Kolek-Ta Capstone Panel Questions & Answers

Project: Kolek-Ta - Waste Collection Management System **Location:** Mati City, Davao Oriental **Technology Stack:** Node.js, Express, MongoDB, Leaflet.js, Tailwind CSS

Table of Contents

1. Technical Architecture

> **Q1: Why did you choose Node.js and Express over other frameworks like Django or Laravel?**

Answer:

Non-blocking I/O: Essential for real-time GPS tracking with multiple concurrent connections

JavaScript Full-Stack: Same language for frontend and backend reduces context switching

Large Ecosystem: npm provides packages for JWT, MongoDB, file uploads, etc.

JSON Native: Natural fit for REST APIs and MongoDB document structure

Real-time Capable: Easy integration with WebSockets for future enhancements

Q2: Why MongoDB instead of MySQL/PostgreSQL?

Answer:

Flexible Schema: Route data varies (different number of stops, photos, notes)

GeoJSON Support: Native geospatial queries with 2dsphere indexing for location-based features

Document Structure: Matches our JSON API response format naturally

Horizontal Scaling: Easier to scale with sharding for future growth

MongoDB Atlas: Free tier perfect for capstone project budget

Q3: Explain your dual-mode architecture (Mock Auth vs MongoDB). Why implement both?

Answer:

``

`USE_MOCK_AUTH=true → JSON file storage (development)`

`USE_MOCK_AUTH=false → MongoDB Atlas (production)`

` `Benefits:`

`Development without database setup`

`Easy local testing and demos`

`Vercel deployment compatibility`

`Seamless mode switching via environment variable`

`Fallback if database is unavailable`

Q4: How does your JWT authentication work? Why 24-hour expiry?

`Answer: Flow:`

- `User submits credentials (username, password, role)`
- `Server validates against database (bcrypt comparison)`
- `Server generates JWT with payload:` { userId, role, username }

- `Token stored in browser's localStorage`
- `All API requests include `Authorization: Bearer
- `Middleware validates token on protected routes`

`Why 24 hours:`

`Drivers work daily shifts`

`Balances security with convenience`

`No need to re-login during work day`

`Forces daily re-authentication for security`

---

2. Real-Time Features

> **Q5: How does your GPS tracking system work? What's the update interval?**

Answer: Technical Implementation: ` javascript

```
// Driver's browser sends location every 5 seconds
navigator.geolocation.watchPosition(position => {
fetch('/api/tracking/update', {
method: 'POST',
body: JSON.stringify({
lat: position.coords.latitude,
lng: position.coords.longitude,
speed: position.coords.speed,
heading: position.coords.heading
})
});
}, error => {}, { enableHighAccuracy: true });
```

` **Data Tracked:**

Latitude/Longitude

Speed (km/h)

Heading (degrees)

Timestamp

Associated route ID

Storage: LiveLocation collection with 10-minute TTL index

> **Q6: How do you handle multiple simultaneous GPS updates from different drivers?**

Answer:

Each driver has unique entry keyed by username

MongoDB handles concurrent writes atomically

TTL index auto-removes stale locations (10 minutes)

Admin dashboard polls /api/tracking/active every 5-10 seconds

No conflicts - each driver updates only their own record

Q7: Why did you implement rate limiting for the Nominatim geocoding API?

Answer: Problem: Nominatim (OpenStreetMap) requires maximum 1 request per second. Multiple trucks updating simultaneously caused:

ERR_HTTP2_SERVER_REFUSED_STREAM

ERR_CONNECTION_RESET

Solution: ` javascript

```
// Queue-based rate limiting
const GEOCODE_DELAY = 1100; // 1.1 seconds between requests
let geocodeQueue = [];
async function processGeocodeQueue() {
while (geocodeQueue.length > 0) {
const request = geocodeQueue.shift();
await fetchLocationName(request);
await delay(GEOCODE_DELAY);
}
}
```

` **Additional Optimizations:**

Cache results by coordinates (4 decimal precision)

Use cached values for popup display

Only geocode on first request, reuse thereafter

---

3. Fuel Management System

> **Q8: Explain your fuel consumption estimation algorithm.**

Answer: Formula: `

Total Fuel = Distance Consumption + Stop Consumption + Idle Consumption

Where:

Distance Consumption = (distance / 100) × baseRate × speedFactor × loadFactor

Stop Consumption = numberOfStops × 0.05 liters

Idle Consumption = (idleMinutes / 60) × 2.5 liters/hour

` **Speed Factors:**

Speed Range	Factor	Reason
< 30 km/h	1.3x	Urban collection, frequent stops
30-50 km/h	1.1x	Mixed traffic
50-70 km/h	1.0x	Optimal efficiency
70-90 km/h	1.15x	Highway, higher consumption
> 90 km/h	1.3x	Very high speed inefficiency

Load Factor: 0.85 + (loadPercentage / 100) × 0.4

> **Q9: How accurate is your fuel estimation? How did you derive the speed factors?**

Answer: Derivation Sources:

Industry standards for garbage truck consumption (25-30 L/100km base)

EPA fuel economy data for commercial vehicles

Research papers on urban vs highway driving efficiency

Consultation with local truck operators

Accuracy Considerations:

±10-15% variance expected

More accurate with consistent driving patterns

GPS distance is accurate (Haversine formula)

Speed averaging smooths out anomalies

`Improvements for Production:`

`Calibrate with actual fuel receipts`

`Machine learning model with historical data`

`OBD-II integration for real fuel flow data`

> **Q10: How does automatic fuel deduction work when a route is completed?**

`Answer: Flow:` `

- `Driver clicks "Mark Route Complete"`
- `System fetches trip data from GPS tracking:`
  - `Total distance traveled`
  - `Average speed`
  - `Number of stops`
  - `Idle time`
- `Algorithm calculates fuel consumed`
- `System updates truck record:`
  - `Deducts fuel from fuelLevel percentage`
  - `Adds to totalFuelConsumed`
  - `Updates mileage`
- `Logs consumption to fuel history`
- `Displays summary to admin in Completion History`

` `Code Location:` routes/completions.js `lines 70-122`

---

4. Security Implementation

Q11: How do you secure user passwords?

Answer: Implementation: ` javascript

```
// Registration - Hash password before saving
userSchema.pre('save', async function(next) {
if (!this.isModified('password')) return next();
this.password = await bcrypt.hash(this.password, 10);
next();
});
// Login - Compare password
userSchema.methods.comparePassword = async function(candidatePassword) {
return await bcrypt.compare(candidatePassword, this.password);
};
```

` **Security Features:**

bcryptjs with 10 salt rounds

Passwords never stored in plain text

Salt is unique per password

Computationally expensive to brute force

Q12: What prevents unauthorized access to admin features?

Answer: Middleware Protection: ` javascript

```
function authenticateToken(req, res, next) {
const token = req.headers['authorization']?.split(' ')[1];
if (!token) return res.status(401).json({ error: 'No token' });
jwt.verify(token, JWT_SECRET, (err, user) => {
if (err) return res.status(403).json({ error: 'Invalid token' });
req.user = user;
next();
});
}

// Admin-only route example

router.get('/admin-data', authenticateToken, (req, res) => {

if (req.user.role !== 'admin') {

return res.status(403).json({ error: 'Admin access required' });

}

// ... return admin data

});
```

` **Frontend Protection:**

Role checked on login, UI adjusted accordingly

Admin menu items hidden for drivers

API validates role on every request

Q13: How do you handle file upload security?

Answer: Multer Configuration: ` javascript

```
const upload = multer({

storage: multer.memoryStorage(),

limits: { fileSize: 5 1024 1024 }, // 5MB max

fileFilter: function(req, file, cb) {

const allowedTypes = /jpeg|jpg|png|gif/;

const extname = allowedTypes.test(file.originalname.toLowerCase());

const mimetype = allowedTypes.test(file.mimetype);

if (mimetype && extname) {

return cb(null, true);

}

cb(new Error('Only image files allowed!'));

}

});
```

` **Security Measures:**

File type validation (JPEG, PNG, GIF only)

MIME type checking

5MB size limit per file

Maximum 10 files per upload

Stored as base64 in MongoDB (no file system access)

Q14: What about SQL injection or XSS attacks?

Answer: SQL Injection:

Not applicable - MongoDB uses BSON, not SQL

Mongoose schemas validate data types

No raw query string concatenation

XSS Prevention:

User input displayed via textContent, not innerHTML

Mongoose sanitizes special characters

Content-Security-Policy headers (can be added)

Example Safe Display: ` javascript

```
// Safe - uses textContent

element.textContent = userInput;

// We avoid this pattern with user data

element.innerHTML = userInput; // Dangerous!
```

`

---

5. Database Design

> **Q15: Explain your database schema for Routes.**

Answer: `` `javascript ``

```javascript
const routeSchema = new mongoose.Schema({
routeId: { type: String, required: true, unique: true },
name: String,
// Assignment
assignedDriver: String, // Username reference
assignedVehicle: String, // TruckId reference
// Status
status: {
type: String,
enum: ['planned', 'active', 'completed'],
default: 'planned'
},
// Geographic Data
startLocation: { lat: Number, lng: Number, address: String },
endLocation: { lat: Number, lng: Number, address: String },
distance: Number, // Estimated km
estimatedTime: Number, // Minutes
// Completion Data
completedAt: Date,
completedBy: String,
completionNotes: String,
completionPhotos: [String], // Base64 data URLs
notificationSent: Boolean,
// Auto-calculated Trip Stats
tripStats: {
distanceTraveled: Number, // Actual GPS distance
fuelConsumed: Number, // Calculated liters
stopsCompleted: Number,
averageSpeed: Number
}
}, { timestamps: true });
// Indexes for performance
routeSchema.index({ assignedDriver: 1 });
routeSchema.index({ status: 1 });
```

`

Q16: Why use TTL index for LiveLocation?

Answer: Implementation: ` javascript

```
const liveLocationSchema = new mongoose.Schema({
username: { type: String, required: true, unique: true },
lat: Number,
lng: Number,
speed: Number,
heading: Number,
routeId: String,
lastUpdate: {
type: Date,
default: Date.now,
expires: 600 // TTL: 600 seconds = 10 minutes
}
});
```

` **Benefits:**

Automatic Cleanup: MongoDB removes documents 10 minutes after lastUpdate

No Manual Maintenance: No cron jobs or scheduled tasks needed

Storage Efficiency: Only active locations kept in database

Fresh Data: Stale locations automatically removed

Simple Query: find({}) returns only active drivers

Q17: How do you handle relationships between Trucks, Routes, and Drivers?

Answer: Approach: String references instead of ObjectId refs `

```
User (Driver)

└── username: "driver1"

│

├── Truck.assignedDriver = "driver1"

│

└── Route.assignedDriver = "driver1"

Route.assignedVehicle = "TRUCK-001"

│

└── Truck.truckId = "TRUCK-001"
```

` **Why String References:**

Simpler queries without populate()

Works in both JSON and MongoDB modes

Flexible - driver/truck can be reassigned easily

No foreign key constraints to manage

Trade-offs:

No automatic referential integrity

Must handle orphaned references in application logic

---

6. Deployment & DevOps

Q18: Why did you choose Vercel for deployment?

Answer:

Feature	Benefit
Free Tier	Perfect for capstone budget
GitHub Integration	Auto-deploy on push
Serverless	No server management
HTTPS	Free SSL certificates
CDN	Fast global distribution
Environment Variables	Secure secrets management
Preview Deployments	Test before production

Limitations:

30-second function timeout

Cold starts on infrequent access

No persistent file storage (solved with base64 in MongoDB)

Q19: How do you manage environment variables between development and production?

Answer: Development (.env): `

```
PORT=3004
USE_MOCK_AUTH=false
JWT_SECRET=local-dev-secret
MONGODB_URI=mongodb+srv://...
NODE_ENV=development
```

` **Production (Vercel Dashboard):** `

```
USE_MOCK_AUTH=false
JWT_SECRET=production-secret-key
MONGODB_URI=mongodb+srv://...
```

NODE_ENV=production

` **Key Points:**

.env files in .gitignore - never committed

Vercel variables set in dashboard

USE_MOCK_AUTH switches between modes

Different secrets for dev/prod

Q20: What's the 30-second maxDuration in vercel.json for?

Answer: ` json

```
{
"builds": [{
"src": "server.js",
"use": "@vercel/node",
"config": { "maxDuration": 30 }
}]
}
```

` **Purpose:**

Serverless function timeout limit

Prevents hanging requests from consuming resources

Default is 10 seconds, extended to 30 for:

MongoDB connection time

Photo upload processing

Complex queries

If exceeded: Request returns 504 Gateway Timeout

---

7. User Experience

> **Q21: How does a driver complete a route? Walk us through the flow.**

Answer: `

- LOGIN

Driver enters credentials → JWT issued → Dashboard loads

- VIEW ASSIGNMENT

Dashboard shows assigned truck and route

Map displays route path and collection points

- START ROUTE

Driver clicks "Start Route"

GPS tracking activates (5-second updates)

System tracks: distance, speed, stops

- PERFORM COLLECTION

Driver follows route on map

System records movement in real-time

Admin can see live location

- COMPLETE ROUTE

Driver clicks "Mark as Complete"

Modal appears with:

  - Auto-calculated trip summary (distance, fuel, stops)
  - Photo upload (1-10 required)
  - Notes field (optional)

- SUBMIT

Photos uploaded as base64

Trip stats saved to route

Fuel auto-deducted from truck

Admin receives notification

- CONFIRMATION

Success message with trip summary

Route status changes to "completed"

GPS tracking stops

`

Q22: Why require photo uploads for route completion?

Answer: Accountability Benefits:

Proof of Service: Visual evidence that collection occurred

Fraud Prevention: Cannot falsely claim completion

Quality Assurance: Admin can verify proper collection

Audit Trail: Historical record for disputes

Transparency: Builds trust with municipality

Technical Implementation:

Minimum 1 photo, maximum 10

5MB per photo limit

JPEG/PNG/GIF formats

Stored as base64 in MongoDB

Lazy-loaded in admin view for performance

Q23: How does the admin get notified of completed routes?

Answer: Notification System: `

- Route completed → notificationSent = false
- Admin dashboard shows notification badge
    - Badge count updates every 30 seconds
    - Shows number of pending completions
- Admin clicks notification or "Completion History"
    - Sees list of completed routes
    - "New" badge on unacknowledged items
- Admin views completion details:
    - Driver name
    - Completion time
    - Trip stats (distance, fuel, stops, speed)
    - Photos (click to view)
    - Notes

- `Admin acknowledges → notificationSent = true`
    - `Badge count decreases`
    - `Item marked as "Acknowledged"`

`

8. Challenges & Solutions

Q24: What was the most challenging part of this project?

Answer: Top 3 Challenges:

- **Real-time GPS Tracking**
  - Challenge: Accurate distance calculation, battery drain, indoor accuracy
  - Solution: Haversine formula, 5-second intervals (balance), high accuracy mode
- **MongoDB Connection Timeouts**
  - Challenge: "Operation buffering timed out after 10000ms"
  - Solution: Increased timeouts to 30-60 seconds, connection pooling, retry logic
- **Rate Limiting External APIs**
  - Challenge: Nominatim 1 req/sec limit caused errors with multiple trucks
  - Solution: Queue system, caching, 1.1-second delays between requests

Lessons Learned:

Always handle external API rate limits

Database connections need generous timeouts

Real-time features require careful resource management

Q25: How did you solve the Nominatim API rate limiting issue?

Answer: Problem: `

// