

IMAP Exam Notes: Unit 2 (Back End) and Unit 5 (Docker)

Part A: theory questions with answers in full sentences. Part B: complete the code with answer key. Part C: traps and one liners.

Part A: Theory Q&A

Unit 2: Back End Development

Q1. Difference between makemigrations and migrate?

makemigrations only writes the model change into a migration file and does not touch the database. migrate then applies the pending migration files to the database. Both commands are needed before a model change reaches the real table.

Q2. Explain the MVT pattern.

MVT stands for Model, View, Template. The model holds the data and lives in models.py, the view holds the logic and lives in views.py, and the template holds the HTML. The file urls.py routes each request to the right view. Django's view is the equivalent of the controller in MVC, and Django's template is the equivalent of the view in MVC.

Q3. Describe the flow of one Django request.

The browser sends a request, and urls.py matches the path to a view. The view runs and may ask a model for data from the database. The view then passes that data to a template, which renders the HTML. The response travels back through the view to the browser. A view may also return a redirect or JSON instead of HTML.

Q4. Project vs app?

A project is the whole site. There is one per site, and it holds the global settings and the root URL table. An app is one feature of the site, such as orders or accounts, and it can be reused across projects. An app only takes effect after it is added to INSTALLED_APPS, so running startapp alone is not enough.

Q5. How does a session work?

When login() is called, Django writes one row into the django_session table and sets a cookie that contains only the session ID. On every later request the browser sends this cookie back, and Django uses it to find the row and identify the user. The password never has to travel again. The session expires after the time set in SESSION_COOKIE_AGE.

Q6. Why use create_user() instead of User()?

create_user() hashes the password before saving it, while User() stores the password without hashing. A user created with User() would have an insecure password and could not log in properly.

Q7. What is CSRF protection?

CSRF protection stops another site from submitting a forged form on behalf of a logged in user. Django ties a one time token to the session, and every POST form must send it back using {% csrf_token %}. From JavaScript, the token is sent in the X-CSRFToken header.

Q8. SQLite vs PostgreSQL?

SQLite is a single file with no setup, but it handles many simultaneous writers poorly, so it is used for learning and prototypes. PostgreSQL runs as a separate server on port 5432 and handles many users well, so it is used in production. To switch, you install psycopg2-binary, change the DATABASES setting and run migrate again. The models, views and templates stay the same.

Q9. What are lazy QuerySets?

A QuerySet is a description of a query and not the rows themselves. The SQL only runs when the data is actually needed, for example when the QuerySet is looped over, counted or displayed. Because of this, filters can be chained step by step without hitting the database each time.

Q10. What is SQL injection and how is it prevented?

SQL injection happens when a user types SQL code into an input field and that code runs on your database. It is prevented by using %s placeholders and passing the values separately, so the driver escapes them safely. The ORM does this automatically for every query it writes.

Q11. What does a DRF serializer do?

A serializer translates between model instances and JSON. For outgoing data it converts a model into JSON. For incoming data it validates the JSON and returns a 400 error with a message if the data is bad. The fields list decides exactly what is exposed, which is why the hashed password never leaks.

Q12. ViewSet vs APIView?

A ModelViewSet provides list, create, retrieve, update and delete from just a queryset and a serializer_class, and a router generates its URLs. An APIView is used for custom logic that is not plain CRUD, such as an ML prediction, and it defines one method for each HTTP verb it answers.

Q13. How does token authentication work in DRF?

Each user is given a token, and the client sends it in the header `Authorization: Token <key>` with every request. To enable it, you add `rest_framework.authtoken` to `INSTALLED_APPS`, which only creates the token table, and you add `TokenAuthentication` to `DEFAULT_AUTHENTICATION_CLASSES`, which makes DRF read the header. A token is created with `python manage.py drf_create_token <username>`.

Q14. Authentication vs permission? Why is the DRF default dangerous?

Authentication only identifies who is making the request, while permissions decide whether that user is allowed in. DRF's default permission is `AllowAny`, so a view without `permission_classes` is open to everyone, even when token authentication is configured. Adding `IsAuthenticated` makes a missing or wrong token return a 401 error.

