Animal) above

```

Networking and files [C22]

```

# socket client: raw HTTP request. Steps: socket, connect, send/recv, close
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("example.com", 80))
s.sendall(b"GET / HTTP/1.1\r\nHost: example.com\r\nConnection:
close\r\n\r\n")
reply = b""
while chunk := s.recv(4096):        # empty bytes means the server closed
    reply += chunk
s.close()
print(reply.decode()[:300])

```

```
# socket server: echo. Steps: bind, listen, accept, recv/send, close
import socket
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server.bind(("127.0.0.1", 9000))
server.listen(5)
while True:
    conn, addr = server.accept()    # new socket per client
    data = conn.recv(1024)
    conn.send(b"Echo: " + data)    # sockets carry bytes, not str
    conn.close()
```

```
# files: r read, w overwrite, a append, rb/wb binary
with open("notes.txt", "w") as f:
    f.write("line one\n")
with open("notes.txt", "a") as f:
    f.write("line two\n")
with open("notes.txt") as f:
    text = f.read()                # or f.readline(), f.readlines(), for line
    in f

# json: dumps/loads on strings, dump/load on files
import json
order = {"id": 42, "items": ["samosa", "chai"], "paid": True}
text = json.dumps(order)
back = json.loads(text)
with open("order.json", "w") as f:
    json.dump(order, f)
with open("order.json") as f:
    data = json.load(f)
```

HTTP method	Intent	Success status
GET	Read	200
POST	Create	201
PUT	Replace	200
DELETE	Remove	204

Request format: `GET /api/orders/42 HTTP/1.1`, then `Host: example.com`, then
`Authorization: Token <token>`. [C2 2.4]

Django project setup [C2 3]

```
python -m venv venv
source venv/bin/activate          # Windows: venv\Scripts\Activate.ps1
pip install django
django-admin startproject mysite .
python manage.py startapp orders
python manage.py migrate
python manage.py createsuperuser
python manage.py runserver        # http://127.0.0.1:8000/
```

```
# mysite/settings.py
INSTALLED_APPS = [
    # ... Django defaults ...
    "orders",                # register every app you create
]
LOGIN_URL = "login"
LOGIN_REDIRECT_URL = "dashboard"
LOGOUT_REDIRECT_URL = "login"
SESSION_COOKIE_AGE = 60 * 60 * 24 * 7    # seconds
```

MVT: model, view, urls, templates [C2 4]

```
# orders/models.py
from django.db import models
from django.contrib.auth.models import User

class Order(models.Model):
    customer = models.ForeignKey(User, on_delete=models.CASCADE)
    item = models.CharField(max_length=100)
    quantity = models.IntegerField(default=1)
    price = models.DecimalField(max_digits=8, decimal_places=2)
    paid = models.BooleanField(default=False)
    created_at = models.DateTimeField(auto_now_add=True)
    def __str__(self):
        return f"{self.item} x {self.quantity}"
```

```
python manage.py makemigrations    # write the migration file
python manage.py migrate           # apply it to the database
```

```
# orders/views.py
from django.shortcuts import render, redirect, get_object_or_404
from .models import Order

def order_list(request):
```

```

orders = Order.objects.filter(paid=False)
return render(request, "orders/list.html", {"orders": orders})

def order_detail(request, pk):
    order = get_object_or_404(Order, pk=pk)
    return render(request, "orders/detail.html", {"order": order})

def order_create(request):
    if request.method == "POST":
        Order.objects.create(
            customer=request.user,
            item=request.POST["item"],
            quantity=int(request.POST["quantity"]),
            price=request.POST["price"],
        )
        return redirect("order-list")          # redirect after POST
    return render(request, "orders/form.html")

```

```

# orders/urls.py (create this file)
from django.urls import path
from . import views

urlpatterns = [
    path("", views.order_list, name="order-list"),
    path("new/", views.order_create, name="order-create"),
    path("<int:pk>/", views.order_detail, name="order-detail"), #
view(request, pk=7)
]

# mysite/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path("admin/", admin.site.urls),
    path("orders/", include("orders.urls")),
]

```

```

<!-- templates/base.html (add BASE_DIR / "templates" to TEMPLATES DIRS) -->
<!DOCTYPE html>
<html>
<body>
    <nav>Hi {{ request.user.username }}</nav>
    {% block content %}{% endblock %}
</body>
</html>

```

```

<!-- orders/templates/orders/list.html -->
{% extends "base.html" %}
{% block content %}
    {% if orders %}
        <p>{{ orders|length }} orders</p>
        {% for o in orders %}
            <p><a href="{% url 'order-detail' o.pk %}">{{ o.item }}</a>, {{
o.created_at|date:"d M Y" }}</p>
        {% endfor %}
    {% else %}
        <p>No orders yet.</p>
    {% endif %}
{% endblock %}

<!-- orders/templates/orders/form.html -->
{% extends "base.html" %}
{% block content %}
    <form method="post">
        {% csrf_token %}
        <input name="item" required>
        <input name="quantity" type="number" value="1">
        <input name="price" type="number" step="0.01" required>
        <button type="submit">Save</button>
    </form>
{% endblock %}

```

Template syntax	Purpose
<code>{{ variable }}</code>	Print a value
<code>{{ value length }}</code>	Filter (also date, default, upper)
<code>{% if %} {% else %} {% endif %}</code>	Branching
<code>{% for x in list %} {% endfor %}</code>	Loop
<code>{% url 'name' arg %}</code>	URL by name
<code>{% extends "base.html" %}</code>	Inherit the base layout
<code>{% block content %}{% endblock %}</code>	Fillable region
<code>{% csrf_token %}</code>	Required in every POST form

Auth and sessions [C2 5]

```
# accounts/views.py
from django.contrib.auth import authenticate, login, logout
from django.contrib.auth.models import User
from django.contrib.auth.decorators import login_required,
permission_required
from django.shortcuts import render, redirect
from django.http import HttpResponse

def login_view(request):
    if request.method == "POST":
        user = authenticate(request, username=request.POST["username"],
                             password=request.POST["password"])
        if user is not None:          # None means wrong credentials
            login(request, user)      # creates the session and cookie
            return redirect("dashboard")
    return render(request, "index.html")

def logout_view(request):
    logout(request)                  # clears the session
    return redirect("login")

def register_view(request):
    if request.method == "POST":
        user = User.objects.create_user( # hashes the password
            username=request.POST["username"],
            password=request.POST["password"])
        login(request, user)
        return redirect("dashboard")
    return render(request, "register.html")

@login_required                     # anonymous users go to LOGIN_URL
def dashboard(request):
    return render(request, "dashboard.html")

@permission_required("orders.delete_order", raise_exception=True) # 403
without it
def delete_order(request, pk):
    ...

def visits(request):                # sessions in code
    request.session["n"] = request.session.get("n", 0) + 1
    return HttpResponse(request.session["n"])
```

```
# accounts/urls.py, included from mysite/urls.py as path("accounts/",
include("accounts.urls"))
urlpatterns = [
    path("login/", views.login_view, name="login"),
    path("logout/", views.logout_view, name="logout"),
    path("register/", views.register_view, name="register"),
    path("dashboard/", views.dashboard, name="dashboard"),
]
```

