College: Sathyama Institute of Science & technology

Name: Nandipati Bhuvaneswari
RegNo: 42110868
email-id: bhuvanareddy226@gmail.cpm
Number: 9515559872

Q1. -> Request lifecycle (fastAPI | Express)

* client sends HTTP request
* Router matches URL + method
* Middleware layer executes
* Dependency Injection resolves DB
* controller. processes logic.
* Response serialization (JSON)
* Response middleware -> It sent back to client

FASTAPI is more structured & type-safe, express is flexible but developer- descipleined.

-> Middleware :-

Middleware executes before/after route handlers.

Examples I've used :-

- JWT auth verification
- Request logging.
- Rate limiting
- CORS handling.

```
@app.middleware("http")
async def log-requests(request, call-next):
    response = await call-next (request)
    return response.
```

→ Authentication (JWT / OAuth2)

- JWT → Stateless, scalable, ideal for robot APIs
- OAuth2 → for human users or third party access.

flow :-

1. Login → generate JWT
2. Store token (client
3. Validate JWT in middleware
4. Inject user/robot identity into request context.

→ Async vs Sync.

Async ↓ It handles I/O (DB, API, MQTT) efficiently.

Sync ↓ It blocks threads ~~for~~ bad for high concurrency.

* fastAPI async → ideal for robot telemetry & logs ingestion.

* Node js async → event-loop based (non-blocking.

→ Scalable Backend folder structure:-

app/
- api/routes
- core (config, security)
- models
- schemas
- services
- db
- main.py

* It seperates business logic, transport & data.

→ Idempotency (POST / PUT)

- use idempotency keys
- PUT is naturally idempotent
- store request hash + key to prevent duplicates

⇒ crucial for robots retrying on network failure

→ Rate Limiting:

→ Prevents abuse & overload.

→ It is Applied at:

- Auth endpoints
- Telemetry ingestion
- Public APIs

Tools : Redis + token bucket / API gateways

Q2. REST API - Robot Management (Design-level)

Endpoints.

POST /robots
PUT /robots/{id}/status
GET /robots
GET /robots/{id}
POST /robots/{id}/logs
GET / robots/ {id} / logs.

key design decisions :-

- UUID for robot_id
- Logs stored append-only
- status updates partially PATCHable
- clear HTTP codes (201, 404, 409)

Error Handling

```
{
  "error": "Robot not found",
   "code": 404
}
```

Bonus Enhancements :-

- JWT auth for robots
- SQLite for prototype → postgreSQL later.
- Dockerized deployment

Q3 State Vs props (React)

- props → read-only, passed from parent; configuration
- state → mutable, internal component data, behavior

Examples when I use:

→ Robotcard receives robot as props

→ online/offline toggles stored in state.

Q4 Component lifecycle :-

→ when a component mounts, it renders on screen.

→ API calls are usually made inside useEffect.

→ when the component updates, it re-renders.

→ on unmount, we clean up things like webSocket connections.

Q5. Live Robot Dashboard :-

→ for live data, I'd use websockets or MQTT from the backend.

→ The frontend listens to updates and refreshes the robot position & logs in real time.

→ If data stops coming, the robot is marked offline.

Q6. Tailwind CSS

Pros:-
- It is very fast to build UI
- No CSS conflicts
- Consistent design

Cons:-
- Long class names
- Slight learning curve.

Q7. Pagination & Caching :-

→ Pagination reduces the amount of data fetched.

→ Caching using tools like Redis or React Query avoids repeated API calls and improves performance.

Q8. CORS :-

→ CORS controls which domains can access the backend.

→ I allow only trusted origins & avoid using wildcard origins in production.

PART-B - Devops:

Q9. Docker & CI|CD

A docker image is a blueprint of the app, while a container is a running instance of that image.

In CI/CD, I usually:

→ Run tests
→ Build the application
→ Create a Docker image
→ Deploy automatically

Q10. Cloud Deployment (AWS):-

I deploy the app on EC2 using Docker. Nginx acts as reverse proxy, and a load balancer helps distribute traffic across multiple instances
→ Horizontal scaling via ASG.

Q11. Logging & Monitoring:

→ I log all robot events with robot IPs.
→ Logs are centralized so errors and performance issues are easy to track.
→ Alerts notify the team when something goes wrong.

PART C- ROS.

Q12. ROS2 concepts
Nodes handle computation, topics stream data,
and services support request - response
communication. I use a bridge node to send
ROS2 data to backend APIs.
→ Robotics → Backend via bridge Node.

Q13. Vision Results to Backend.
→ MQTT works best for real time vision data.
→ webhooks are event based and polling
are alternatives but less efficient.
In this case, MQTT preferred.

Q14. Vision Data schema:-
I store camera ID, timestamp, inspection result,
confidence score, and bounding box details.
In this way I design a schema for
vision inspection results.

Q15. Message Queues:-
→ RabbitMQ is good for reliable task execution.
→ MQTT is ideal for robots and IOT systems
→ kafka works well for high-volume analytics
data.

PART D - SYSTEM DESIGN.

Q16. Multi - Cameras Inspection System.

→ Cameras publish inspection results to the backend.

→ data is stored in a database and desplayed on a dashboard.

→ failure results trigger alerts automatically.

Q17. Live AMR Dashboard.

→ ROS 2 data is sent via MQTT to the backend

→ The backend streams it to the UI using web-sockets

→ Location history is stored for analysis.

Q18. white Label API.

I wrap the vendor API behind my own API layer.

This lets me control authentication, error handling, logging and data format.

Q19. Easy level :-

```
def sort_tasks (tasks):
    return sorted (tasks, key = lambda x: x['priority'])
```