FastAPI Advanced Interview Q&A

1. **What are dependency injections in FastAPI, and why are they powerful?**

A: Dependency injection in FastAPI allows you to define reusable components (DB sessions, authentication, logging, etc.) and inject them into endpoints automatically. It helps keep code clean, testable, and modular.

2. **How do you handle authentication in FastAPI?**

A:

- Basic Auth → fastapi.security.HTTPBasic
- OAuth2 with JWT → OAuth2PasswordBearer, PyJWT
- Custom middlewares for API key / header-based auth.

JWT is most commonly used in production.

3. **Explain Middleware in FastAPI.**

A: Middleware is a function that runs before and after each request.
Use cases: logging, monitoring, request validation, adding custom headers, authentication.

```
@app.middleware("http")
async def add_process_time_header(request, call_next):
    response = await call_next(request)
    response.headers["X-Process-Time"] = "123ms"
    return response
```

4. **How does FastAPI handle background tasks?**

A: With BackgroundTasks dependency.
Example: Sending emails, writing logs asynchronously after returning response.

5. **How to integrate FastAPI with databases?**

A:

- SQLAlchemy (with async support)
- Tortoise ORM
- Gino ORM
- Use Depends(get_db) pattern to inject DB sessions.

6. **Difference between async def and normal def in FastAPI endpoints?**

A:

- async def → non-blocking, uses asyncio, great for I/O bound tasks.
- def → blocking, runs in a threadpool (FastAPI executes it using starlette.concurrency.run_in_threadpool).

7. **How does FastAPI validate request data?**

A: Using Pydantic models.

- Validates type, format (like EmailStr, HttpUrl), constraints.
- Raises 422 Unprocessable Entity if invalid.

8. **How do you implement pagination in FastAPI?**

A: Use query params (skip, limit) and apply them in DB queries.

9. **How to secure FastAPI APIs?**

A:

- HTTPS (via reverse proxy)
- JWT/OAuth2 authentication
- Rate limiting (middleware / API Gateway)
- CORS middleware
- Input validation with Pydantic

10. **Explain WebSockets in FastAPI**.

A: FastAPI supports real-time bi-directional communication using WebSocket.
Use cases: chat apps, live notifications, dashboards.

```
@app.websocket("/ws")
async def websocket_endpoint(ws: WebSocket):
    await ws.accept()
    await ws.send_text("Hello WebSocket")
```

11. **How do you test FastAPI applications?**

A:

- Use TestClient from fastapi.testclient (built on requests).
- Use pytest for async tests.
- Mock DB / external services.

12. **Explain FastAPI performance vs Flask/Django**.

A:

- FastAPI is built on Starlette + Pydantic, async-native → much faster than Flask/Django for I/O-heavy apps.
- Flask/Django = synchronous by default (but can add async support).

13. **How do you handle file uploads in FastAPI?**

A: Using UploadFile with File(...).

- UploadFile stores file in temporary location and provides async read/write.

14. **What is the role of Depends in FastAPI?**

A: Injects dependencies (auth check, DB session, background task, configuration).

15. **How do you scale FastAPI in production?**

A:

- Use Uvicorn with Gunicorn (uvicorn.workers.UvicornWorker).
- Horizontal scaling via Kubernetes/Docker.
- Use caching (Redis) + DB connection pooling.

16. **How do you implement caching in FastAPI?**

A: With Redis (via aioredis, fastapi-cache2).
Example: cache query results or JWT verification.

17. **How to implement role-based access control (RBAC) in FastAPI?**

A:

- Store roles in JWT claims.
- Use dependency Depends(get_current_user) to enforce role checks at endpoint level.

18. **What is the difference between response_model and Pydantic model?**

A:

- response_model → filters and validates output schema.
- Prevents sending extra/unwanted fields.

19. **How do you handle long-running tasks in FastAPI?**

A:

- Use BackgroundTasks (for short tasks).
- For heavy tasks → use Celery with Redis/RabbitMQ or RQ.

20. **What is Starlette in FastAPI?**

A: FastAPI is built on top of Starlette (for ASGI, routing, middleware, WebSockets). Pydantic handles validation, Starlette handles web layer.

21. **What is the ASGI protocol, and how does it relate to FastAPI?**

A: ASGI (Asynchronous Server Gateway Interface) is the successor of WSGI, designed for async apps. FastAPI runs on ASGI (via Uvicorn/Hypercorn) which allows async I/O, WebSockets, and background tasks.

22. **Difference between WSGI and ASGI?**

A:

- WSGI → Synchronous, single request per worker (Flask, Django).
- ASGI → Asynchronous, supports multiple requests concurrently + WebSockets (FastAPI, Starlette).

23. **How do you handle database transactions in FastAPI?**

A:

- Use sessionmaker from SQLAlchemy with Depends(get_db).
- Commit or rollback inside dependency.
- For async DB → use async_sessionmaker.

24. **How do you implement custom exception handling in FastAPI?**

A: Use @app.exception_handler(ExceptionType) to override responses.

```
@app.exception_handler(HTTPException)
async def custom_http_exception_handler(request, exc):
   return JSONResponse(status_code=exc.status_code, content={"detail":
str(exc.detail)})
```

25. **How do you handle request validation beyond Pydantic (e.g., business rules)?**

A:

- Add custom validators in Pydantic models.
- Use @validator decorators.
- Implement dependencies that enforce rules before hitting DB.

26. **How do you structure a large FastAPI project?**

A: Common structure:

```
app/
  ├ main.py
  ├ api/
  |   ├ v1/
  |   |   ├ routes/
  ├ core/   # settings, config
  ├ db/     # database, models
  ├ schemas/ # pydantic models
  ├ services/ # business logic
  ├ tests/
```

27. **How do you implement rate limiting in FastAPI?**

A:

Use Starlette middlewares or external libs like slowapi.

Can also integrate with Redis for distributed rate limiting.

28. **How do you serve static files in FastAPI?**

A: Use StaticFiles from starlette.staticfiles.

```
app.mount("/static", StaticFiles(directory="static"), name="static")
```

29. **How do you configure CORS in FastAPI?**

A:

```
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  # restrict in prod
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

30. **What are response classes in FastAPI?**

A:

- JSONResponse, HTMLResponse, FileResponse, StreamingResponse.
- Choose based on response type.

31. **How do you stream large files in FastAPI?**

A: Use StreamingResponse.

```
def iterfile():
    with open("large.csv", mode="rb") as file:
        yield from file

return StreamingResponse(iterfile(), media_type="text/csv")
```

32. **How do you integrate FastAPI with Celery for background jobs?**

A:

- Setup Celery worker with Redis/RabbitMQ.
- Trigger tasks inside FastAPI endpoint.
- Monitor with Flower.

33. **How do you implement logging in FastAPI?**

A: Use Python's logging module or structured logging (e.g., loguru).
Attach middleware to log requests/responses.

34. **How do you handle API versioning in FastAPI?**

A: Create separate routers:

```
app.include_router(v1_router, prefix="/api/v1")
app.include_router(v2_router, prefix="/api/v2")
```

35. **How do you deploy FastAPI to production?**

A:

- Run via gunicorn -k uvicorn.workers.UvicornWorker.
- Use Nginx/Apache as reverse proxy.
- Deploy in Docker/Kubernetes.

36. **How do you improve FastAPI performance?**

A:

- Use async DB drivers.
- Use caching (Redis).
- Enable gzip compression.
- Scale horizontally with multiple workers.

37. **How do you implement OAuth2 login with Google/Facebook in FastAPI?**

A: Use Authlib or fastapi-users.

- Redirect to provider → get access token → store user info.

38. **How do you document APIs in FastAPI?**

A: Swagger (/docs) and ReDoc (/redoc) auto-generated.
Can add metadata with tags, summary, description.

39. **How do you handle environment-based configuration in FastAPI?**

A:

- Use Pydantic BaseSettings.
- Load .env via python-dotenv.
- Inject settings as dependency.

40. **What are some FastAPI limitations?**

A:

- Relatively new (smaller ecosystem than Django/Flask).
- Async DB support still maturing.
- Requires good understanding of async programming.