```
<!-- every POST form needs the csrf tag -->
<form method="post">
    {% csrf_token %}
    <input name="username">
    <input name="password" type="password">
    <button type="submit">Log in</button>
</form>

<!-- checks in templates -->
{% if request.user.is_authenticated %}Welcome {{ request.user.username }}{%
endif %}
{% if perms.orders.delete_order %}<button>Delete</button>{% endif %}
```

```
# checks in code, and groups (python manage.py shell)
request.user.is_authenticated
request.user.has_perm("orders.delete_order")    # add_, change_, delete_,
view_ per model

from django.contrib.auth.models import Group, Permission
editors = Group.objects.create(name="editors")
editors.permissions.add(Permission.objects.get(codename="delete_order"))
User.objects.get(username="asha").groups.add(editors)
```

Database connectivity [C26]

```
pip install psycopg2-binary
docker run --name imap-postgres -e POSTGRES_PASSWORD=yourpassword \
    -e POSTGRES_DB=imapdb -p 5432:5432 -d postgres:16
docker start imap-postgres          # next time
```

```
# mysite/settings.py
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": "imapdb",
        "USER": "postgres",
        "PASSWORD": "yourpassword",    # in real projects:
os.environ["DB_PASSWORD"]
        "HOST": "localhost",
        "PORT": "5432",
    }
}
# then: python manage.py migrate    and    python manage.py createsuperuser
```

```
# ORM
from django.db.models import Q, Sum, Count

Order.objects.all()
Order.objects.get(pk=7)                # exactly one, else
an error
Order.objects.filter(paid=False, price__gte=50)    # AND
Order.objects.filter(Q(item="tea") | Q(item="chai")) # OR
Order.objects.exclude(paid=True)
Order.objects.filter(item__icontains="te")        # also __startswith,
__in, __lt, __range
Order.objects.order_by("-price")[:5]              # top 5 by price
Order.objects.filter(paid=False).count()
Order.objects.aggregate(total=Sum("price"))
Order.objects.create(customer=u, item="tea", price="40")
Order.objects.filter(paid=False).update(paid=True)
o.save(); o.delete()
print(Order.objects.filter(paid=False).query)      # the SQL Django
writes
```

Lookup	Meaning
<code>__gt</code> , <code>__gte</code> , <code>__lt</code> , <code>__lte</code>	Greater, less (or equal)
<code>__contains</code> , <code>__icontains</code>	Text contains (case insensitive)
<code>__startswith</code>	Text starts with
<code>__in=[...]</code>	Value in a list
<code>__range=(a, b)</code>	Between two values

```
# raw SQL: always placeholders, never string concatenation
from django.db import connection
with connection.cursor() as cursor:
    cursor.execute("SELECT username FROM auth_user WHERE id = %s", [user_id])
    row = cursor.fetchone()
```

Django REST Framework [C27]

```
# install: pip install djangorestframework
# mysite/settings.py
INSTALLED_APPS += ["rest_framework", "rest_framework.authtoken"]
REST_FRAMEWORK = {
    "DEFAULT_AUTHENTICATION_CLASSES": [
        "rest_framework.authentication.TokenAuthentication",
        "rest_framework.authentication.SessionAuthentication",
    ],
}
```

```
# orders/serializers.py
from rest_framework import serializers
from django.contrib.auth.models import User

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ["id", "username", "email"]    # password never exposed
```

```
# orders/api_views.py
from rest_framework import viewsets
from rest_framework.permissions import IsAuthenticated
from django.contrib.auth.models import User
from .serializers import UserSerializer

class UserViewSet(viewsets.ModelViewSet):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    permission_classes = [IsAuthenticated]    # default is AllowAny
```

```
# mysite/urls.py
from rest_framework.routers import DefaultRouter
from orders.api_views import UserViewSet
```

```

router = DefaultRouter()
router.register("users", UserViewSet)          # /api/users/ and
/api/users/<id>/
urlpatterns += [path("api/", include(router.urls))]

```

```

python manage.py migrate                      # creates the token table
python manage.py drf_create_token asha       # prints the token

```

```

curl http://127.0.0.1:8000/api/users/ -H "Authorization: Token <token>" #
200
curl -i http://127.0.0.1:8000/api/users/                                     #
401

```

Status	Meaning
200, 201, 204	OK, created, deleted
400	Serializer validation failed
401	Missing or wrong token
403	Logged in but not allowed
404	Object not found

Machine learning integration [C28]

```

pip install scikit-learn joblib
python manage.py startapp mlpredict_iris
mkdir mlpredict_iris/ml

```

```

# train_model.py (run once: python train_model.py)
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
import joblib

X, y = load_iris(return_X_y=True)
clf = RandomForestClassifier().fit(X, y)
joblib.dump(clf, "mlpredict_iris/ml/iris_model.pkl")

```

```
# mlpredict_iris/ml_model.py : loaded once at import, not per request
import joblib
from pathlib import Path

MODEL_PATH = Path(__file__).resolve().parent / "ml" / "iris_model.pkl"
model = joblib.load(MODEL_PATH)
```

```
# mlpredict_iris/views.py
from rest_framework import serializers
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.permissions import IsAuthenticated
from .ml_model import model

class IrisInputSerializer(serializers.Serializer):
    sepal_length = serializers.FloatField()
    sepal_width = serializers.FloatField()
    petal_length = serializers.FloatField()
    petal_width = serializers.FloatField()

class PredictView(APIView):
    permission_classes = [IsAuthenticated]      # without it the model is
open to all

    def post(self, request):
        s = IrisInputSerializer(data=request.data)
        s.is_valid(raise_exception=True)        # bad input returns 400
        d = s.validated_data
        p = model.predict([[d["sepal_length"], d["sepal_width"],
                                d["petal_length"], d["petal_width"]]]])[0]    #
list inside a list
        return Response({"prediction": int(p)}) # int(): NumPy ints are not
JSON safe

# mlpredict_iris/urls.py : path("api/predict/", PredictView.as_view())
# mysite/urls.py         : path("", include("mlpredict_iris.urls"))
```

```
curl -X POST http://127.0.0.1:8000/api/predict/ \
-H "Authorization: Token <token>" -H "Content-Type: application/json" \
-d
'{"sepal_length":5.1,"sepal_width":3.5,"petal_length":1.4,"petal_width":0.2}'
# {"prediction": 0}
```

```
// from a logged in browser page: session cookie plus CSRF header, no token
function getCookie(name) {
  return document.cookie.split("; ")
    .find(row => row.startsWith(name + "="))?.split("=")[1];
}
const res = await fetch("/api/predict/", {
  method: "POST",
  headers: { "Content-Type": "application/json", "X-CSRFToken":
getCookie("csrftoken") },
  body: JSON.stringify({ sepal_length: 5.1, sepal_width: 3.5, petal_length:
1.4, petal_width: 0.2 }),
});
console.log(await res.json());
```

Common errors and fixes [C2 9]

Error	Fix
NameError in urls.py or views.py	Add the missing import (path, include, views)
Model or template not found	Add the app to INSTALLED_APPS
no such table / relation does not exist	<code>python manage.py migrate</code>
Model change not in database	Run both makemigrations and migrate
TemplateDoesNotExist	Path must be app/templates/app/page.html
NoReverseMatch	<code>{% url %}</code> name must match <code>name=</code> in urls.py, pass o.pk if the path has <code><int:pk></code>
MultiValueDictKeyError	Form <code>name=</code> must match <code>request.POST["x"]</code>
403 CSRF verification failed	Add <code>{% csrf_token %}</code> or send X-CSRFToken
User cannot log in after creation	Use <code>create_user</code> , not <code>User(...)</code>
connection refused on 5432	<code>docker start imap-postgres</code>
ModuleNotFoundError	pip install the package, check the venv is active
API 401	Header must be <code>Authorization: Token <key></code>
API open to everyone	Set <code>permission_classes</code> , default is AllowAny
int64 is not JSON serializable	Wrap in <code>int(p)</code>
Expected 2D array	<code>model.predict([[a, b, c, d]])</code>
a bytes-like object is required	<code>.encode()</code> before send, <code>.decode()</code> after recv

Unit 5 theory: Project and DevOps with Docker

Tags like [N5 4.1] mean section 4.1 of the Unit 5 notes.

Why Docker exists [N5 2]

- An app is code plus runtime, libraries, OS packages and config. Git ships only the code, so the rest drifts between machines. That is the "works on my machine" problem.
- Running two projects that need different Python, Django and PostgreSQL versions on one laptop is called dependency hell.
- Docker packages the app with everything it needs into an **image** and runs it as an isolated process called a **container**.

- Shipping container idea: one standard box handled the same way by every ship, train and truck. Image is the box, Docker Engine is the crane.

Fix	What it isolates	Falls short because
venv	Python packages only	Not PostgreSQL, system libraries or the OS
README steps	Nothing	Goes out of date
Virtual machine	Everything	Gigabytes, minutes to boot
Container	Everything the app needs	Megabytes, starts in about a second

Why teams use Docker: consistency, speed, density (dozens of containers per laptop), isolation, and a rich ecosystem (Docker Hub official images). Docker is not a VM, not a programming language, and not a replacement for Git or venv.

Containers vs virtual machines [N5 3]

	Virtual machine	Container
Kernel	Its own, one per VM	Shared with the host
Size	Gigabytes	Megabytes
Start time	Tens of seconds to minutes	About a second
Per laptop	2 or 3	Dozens
Isolation	Strong, hardware level	Process level

- A container is just a Linux process restricted by the kernel.
- **Namespaces** control what it can see (own process list, network, hostname, filesystem).
- **Cgroups** control what it can use (CPU, memory, number of processes).
- On Windows and macOS, Docker Desktop runs a small Linux VM (WSL2 on Windows), because Linux containers need a Linux kernel.

Docker architecture [N5 4]

- **Client:** the `docker` command you type. It only sends requests.
- **Daemon:** background service that builds images, runs containers, manages storage and networks.
- **Registry:** stores and shares images, for example Docker Hub.

Object	What it is
Image	Read only template, built from a Dockerfile
Container	A running or stopped instance of an image
Volume	Docker managed storage that outlives containers
Network	Virtual switch; containers reach each other by name
Registry	Where images are stored and shared

Image and container relate like a class and an object: one image can make many independent containers.

What happens on `docker run` :

1. Client asks the daemon to create and start a container.
2. Daemon checks the local cache for the image.
3. If missing, it pulls each layer from the registry.
4. It stacks the layers and adds a thin writable layer on top.
5. It gives the container a network interface and publishes ports.
6. Kernel sets up namespaces and cgroups, and the app starts as process 1.

Images and the Dockerfile [N5 5]

- An image is a stack of **read only layers**, one per instruction that changes files, plus config (default command, env vars, ports).
- Identical layers are shared and cached. Images are immutable; a change makes a new image.
- Only `RUN`, `COPY` and `ADD` add size. `CMD` and `EXPOSE` only record settings.

Image name parts: registry (default docker.io), namespace (default library), repository, tag (default latest). So `postgres:16-alpine` means `docker.io/library/postgres:16-alpine`.

- A **digest** is a hash of exact content and never changes. A **tag** is only a label and can be moved.
- `latest` does not mean stable, only the most recent push. **Pin at least major and minor**, such as `python:3.12-slim`.

Base flavour	Use when
Full (python:3.12)	You need compilers, only in a build stage
slim	Default for Python apps
alpine	Tiny, but some Python packages do not install cleanly
distroless	Production, no shell or package manager, very little to attack

Lab result: 1.65 GB on python:3.12 vs 235 MB on python:3.12-slim, about seven times smaller.

Dockerfile order and the cache:

- Dockerfile is the recipe: FROM , WORKDIR , COPY , RUN , CMD .
- A cached layer is reused while that instruction and everything before it are unchanged. Once one layer changes, all later layers rebuild.
- So copy and install requirements.txt first, copy source code last. Lab: 1.0 s with the right order, 5.4 s with the wrong one.

Pairs that get confused:

Pair	Difference
CMD vs ENTRYPOINT	ENTRYPOINT is the program, CMD is its default arguments. Text after the image name in docker run replaces CMD
COPY vs ADD	COPY only copies. ADD also unpacks archives and downloads URLs. Prefer COPY
ARG vs ENV	ARG exists only at build time. ENV is baked into the image and seen by the app. Neither is safe for secrets

- **Build context:** the . at the end of docker build is the folder sent to the builder. Without .dockerignore , COPY . . would copy venv, .git and the .env file with passwords.
- **Multi stage builds:** one stage with build tools, a second slim stage copying only the result. A React app built with Node (about 1 GB) ships as static files on nginx (about 50 MB). For Python the size barely changes but the image is cleaner and safer.

Containers, storage and networking [N5 6]

- Files a container changes live in its thin writable layer, deleted with the container. **Code belongs in images, data belongs in volumes.**
- Lifecycle:** created, running, exited, removed. `docker run` does the first two. `docker ps` shows running, `docker ps -a` shows all. `--rm` removes the container on exit.

docker run option	Meaning
<code>--name imap-postgres</code>	Readable name instead of random ID
<code>-e POSTGRES_PASSWORD=...</code>	Sets an environment variable
<code>-p 5432:5432</code>	Publish a port, host on the left, container on the right
<code>-d</code>	Run in the background
<code>postgres:16</code>	Image and tag

Rule: options go before the image name. Anything after it is the command to run inside the container.

Storage	Volume	Bind mount	tmpfs
Managed by	Docker	You (host folder)	Memory
Best for	Databases, uploads	Source code in development	Temporary scratch files
Survives container removal	Yes	Yes	No

A bind mount gives real access to a folder on your laptop, so isolation is only as good as what you share.

Ports:

- A container is not reachable until a port is published with `-p HOST:CONTAINER` ("outside : inside").
- `EXPOSE` only documents a port; `-p` publishes it.
- The app must listen on `0.0.0.0`, not `127.0.0.1`, which inside a container means "this container only".
- "Port is already allocated" means the host port is taken; use another, such as `-p 8001:8000`.

Containers find each other by name: on a user defined network, Docker runs a small DNS server, so web connects to the database as `db:5432`. Names stay stable across restarts, IPs do not.

The localhost trap: inside the web container, `localhost` means the web container itself, so the database is not there. Most common beginner bug.

You are here	To reach PostgreSQL use
Your laptop, port published	<code>localhost:5432</code>
Another container on the same network	<code>db:5432</code> (service name)
A container reaching your laptop	<code>host.docker.internal:5432</code>

One image, many configurations: the same image used SQLite with no settings and PostgreSQL when given a `DATABASE_URL` env var, with no rebuild. Config comes from the environment, not the code.

Docker Compose [N57]

- Replaces several long commands (network, volume, each container) with one file `compose.yaml` and one command `docker compose up`.
- The file is declarative, version controlled and reproducible. Compose creates the network and volumes itself, and each **service name becomes a hostname**.

Compose key	Equivalent to
build / image	Build an image, or name one to pull
ports	<code>-p</code>
environment	<code>-e</code>
volumes	<code>-v</code>
restart	<code>--restart</code>
depends_on	Start order between services

- **Secrets in `.env`:** Compose reads `.env` and fills `${VARIABLE}`. Keep `.env` out of Git and images; commit a `.env.example` with placeholders.
- **Started is not ready:** a DB container can run while PostgreSQL is still starting. A **healthcheck** tests real readiness, and `depends_on` with `condition: service_healthy` makes web wait.

- **down vs down -v** : `down` removes containers and network but keeps volumes. `down -v` also deletes volumes and wipes the database.

Integrating Django with Docker [N5 8]

1. **gunicorn replaces runserver.** `runserver` is for development. `gunicorn` is a production server pointed at `wsgi.py` and bound to `0.0.0.0:8000`.
2. **Settings come from the environment.** `SECRET_KEY`, `DEBUG`, `ALLOWED_HOSTS` and the DB password move to `.env`. `DB_HOST` becomes `db`.
3. **Migrations run at start up, not build time.** The database does not exist during the build, so `migrate` runs when the web container starts, before `gunicorn`. The superuser is created inside the running container.
4. **Static files need serving.** With `DEBUG` off the admin looks unstyled. Run `collectstatic` and serve with **WhiteNoise**.

The trained `.pkl` model travels inside the image with the code. The scikit-learn version in the image must match the one that trained it.

Final stack: **web** (built from the Django Dockerfile, port 8000) and **db** (official PostgreSQL image, named volume, healthcheck). One `docker compose up` runs the whole Unit 2 app, API and ML endpoint.

Deployment and security [N5 9]

The image is the deliverable: tag it with a registry name, push it, pull and run it anywhere. Registries: Docker Hub (default), GitHub Container Registry, cloud ones like Amazon ECR.

Tag type	Example	Changes?	Use for
Version	1.0, 1.4.2	Never	Releases, what production runs
Git commit	sha-3f2a9c1	Never	Tracing an image to its code
Branch	main, dev	Every push	Test and staging
latest	latest	Every push	Convenience only

Never push different content under an existing version tag, and never run `latest` in production.

Apple M series Macs build `arm64` images while most servers are `amd64`, giving "exec format error". `docker buildx` builds for both.

CI/CD (for example GitHub Actions) runs on every `git push` on a fresh machine:

1. Build the image, reusing cached layers.
2. Test it by starting it and checking it responds.
3. Scan for known vulnerabilities.
4. Push to a registry with proper tags.
5. Deploy: the server pulls the new image and restarts.

A production stack adds: a reverse proxy (nginx, the only service open to the outside), scaling (several identical copies behind nginx), separate front and back networks (proxy cannot reach the database), and CPU and memory limits.

Security habits:

- Never put secrets in an image. `ENV` passwords can be read with `docker inspect`, and deleting them in a later layer does not remove them from the earlier one. Supply secrets at run time through `.env`.
- Run as a non root user created in the Dockerfile.
- Use a minimal, pinned base image.
- Use `.dockerignore` to keep `.env`, `.git` and keys out.
- Limit resources.
- Scan images and rebuild regularly.
- Never use `--privileged`, and never mount Docker's control socket into a container.
- Containers share one kernel, so they are isolated, but less strongly than VMs.

Best practice and debugging [N5 10]

Ten rules:

1. Pin a slim base image.
2. Copy dependencies first, code last.
3. One concern per container.
4. Bind to `0.0.0.0` inside containers.
5. Use `.dockerignore`.
6. Pass config through environment variables.
7. Log to the screen so `docker logs` shows it.
8. Keep data in volumes.
9. Run as a non root user.
10. Add a healthcheck and make dependents wait for it.

Debugging playbook, in order:

1. Is it running? `docker ps -a` (status and exit code).

2. What does it say? `docker logs` (most problems end here).
3. How is it configured? `docker inspect` (ports, env, mounts, networks).
4. Look inside: `docker exec` opens a shell.
5. Check the network: same network, service name not localhost, bound to 0.0.0.0.

Error	Usual cause
port is already allocated	Another program uses that host port
connection refused to the database	<code>localhost</code> used instead of service name, or DB not ready
exec format error	Image built for a different CPU architecture
Exited (137)	Out of memory
no space left on device	Old images and cache; run <code>docker system prune</code>
pull access denied	Typo in image name, or private repository
Database is uninitialized and superuser password is not specified	PostgreSQL started without <code>POSTGRES_PASSWORD</code>

Unit 5 code: Dockerfile, Compose and commands

Tags like [C5 4.2] mean section 4.2 of the Unit 5 Coding Reference Manual. Docker Desktop must be running before any command works.

Setup and images [C5 1, 2]