Q15. Explain the ML integration steps.

First, the model is trained once in a separate script and saved with `joblib.dump`. Second, it is loaded once at module import and not inside the view, so it is not re-read on every request. Third, the input is validated with a plain `Serializer` and `is_valid(raise_exception=True)`. Fourth, `model.predict` is called with a list inside a list, because `scikit-learn` expects a batch of rows. Fifth, the result is returned as `int(p)`, because `NumPy` integers cannot be converted to JSON. Finally, the view is protected with `IsAuthenticated`.

Q16. The four pillars of OOP?

Encapsulation means data and the methods that act on it are bundled in one class. Abstraction means the caller uses a simple interface without knowing the internal details. Inheritance means a child class reuses and extends a parent class. Polymorphism means the same method call behaves differently depending on the object, for example `speak()` returns `Woof` for a `Dog` and `Meow` for a `Cat`.

Q17. `__str__` vs `__repr__`?

`__str__` gives a readable version of the object for users and is what `print()` shows. `__repr__` gives an unambiguous version for developers and debugging. If only `__repr__` is defined, Python uses it wherever `__str__` would be used.

Q18. Class attribute vs instance attribute?

An attribute set with `self.x` inside `__init__` belongs to one instance, so every object has its own copy. An attribute written directly in the class body is shared by all instances unless a specific object overrides it.

Q19. Explain the try, except, else, finally block.

The try block contains the risky code and always runs. The except block runs only if a matching error occurs. The else block runs only if no error occurred, and the finally block always runs. You should catch specific errors and never use a bare except, because it hides unexpected mistakes such as a `NameError`.

Q20. Role of a socket and the client and server calls?

A socket is one endpoint of a connection and is identified by an IP address and a port number. A client calls `socket()`, `connect()`, `send()` and `recv()`, and then `close()`. A server calls `bind()`, `listen()` and `accept()`, where `accept()` returns a new socket for each client. `AF_INET` means a normal IP address and `SOCK_STREAM` means TCP. Sockets carry bytes, so strings are encoded before sending and decoded after receiving. Django's development port is 8000 and the default HTTP port is 80.

Q21. HTTP methods, idempotence and status codes?

GET reads a resource and returns 200, POST creates one and returns 201, PUT replaces one and returns 200, and DELETE removes one and returns 204. GET and PUT are idempotent, which means repeating them has the same effect as doing them once. That is why forms that create something use POST. After a POST, the view should return `redirect()` so that refreshing the page does not submit the form again.

Q22. File modes and JSON functions?

Mode `r` reads and the file must exist, `w` writes and overwrites the file, `a` appends to the end, and `r+` reads and writes without truncating. Adding `b` switches to binary mode. The `with open()` pattern closes the file automatically. In the `json` module, `dumps` and `loads` work on strings, while `dump` and `load` work on files.

Unit 5: Docker

Q23. Why Docker? What problem does it solve?

Git ships only the code, so the runtime, libraries and configuration differ from machine to machine, which causes the 'works on my machine' problem. Docker packages the code together with its runtime, libraries and configuration into an image, so the application runs the same way on a laptop, in CI and in production.

Q24. Container vs virtual machine?

A virtual machine virtualises the hardware and boots its own full operating system with its own kernel, so it takes gigabytes and starts in tens of seconds to minutes. A container shares the host's kernel, so it takes only megabytes and starts in about a second. Containers have process level isolation, which is weaker than the hardware level isolation of a VM, but you can run dozens of them on a laptop.

Q25. What is a container, technically?

A container is an ordinary Linux process that the kernel restricts. Namespaces limit what the process can see, such as its own process list, network and filesystem. Control groups, or cgroups, limit what it can use, such as CPU and memory.

Q26. Image vs container?

An image is a read only template built from a Dockerfile. A container is a running or stopped instance of that image with a thin writable layer on top. This is like a class and an object in Python, because one image can produce many independent containers.

Q27. Docker architecture?

The client is the docker command, and it only sends requests. The daemon is the background service that does the real work, such as building images, running containers and managing storage and networks. The registry, such as Docker Hub, is the server that stores and shares images.

Q28. Why copy requirements.txt before the code?

Docker caches each layer, and when one layer changes it rebuilds that layer and every layer after it. Code changes on almost every build, while dependencies rarely change. By copying and installing requirements.txt first and copying the code last, the slow install layer stays cached and builds are much faster.

Q29. CMD vs ENTRYPOINT? ARG vs ENV? COPY vs ADD?

ENTRYPOINT is the fixed program, and CMD provides its default arguments, so anything typed after the image name in docker run replaces CMD. ARG exists only during the build, while ENV stays in the image and is visible to the app, and neither is safe for secrets. COPY only copies files, while ADD can also unpack archives and download URLs, so COPY is preferred.

Q30. Why avoid the latest tag?

A tag is only a label that the publisher can move to new content at any time, and latest simply means whatever was pushed most recently. A build that uses latest can therefore work today and break later. You should pin at least the major and minor version, such as python:3.12-slim, and never run latest in production.

Q31. Choosing a base image?

The full python:3.12 image is about 1.6 GB and should only be used in build stages that need compilers. The slim image is about 190 MB and is the default choice for Python apps. Alpine is very small, but some Python packages do not install on it. Distless images have no shell or package manager, so they offer very little for an attacker to use.

Q32. What is .dockerignore for?

A .dockerignore file keeps the venv, the .git history, the .env file and keys out of the build context, so passwords and unnecessary files do not end up in the image. You should not ignore *.pkl files, because the trained model must travel inside the image.

Q33. Why do containers use service names, not localhost?

Inside a container, localhost means that container itself, and there is no database there. Containers on the same user defined network reach each other by service name, because Docker runs a small DNS server on that network, so Django connects to db:5432. From your laptop you use localhost:5432 through the published port, and from a container to your laptop you use host.docker.internal.

Q34. -p vs EXPOSE? Why bind to 0.0.0.0?

EXPOSE in a Dockerfile only documents a port, while -p HOST:CONTAINER actually publishes it. The app must bind to 0.0.0.0 because inside a container 127.0.0.1 means this container only, so traffic from outside would be refused.

Q35. Where should code and data live?

Code belongs in images and data belongs in volumes. A container's writable layer is deleted when the container is removed, but a named volume outlives it. Bind mounts share a host folder and suit source code during development, and tmpfs stores temporary files in memory only.

Q36. Docker Compose: what and why?

Docker Compose describes the whole stack, including its services, volumes and networks, in one declarative compose.yaml file, and docker compose up starts everything with one command. The file is version controlled and reproducible, and it replaces many long docker run commands. Each service name also becomes a hostname.

Q37. depends_on vs healthcheck?

A database container can be running while PostgreSQL inside it is still starting, so started does not mean ready. A healthcheck, such as pg_isready, tests whether the database is really ready. Using depends_on with condition: service_healthy makes the web service wait until then.

Q38. down vs down -v?

docker compose down removes the containers and the network but keeps the volumes, so the data survives. docker compose down -v also deletes the volumes, which wipes the database.

Q39. What changes when Django moves into Docker?

Django is served by gunicorn instead of runserver, bound to 0.0.0.0:8000 and pointed at wsgi.py. Settings such as SECRET_KEY, DEBUG, ALLOWED_HOSTS and the database password come from .env, and the database host becomes db. Migrations run when the container starts, not at build time, because the database does not exist during the build. Static files are gathered with collectstatic and served by WhiteNoise. The scikit-learn version in the image must also match the one that trained the model.

Q40. Container security best practices?

Secrets must never be placed in an image, because docker inspect shows ENV and ARG values and deleting them in a later layer does not remove them from the earlier one. The container should run as a non root user, use a minimal and pinned base image, and use .dockerignore. Resources should be limited, and images should be scanned and rebuilt regularly. Containers must never run with --privileged or have Docker's control socket mounted.

Q41. What is CI/CD?

CI/CD automates the path from code to server. On every git push, a pipeline such as GitHub Actions builds the image, tests it, scans it, pushes it to a registry with proper tags, and deploys it. Nothing depends on anyone's laptop.

Q42. Tagging strategy?

Version tags such as 1.0 never change and are used for releases. Git commit tags such as sha-3f2a9c1 trace an image to its exact code. Branch tags and latest change on every push and are only for convenience or testing. You should never push different content under an existing version tag.

Q43. How do you debug a failing container?

First, check docker ps -a to see whether the container is running and what its exit code is. Second, read docker logs, which usually shows the real error. Third, use docker inspect to see the ports, environment variables, mounts and networks. Fourth, use docker exec to look inside the container. Fifth, check the network: the services should be on the same network, use the service name instead of localhost, and bind to 0.0.0.0. An exit code of 0 means normal, 1 means the app crashed, 127 means command not found, and 137 means it was killed, usually from running out of memory.

Q44. Containerise the Unit 2 Django ML app (10 mark chain).

First, create requirements.txt with pinned versions, making sure scikit-learn matches the version that trained the model. Second, write the Dockerfile with a slim base, requirements before code, collectstatic, a non root user, EXPOSE 8000 and a gunicorn CMD. Third, add a .dockerignore that keeps .env out but does not ignore *.pkl. Fourth, make settings.py read environment variables, set the database host to db and add WhiteNoise. Fifth, write compose.yaml with web and db services, an env_file, a pgdata volume, a healthcheck and depends_on with service_healthy. Sixth, run docker compose up -d --build. Seventh, create the superuser and the API token using docker compose exec. Finally, test the endpoint with curl using the Authorization: Token header.

Part B: Complete the Code

Fill each blank (1), (2), ... then check the key in green.

1. Django Dockerfile

```
FROM ____ (1)
WORKDIR /app
COPY ____ (2) .
RUN pip install --no-cache-dir -r ____ (3)
COPY . .
EXPOSE ____ (4)
CMD ["gunicorn", "mysite.wsgi:application", "--bind", "____ (5):8000"]
```

Key: (1) python:3.12-slim (2) requirements.txt (3) requirements.txt (4) 8000 (5) 0.0.0.0

2. compose.yaml

```
services:
  web:
    build: .
    ports:
      - "____ (1)"
    env_file: ____ (2)
    environment:
      POSTGRES_HOST: ____ (3)
    depends_on:
      db:
        condition: ____ (4)
  db:
    image: postgres:16-alpine
    volumes:
      - ____ (5):/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "____ (6) -U ${POSTGRES_USER} -d ${POSTGRES_DB}"]
volumes:
  pgdata:
```

Key: (1) 8000:8000 (2) .env (3) db (4) service_healthy (5) pgdata (6) pg_isready

3. Docker and curl commands

```
docker run -d ____ (1) 8080:80 --name web nginx:alpine
docker compose down ____ (2) # also deletes the volume
curl http://127.0.0.1:8000/api/users/ -H "Authorization: ____ (3) <token>"
docker compose exec web python manage.py ____ (4) <user> # creates API token
```

Key: (1) -p (2) -v (3) Token (4) drf_create_token

4. Model

```
class Order(models.Model):
    customer = models.ForeignKey(User, on_delete=models.____ (1))
    item = models.CharField(____ (2)=100)
    price = models.DecimalField(max_digits=8, decimal_places=2)
    created_at = models.DateTimeField(____ (3)=True)
    def ____ (4)(self):
        return f"{self.item} x {self.quantity}"
```

Key: (1) CASCADE (2) max_length (3) auto_now_add (4) __str__

5. View and URLs

```
def order_detail(request, pk):
    order = ____ (1)(Order, pk=pk)
    return ____ (2)(request, "orders/detail.html", {"order": order})

# orders/urls.py
urlpatterns = [path("____ (3)/", views.order_detail, name="order-detail")]
# project urls.py
urlpatterns = [path("orders/", ____ (4)("orders.urls"))]
```

Key: (1) get_object_or_404 (2) render (3) <int:pk> (4) include

6. Auth

```
user = ____ (1)(request, username=u, password=p)
if user is not ____ (2):
    ____ (3)(request, user)

@____ (4)
def dashboard(request): ...

User.objects.____ (5)(username=u, password=p) # hashes the password
```

Key: (1) authenticate (2) None (3) login (4) login_required (5) create_user

7. DRF serializer, ViewSet, router

```
class UserSerializer(serializers.ModelSerializer):
    class ____ (1):
        model = User
        fields = ["id", "username", "email"]

class UserViewSet(viewsets.____ (2)):
    queryset = User.objects.all()
    serializer_class = ____ (3)

router = DefaultRouter()
router.____ (4)("users", UserViewSet)
urlpatterns += [path("api/", ____ (5)(router.urls))]
```

Key: (1) Meta (2) ModelViewSet (3) UserSerializer (4) register (5) include

8. Settings (DB and token auth) and raw SQL

```
"ENGINE": "django.db.backends.____ (1)"
"PORT": "____ (2)"
INSTALLED_APPS += ["rest_framework", "rest_framework.____ (3)"]
"DEFAULT_AUTHENTICATION_CLASSES": ["rest_framework.authentication.____ (4)", ...]
cursor.execute("SELECT username FROM auth_user WHERE id = ____ (5)", [user_id])
```

Key: (1) postgresql (2) 5432 (3) authtoken (4) TokenAuthentication (5) %s

9. ML: load once and prediction view

```
MODEL_PATH = Path(____ (1)).resolve().parent / "ml" / "iris_model.pkl"
model = joblib.____ (2)(MODEL_PATH) # at module level, not in the view

class PredictView(APIView):
    permission_classes = [____ (3)]
    def post(self, request):
        s = IrisInputSerializer(data=request.data)
        s.is_valid(____ (4)=True)
        d = s.validated_data
        p = model.predict(____ (5))[0]
        return Response({"prediction": ____ (6)(p)})
```

Key: (1) __file__ (2) load (3) IsAuthenticated (4) raise_exception (5) [[d["sepal_length"], d["sepal_width"], d["petal_length"], d["petal_width"]]] (6) int

10. Socket server

```
server = socket.socket(socket.____ (1), socket.____ (2))
server.____ (3)(("127.0.0.1", 9000))
server.____ (4)(5)
conn, addr = server.____ (5)()
data = conn.recv(1024)
conn.send(____ (6) + data)
```

Key: (1) AF_INET (2) SOCK_STREAM (3) bind (4) listen (5) accept (6) b"Echo: "

11. Python core

```
def order_pizza(size, ____ (1), ____ (2)): ...

try:
    risky()
except ____ (3) as e:
    handle(e)
____ (4):
    print("only if no error")
____ (5):
    print("always runs")

class Puppy(Dog):
    def speak(self):
        return ____ (6).speak() + "!"
```

Key: (1) *toppings (2) **extras (3) ValueError (4) else (5) finally (6) super()

Part C: Traps and One Liners

- makemigrations only writes the migration file, and migrate is the step that changes the database. Both are needed.
- Running startapp is not enough, because the app must also be added to INSTALLED_APPS.
- The default DRF permission is AllowAny, so you must set permission_classes = [IsAuthenticated] to protect a view.
- Adding authtoken alone does not enable token authentication. You must also add TokenAuthentication to DEFAULT_AUTHENTICATION_CLASSES.
- DRF's built in token system uses the word Token in the header, not Bearer.
- Only the session ID is stored in the cookie, and the real data lives in the django_session table.
- QuerySets are lazy, and json.dumps and loads work on strings while dump and load work on files.
- Class attributes are shared by all instances, while self.x attributes belong to one instance.
- The ML model is loaded once at import and never inside the view. Remember int(p) and the list inside a list.
- Copy requirements before code so the cache works, and always pin a slim base image instead of using latest.
- -p publishes a port while EXPOSE only documents it. Bind to 0.0.0.0 and use the service name db instead of localhost.
- Code belongs in images and data belongs in volumes. down keeps the data, but down -v deletes it.
- Started does not mean ready, so use a healthcheck together with service_healthy.
- Never put secrets in an image. Keep them in .env, which stays out of Git, and commit a .env.example instead.
- Migrations run when the container starts and not at build time, and *.pkl must not be ignored.