```

docker version          # client and daemon; error means Desktop not
started
docker compose version
docker run hello-world  # test image
docker ps -a            # finished container, status Exited (0)
docker images           # image stays in the local cache

```

Command	Purpose
<code>docker build -t name:tag .</code>	Build an image from the Dockerfile in this folder

Command	Purpose
<code>docker build -f Dockerfile.prod -t name:tag .</code>	Build from a Dockerfile with a different name
<code>docker images</code>	List images and sizes
<code>docker history name:tag</code>	Show each layer and its size
<code>docker pull postgres:16-alpine</code>	Download from a registry
<code>docker tag name:tag user/name:tag</code>	Give another name before pushing
<code>docker push user/name:tag</code>	Upload to a registry
<code>docker rmi name:tag</code>	Remove an image

Base image	Size	Use for
python:3.12	1.62 GB	Build stages that need compilers
python:3.12-slim	190 MB	Default for Python apps
alpine:3.20	12 MB	Tiny utilities; some Python packages will not install

Always pin at least major.minor. Never rely on `latest`.

Containers [C5 3]

Option	Purpose
<code>-d</code>	Run in the background
<code>-p 8000:8000</code>	Publish port, HOST:CONTAINER
<code>--name web</code>	Readable name
<code>-e KEY=value</code>	Set an environment variable
<code>--env-file .env</code>	Load variables from a file
<code>-v name:/path</code>	Mount a named volume
<code>--network net</code>	Join a user defined network
<code>-it</code>	Interactive terminal, for shells
<code>--rm</code>	Remove container on exit
<code>--restart unless-stopped</code>	Restart after crash or reboot

Options go before the image name. Anything after it replaces the image's CMD.

```
docker run -d -p 8080:80 --name web nginx:alpine      # open
http://localhost:8080
docker run -it --rm python:3.12-slim bash            # a throwaway shell
```

Everyday command	Purpose
<code>docker ps</code> / <code>docker ps -a</code>	Running containers / all including exited
<code>docker logs -f web</code>	Follow output
<code>docker exec -it web sh</code>	Shell inside a running container
<code>docker inspect web</code>	Ports, env vars, mounts, network, health
<code>docker stats</code>	Live CPU and memory
<code>docker stop web</code> / <code>docker start web</code>	Stop gracefully / start again
<code>docker rm web</code> / <code>docker rm -f web</code>	Remove a stopped / running container

Exit codes in `docker ps -a` : 0 finished normally, 1 app crashed, 127 command not found, 137 killed (usually out of memory).

The Dockerfile [C5 4]

Instruction	Purpose
FROM	Base image; every Dockerfile starts here
WORKDIR	Set and create the working directory
COPY	Copy files from the project into the image
RUN	Run a command at build time, saved as a layer
ENV	Variable at build and run time
ARG	Variable only during the build
EXPOSE	Documents the port; publishes nothing
USER	Run everything after as this user
CMD	Default command; replaced by arguments to <code>docker run</code>
ENTRYPOINT	Fixed program; CMD becomes its default arguments
HEALTHCHECK	Command Docker runs to check app health

```

# Dockerfile : the Django image
FROM python:3.12-slim                                # 1 pinned slim base

ENV PYTHONDONTWRITEBYTECODE=1 PYTHONUNBUFFERED=1    # 2 no .pyc, logs
show at once

WORKDIR /app                                          # 3 later paths
relative to /app

COPY requirements.txt .                               # 4 requirements
first (cache)
RUN pip install --no-cache-dir -r requirements.txt

COPY . .                                             # 5 code last
(changes most)
RUN python manage.py collectstatic --noinput        # 6 admin styled with
DEBUG off

RUN useradd --create-home --uid 10001 appuser \      # 7 non root user
  && chown -R appuser /app
USER appuser

EXPOSE 8000                                           # 8 documents the
port
CMD ["gunicorn", "mysite.wsgi:application", "--bind", "0.0.0.0:8000", "--
workers", "2"]

```

Why each line: 1 pinned slim base, 2 no `.pyc` and live logs in `docker logs`, 3 base path, 4 slow install stays cached, 5 code last, 6 static files gathered, 7 non root, 8 gunicorn serves `wsgi.py` on `0.0.0.0` so it is reachable from outside.

```

# .dockerignore (keeps venv, Git history and passwords out of the image)
.git
venv/
__pycache__/
*.pyc
.env
db.sqlite3
staticfiles/
Dockerfile*
compose*.yaml

```

Do not add `*.pkl`: the trained model must travel inside the image.

Want	Change
A health status in <code>docker ps</code>	Add HEALTHCHECK CMD <code>python -c "import urllib.request; urllib.request.urlopen('http://localhost:8000/admin/login/')"</code>
Fixed program, changeable arguments	<code>ENTRYPOINT ["ping", "-c", "3"]</code> plus CMD <code>["localhost"]</code> ; <code>docker run img google.com</code> replaces only CMD
Build tools out of the final image	Multi stage: <code>FROM python:3.12 AS builder</code> installs, then <code>COPY --from=builder</code> into a slim final stage

Never put secrets in ENV or ARG. `docker inspect` shows them to anyone with the image.

Preparing Django for Docker [C5 5]

```
# requirements.txt (generate the real one with: pip freeze >
requirements.txt)
Django==5.0.6
django-rest-framework==3.15.1
psycopg2-binary==2.9.9
gunicorn==23.0.0
whitenoise==6.7.0
scikit-learn==1.5.0
joblib==1.4.2
```

scikit-learn must match the version that trained `iris_model.pkl`.

```
# mysite/settings.py
import os

SECRET_KEY = os.environ.get("DJANGO_SECRET_KEY", "dev-only-not-secret")
DEBUG = os.environ.get("DJANGO_DEBUG", "0") == "1"
ALLOWED_HOSTS = os.environ.get("DJANGO_ALLOWED_HOSTS",
"localhost,127.0.0.1").split(",")

MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",
    "whitenoise.middleware.WhiteNoiseMiddleware",      # add this line here
    # ... the rest unchanged ...
]
```

```

DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": os.environ.get("POSTGRES_DB", "imapdb"),
        "USER": os.environ.get("POSTGRES_USER", "postgres"),
        "PASSWORD": os.environ.get("POSTGRES_PASSWORD", ""),
        "HOST": os.environ.get("POSTGRES_HOST", "localhost"),
        "PORT": "5432",
    }
}

STATIC_ROOT = BASE_DIR / "staticfiles"

```

The defaults keep `runserver` on your laptop working as in Unit 2. Inside Compose, `.env` and `POSTGRES_HOST=db` override them.

Volumes and networks [C5 6]

Command	Purpose
<code>docker volume create pgdata</code>	Create a named volume
<code>docker volume ls</code> / <code>docker volume inspect pgdata</code>	List / show where one lives
<code>docker volume rm pgdata</code>	Delete a volume and its data
<code>docker network create imap-net</code>	Network with name based DNS
<code>docker network inspect imap-net</code>	Show attached containers

The stack by hand (what Compose automates):

```

docker network create imap-net
docker volume create pgdata

docker run -d --name db --network imap-net \
    -e POSTGRES_PASSWORD=secret -e POSTGRES_DB=imapdb \
    -v pgdata:/var/lib/postgresql/data postgres:16-alpine

docker run -d --name web --network imap-net -p 8000:8000 \
    -e POSTGRES_HOST=db -e POSTGRES_PASSWORD=secret \
    imap-django:1.0

```

From	Reach PostgreSQL at
Your laptop, port published	localhost:5432
Another container, same network	db:5432
A container reaching your laptop	host.docker.internal:5432

Docker Compose [C5 7]

```
# .env (never committed; commit a .env.example with placeholders)
DJANGO_SECRET_KEY=replace-with-a-long-random-string
DJANGO_DEBUG=0
DJANGO_ALLOWED_HOSTS=localhost,127.0.0.1
POSTGRES_USER=postgres
POSTGRES_PASSWORD=change-me-please
POSTGRES_DB=imapdb
```

```
# compose.yaml
services:
  web:
    build: .
    image: imap-django:1.0
    command: sh -c "python manage.py migrate && gunicorn
mysite.wsgi:application --bind 0.0.0.0:8000 --workers 2"
    ports:
      - "8000:8000"
    env_file: .env
    environment:
      POSTGRES_HOST: db
    depends_on:
      db:
        condition: service_healthy
    restart: unless-stopped

  db:
    image: postgres:16-alpine
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    volumes:
      - pgdata:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d
```

```

${POSTGRES_DB}"]
    interval: 5s
    retries: 10

volumes:
    pgdata:

```

Key	Purpose
build / image	Build from the Dockerfile, and name the result
command	Runs migrations, then gunicorn, every time web starts
ports	"HOST:CONTAINER" , always quoted
env_file / environment	Variables from .env / set inline
POSTGRES_HOST: db	The service name, never localhost
depends_on: service_healthy	Web waits until PostgreSQL is actually ready
volumes	Database files live in the named volume pgdata
`\${...}`	Escapes \$ so Compose leaves it for the container's shell

The database publishes no port, so only the web container can reach it, and it will not clash with the Unit 2 `imap-postgres` container.

Compose command	Purpose
<code>docker compose up -d --build</code>	Build and start everything in the background
<code>docker compose ps</code>	Status of each service
<code>docker compose logs -f web</code>	Follow web logs
<code>docker compose exec web python manage.py createsuperuser</code>	Create an admin inside the container
<code>docker compose exec web python manage.py drf_create_token <user></code>	Create an API token
<code>docker compose exec db psql -U postgres -d imapdb</code>	SQL prompt inside the database
<code>docker compose config</code>	Print final YAML with .env filled in

Compose command	Purpose
<code>docker compose down</code>	Stop everything, keep the database
<code>docker compose down -v</code>	Also delete the volume and all data

```

cp .env.example .env                                # PowerShell: copy .env.example
.env
docker compose up -d --build
docker compose exec web python manage.py createsuperuser
# open http://localhost:8000/admin/

curl -X POST http://localhost:8000/api/predict/ \
  -H "Authorization: Token <your-token>" -H "Content-Type: application/json" \
  -d '{"sepal_length": 5.1, "sepal_width": 3.5, "petal_length": 1.4,
    "petal_width": 0.2}'
# {"prediction": 0}

```

After changing code, `docker compose up -d --build` rebuilds only what changed. After changing a model, run `makemigrations` on your laptop, then rebuild; the container applies the migration on start.

Registries and deployment [C58]

```

# on your laptop
docker login
docker tag imap-django:1.0 <your-hub-user>/imap-django:1.0
docker push <your-hub-user>/imap-django:1.0

# on the server: copy compose.yaml and .env, change build: . to
#   image: <your-hub-user>/imap-django:1.0
docker compose up -d

# Apple M series Mac to an Intel or AMD server: build for both
docker buildx build --platform linux/amd64,linux/arm64 \
  -t <your-hub-user>/imap-django:1.0 --push .

```

Tag	Use for
1.0, 1.4.2	Releases; never reuse a version tag
sha-3f2a9c1	Tracing an image to its exact commit
latest	Convenience only, never production

```
# .github/workflows/docker-publish.yml
name: docker-publish
on:
  push:
    branches: [main]

jobs:
  docker:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write
    steps:
      - uses: actions/checkout@v4
      - uses: docker/setup-buildx-action@v3
      - uses: docker/login-action@v3
        with:
          registry: ghcr.io
          username: ${GITHUB_ACTOR}
          password: ${GITHUB_TOKEN}
      - uses: docker/build-push-action@v6
        with:
          context: .
          push: true
          tags: ghcr.io/${GITHUB_REPOSITORY}:latest
```

Every push to main builds the image and pushes it to `ghcr.io/<you>/<repo>`. GitHub supplies the token automatically. Repository names must be lowercase.

Debugging and clean up [C59]

Step	Command	Look for
1	<code>docker compose ps</code> or <code>docker ps -a</code>	Running, exited, restarting; the exit code
2	<code>docker compose logs web</code>	The actual error

Step	Command	Look for
3	<code>docker inspect <c></code>	Ports, env vars, mounts, network
4	<code>docker compose exec web sh</code>	Look around from inside
5	<code>docker compose config</code>	Variables filled in correctly?

Error	Fix
port is already allocated	Change host port ("8001:8000") or stop the other container
connection refused / could not translate host name "db"	Use <code>db</code> , not localhost; check <code>POSTGRES_HOST</code> and that both services are in the same compose file
DisallowedHost	Add the host to <code>DJANGO_ALLOWED_HOSTS</code>
Admin page has no styling	WhiteNoise missing from MIDDLEWARE, or <code>collectstatic</code> did not run
Database is uninitialized and superuser password is not specified	Set <code>POSTGRES_PASSWORD</code> in <code>.env</code>
exec format error	Rebuild with <code>docker buildx build --platform ...</code>
Exited (137)	Out of memory
App unreachable but container running	App bound to 127.0.0.1; bind to 0.0.0.0

Clean up command	Removes
<code>docker system df</code>	Nothing; shows disk use
<code>docker system prune</code>	Stopped containers, unused networks, dangling images, build cache
<code>docker system prune -a --volumes</code>	Everything unused, including images and volumes

Docker cheat sheet: commands by what you want to do

From the Docker CLI cheat sheet. Replace `<image_name>` and `<container_name>` with your own names.

General

Want to	Command
Start the Docker daemon	<code>docker -d</code>
Get help (works on any subcommand)	<code>docker --help</code>
Show system wide information	<code>docker info</code>

Build and manage images

Want to	Command
Build an image from a Dockerfile	<code>docker build -t <image_name> .</code>
Build without the cache	<code>docker build -t <image_name> . --no-cache</code>
List local images	<code>docker images</code>
Delete an image	<code>docker rmi <image_name></code>
Remove all unused images	<code>docker image prune</code>

Run containers

Want to	Command
Create and run with a custom name	<code>docker run --name <container_name> <image_name></code>
Publish a port to the host	<code>docker run -p <host_port>:<container_port> <image_name></code>
Run in the background	<code>docker run -d <image_name></code>
Start or stop an existing container	<code>docker start stop <container_name> (or container id)</code>
Remove a stopped container	<code>docker rm <container_name></code>

Look inside and troubleshoot

Want to	Command
Open a shell in a running container	<code>docker exec -it <container_name> sh</code>

Want to	Command
Follow the logs	<code>docker logs -f <container_name></code>
Inspect a container	<code>docker inspect <container_name></code>
List running containers	<code>docker ps</code>
List all containers (running and stopped)	<code>docker ps --all</code>
View resource usage	<code>docker container stats</code>

Docker Hub

Want to	Command
Log in	<code>docker login -u <username></code>
Publish an image	<code>docker push <username>/<image_name></code>
Search for an image	<code>docker search <image_name></code>
Pull an image	<code>docker pull <image_name></code>

Key definitions from the sheet:

- **Image:** a lightweight, standalone, executable package with everything needed to run an application: code, runtime, system tools, system libraries and settings.
- **Container:** a runtime instance of an image. It always runs the same regardless of the infrastructure, and isolates software from its environment (for example development vs staging).
- **Docker Hub:** a service for finding and sharing container images with your team (hub.docker.com).
- **Docker Desktop** is available for Mac, Linux and Windows.

Practice questions with answers

Cover the answer, try it, then check. All answers come from your five PDFs.

Part A: Short answers (2 to 3 marks)

1. What is the difference between `*args` and `**kwargs`? `*args` collects extra positional arguments into a tuple. `**kwargs` collects extra keyword arguments into a dict. Example: `order_pizza("L", "olive", "corn", cheese=True)` gives toppings `('olive', 'corn')` and extras `{'cheese': True}`.

2. Explain try, except, else and finally. `try` holds risky code. `except` handles a named error. `else` runs only if `try` had no error. `finally` always runs. `raise` throws your own error, for example `raise ValueError("age cannot be negative")`.

3. What do `super()` and `__init__` do? `__init__` is the constructor that sets up instance attributes. `super().__init__(name)` runs the parent's constructor, and `super().speak()` reuses the parent's method inside an overridden one.

4. Name the four OOP pillars with one line each. Encapsulation: data kept inside the class (`_balance`, `__pin`). Abstraction: callers use `deposit()` without seeing checks. Inheritance: `Dog(Animal)`. Polymorphism: same call `pet.speak()` gives different results per class.

5. List the steps of a socket client and a socket server. Client: `socket`, `connect`, `send`, `recv`, `close`. Server: `socket`, `bind`, `listen`, `accept`, `recv/send`, `close`. Sockets carry bytes, so `.encode()` before `send` and `.decode()` after `recv`.

6. What do the file modes `r`, `w`, `a` mean? Why use `with`? `r` read, `w` overwrite, `a` append, `rb`/`wb` binary. `with open(...)` closes the file automatically. JSON: `dumps/loads` work on strings, `dump/load` work on files.

7. Match HTTP methods to intent and success status. GET read 200, POST create 201, PUT replace 200, DELETE remove 204.

8. What is Django MVT? Model is the database layer, View holds the logic and returns a response, Template is the HTML. URLs route a request to a view.

9. Project vs app. Why add an app to `INSTALLED_APPS`? A project is the whole site (`mysite`), an app is one feature (`orders`). Without registering it, Django cannot find the app's models or templates.

10. `makemigrations` vs `migrate`? `makemigrations` writes the migration file from model changes. `migrate` applies it to the database. After changing a model you need both.

11. What does `{% csrf_token %}` do and when is it needed? It protects against cross site request forgery. Every POST form needs it, otherwise Django returns 403 CSRF verification failed. From JavaScript, send the `X-CSRFToken` header.

12. Difference between `{{ }}` and `{% %}` in templates? `{{ }}` prints a value. `{% %}` runs logic such as `if`, `for`, `url`, `extends`, `block`.

13. What does `authenticate()` return, and what does `login()` do? `authenticate` returns the user, or `None` for wrong credentials. `login` creates the session and cookie.

14. `@login_required` vs `@permission_required`? `login_required` sends anonymous users to `LOGIN_URL`. `permission_required("orders.delete_order", raise_exception=True)` returns 403 if the user lacks that permission. Permissions are

add_, change_, delete_, view_ per model.

15. Why use `create_user` instead of `User(...)`? `create_user` hashes the password. With `User(...)` the user cannot log in.

16. How do sessions and cookies work together? The session data is stored on the server (table `django_session`), and the browser holds only a cookie with the session id. Lifetime is set by `SESSION_COOKIE_AGE`.

17. Why avoid string concatenation in raw SQL? It allows SQL injection. Use placeholders: `cursor.execute("... WHERE id = %s", [user_id])`.

18. What is a lazy QuerySet? The database is not hit when you build `Order.objects.filter(...)`. It runs only when the result is used, for example by iterating or `count()`.

19. Which DRF permission class is the default? Name the others. `AllowAny` is the default, so an API with no `permission_classes` is open to everyone. Others: `IsAuthenticated`, `IsAdminUser`, `IsAuthenticatedOrReadOnly`.

20. What do 400, 401, 403 and 404 mean in an API? 400 serializer validation failed. 401 missing or wrong token. 403 logged in but not allowed. 404 object not found.

21. How do you call a token protected API? Send the header `Authorization: Token <key>`. Create the key with `python manage.py drf_create_token <user>`.

22. Why load the ML model once at import and not per request? Loading is slow. `joblib.load` at module level in `ml_model.py` happens once, then every request just calls `model.predict`.

23. Fix: "Expected 2D array" and "int64 is not JSON serializable". Use `model.predict([[a, b, c, d]])` (a list inside a list) and return `{"prediction": int(p)}`.

24. Why does a container start in seconds but a VM takes minutes? A container shares the host kernel and boots no OS, so it starts like a normal process. A VM boots a full OS with its own kernel.

25. What are namespaces and cgroups? Namespaces control what a container can see (processes, network, hostname, filesystem). Cgroups control what it can use (CPU, memory, number of processes).

26. Image vs container. Client vs daemon vs registry. Image is a read only template, container is a running instance of it (like class and object). Client sends commands, daemon does the work, registry stores images (Docker Hub).

27. Why copy `requirements.txt` before the source code in a Dockerfile? A cached layer is reused until something before it changes. Code changes often, dependencies rarely, so the slow `pip install` layer stays cached. Lab: 1.0 s vs 5.4 s.

28. CMD vs ENTRYPOINT, COPY vs ADD, ARG vs ENV. ENTRYPOINT is the program, CMD its default arguments (replaced by text after the image name). COPY only copies, ADD also unpacks archives and downloads URLs. ARG exists at build time only, ENV is baked into the image. Neither is safe for secrets.

29. What does `.dockerignore` do? Keeps `.git`, `venv` and `.env` (passwords) out of the build context so `COPY . .` does not put them in the image. Do not ignore `*.pkl`, the model must be in the image.

30. Why not use `python:latest`? `latest` just means the most recent push, not stable, and can move to a new Python and break the build. Pin major and minor, such as `python:3.12-slim`.

31. Volume vs bind mount vs tmpfs. Volume: Docker managed, for databases and uploads, survives removal. Bind mount: a host folder, for source code in development, survives. tmpfs: memory, temporary, does not survive.

32. What does `-p 8001:8000` mean? Host port 8001 on the left, container port 8000 on the right ("outside : inside").

33. Why must an app in a container bind to `0.0.0.0`? Inside a container `127.0.0.1` means this container only, so outside traffic is refused. `0.0.0.0` accepts it.

34. What is the localhost trap? Inside the web container, `localhost` is the web container itself, with no database. Use the service name, for example `db:5432`.

35. `docker compose down` vs `down -v`? `down` removes containers and network and keeps volumes (data survives). `down -v` also deletes volumes and wipes the database.

36. Started is not ready. How does Compose fix this? A healthcheck tests real readiness (`pg_isready`), and `depends_on` with `condition: service_healthy` makes web wait.

37. Why does the admin look unstyled after moving Django to Docker? With `DEBUG` off Django stops serving static files. Run `collectstatic` and add `WhiteNoise` to `MIDDLEWARE`.

38. Why do migrations run at start up and not in the build? The database does not exist while the image is built. `migrate` runs in the web `command` before `gunicorn`.

39. Why must you never put a secret in `ENV` in a Dockerfile? Anyone can read it with `docker inspect`, and deleting it in a later layer does not remove it from the earlier layer. Supply secrets at run time through `.env`.

40. Give the version, commit, branch and latest tag rules. Version tags (1.0) and commit tags (sha-3f2a9c1) never change. Branch tags and latest change every push. Never reuse a version tag, never run latest in production.

Part B: Longer answers (5 marks)

41. Explain the Django request flow with a small example: model, view, url, template.

1. The browser requests `/orders/`. The project `urls.py` matches `path("orders/", include("orders.urls"))`, then `orders/urls.py` matches `path("", views.order_list, name="order-list")`.
2. The view `order_list` asks the model: `Order.objects.filter(paid=False)`.
3. The view calls `render(request, "orders/list.html", {"orders": orders})`.
4. The template extends `base.html` and loops `{% for o in orders %}` printing `{{ o.item }}`.
5. Django returns the HTML as the response.

42. Write a Django login view and explain each step.

```
def login_view(request):
    if request.method == "POST":
        user = authenticate(request, username=request.POST["username"],
                             password=request.POST["password"])
        if user is not None:
            login(request, user)
            return redirect("dashboard")
        return render(request, "index.html")
```

`authenticate` checks credentials and returns `None` if wrong. `login` creates the session and sets the cookie. `redirect` sends the user to the dashboard. The form needs `{% csrf_token %}`, and protected pages use `@login_required`.

43. Show how to connect Django to PostgreSQL running in Docker.

1. `pip install psycopg2-binary`.
2. `docker run --name imap-postgres -e POSTGRES_PASSWORD=yourpassword -e POSTGRES_DB=imapdb -p 5432:5432 -d postgres:16`.
3. In `settings.py` set `ENGINE` to `django.db.backends.postgresql`, with `NAME` `imapdb`, `USER` `postgres`, `HOST` `localhost`, `PORT` `5432`.
4. Run `migrate` and `createsuperuser`. Next time: `docker start imap-postgres`.

If the error is "connection refused on 5432", the container is not running.

44. Build a DRF API for users with token authentication.

1. `pip install djangorestframework`. Add `rest_framework` and `rest_framework.authtoken` to `INSTALLED_APPS` and set `REST_FRAMEWORK` auth classes (Token and Session).
2. `UserSerializer(ModelSerializer)` with fields `id`, `username`, `email`.
3. `ViewSet(ModelViewSet)` with `permission_classes = [IsAuthenticated]`.
4. `DefaultRouter().register("users", UserViewSet)` and `path("api/", include(router.urls))`.
5. `migrate`, then `drf_create_token asha`.
6. Test: with `-H "Authorization: Token <token>"` returns 200, without it 401.

45. Explain how an ML model is served through Django.

1. Train once: `RandomForestClassifier` on Iris, saved with `joblib.dump` to `iris_model.pkl`.
2. `ml_model.py` loads it once with `joblib.load`.
3. `IrisInputSerializer` has four `FloatField`s, and `is_valid(raise_exception=True)` returns 400 on bad input.
4. `PredictView(APIView)` with `IsAuthenticated` calls `model.predict([[...]])[0]` and returns `Response({"prediction": int(p)})`.
5. Route it at `api/predict/`. Test with curl POST, expected `{"prediction": 0}`.

46. Why was Docker created? Compare containers and virtual machines. Git ships only code, while an app also needs a runtime, libraries, OS packages and config, which drift between machines ("works on my machine", dependency hell). Docker packages all of it into an image and runs it as a container. A VM boots its own OS and kernel, so it is gigabytes and slow to start. A container shares the host kernel, so it is megabytes, starts in about a second, and dozens fit on a laptop. VM isolation is stronger (hardware level) than container isolation (process level).

47. Explain what happens when you type `docker run`.

1. Client asks the daemon to create and start a container.
2. Daemon checks the local image cache.
3. If missing, it pulls the layers from the registry.
4. It stacks the layers and adds a thin writable layer.
5. It gives the container a network interface and publishes ports.
6. Kernel sets up namespaces and cgroups, and the app starts as process 1.

48. Write the Dockerfile for the Django project and justify the order of lines.

```
FROM python:3.12-slim
ENV PYTHONDONTWRITEBYTECODE=1 PYTHONUNBUFFERED=1
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
RUN python manage.py collectstatic --noinput
RUN useradd --create-home --uid 10001 appuser && chown -R appuser /app
USER appuser
EXPOSE 8000
CMD ["gunicorn", "mysite.wsgi:application", "--bind", "0.0.0.0:8000", "--workers", "2"]
```

Pinned slim base keeps the image small and repeatable. Requirements are copied and installed before code so the slow install layer stays cached. Code is last because it changes most. `collectstatic` styles the admin with DEBUG off. A non root user limits damage. gunicorn binds `0.0.0.0` so it is reachable from outside.

49. Write a compose file for Django and PostgreSQL and explain the key parts. Use the `compose.yaml` from the Unit 5 code section. Key points: `web` is built from the Dockerfile and publishes `"8000:8000"`. Its `command` runs `migrate` and then gunicorn. `env_file:` `.env` supplies secrets, and `POSTGRES_HOST: db` uses the service name, never localhost. `depends_on` with `service_healthy` waits for the `db` healthcheck (`pg_isready`). `db` uses `postgres:16-alpine` with the named volume `pgdata`, so data survives `docker compose down`. The database publishes no port, so only `web` can reach it.

50. List the changes needed to run Django in Docker.

1. gunicorn replaces `runserver`.
2. `SECRET_KEY`, `DEBUG`, `ALLOWED_HOSTS` and DB settings come from environment variables, and DB `HOST` becomes `db`.
3. Migrations run at start up, and the superuser is created inside the running container.
4. `collectstatic` plus WhiteNoise serve the static files.
5. The `.pkl` model ships inside the image, with a matching scikit-learn version in `requirements.txt`.

51. Explain the debugging playbook and fix these errors: port already allocated, connection refused to db, Exited (137), exec format error. Order: `docker ps -a` (status and exit code), `docker logs` (the real error), `docker inspect` (ports, env, mounts, networks), `docker exec -it <c> sh` (look inside), then check the network (same network, service name, bound to `0.0.0.0`). Fixes: port already allocated means the host port is taken, so use `"8001:8000"`. Connection refused to db means localhost was used

instead of the service name, or the DB is not ready. Exited (137) is out of memory. Exec format error means a CPU architecture mismatch, so build with `docker buildx build --platform linux/amd64,linux/arm64`.

52. Describe the deployment flow and CI/CD pipeline. Deploy: `docker login`, `docker tag imap-django:1.0 <user>/imap-django:1.0`, `docker push`, then on the server change `build: .` to the `image: name` and run `docker compose up -d`. CI/CD (GitHub Actions on every push to main): build the image with cached layers, test it, scan for vulnerabilities, push to a registry with proper tags, and deploy by having the server pull the new image and restart. Nothing depends on anyone's laptop.

53. List the Docker security best practices. No secrets in images (use `.env` at run time). Run as a non root user. Use a minimal pinned base image. Use `.dockerignore` to keep `.env`, `.git` and keys out. Limit CPU and memory. Scan images and rebuild regularly. Never use `--privileged` and never mount Docker's control socket. Containers share one kernel, so they are isolated, but less strongly than VMs.

Part C: Quick fill in the blank

Question	Answer
Code belongs in ____, data belongs in ____	Images, volumes
<code>docker ps</code> shows ____, <code>docker ps -a</code> shows ____	Running containers, all containers
EXPOSE only ____ a port; ____ actually publishes it	Documents, <code>-p</code>
Docker Compose file name	<code>compose.yaml</code>
Command to start the whole stack in the background and rebuild	<code>docker compose up -d --build</code>
Command to create an admin inside the container	<code>docker compose exec web python manage.py createsuperuser</code>
Production server replacing runserver	gunicorn
Package that serves static files with DEBUG off	WhiteNoise
Exit code 137 means	Killed, usually out of memory
Exit code 127 means	Command not found

Question	Answer
Pre-Compose way to reach a DB from another container on the same network	<code>db:5432</code> (service name)
To reach your laptop from a container	<code>host.docker.internal:5432</code>
Command that removes all unused images, containers, networks, volumes	<code>docker system prune -a --volumes</code>
Command to see each layer and its size	<code>docker history name:tag</code>
Command to follow container logs	<code>docker logs -f <container_name></code>
Command to open a shell inside a running container	<code>docker exec -it <container_name> sh</code>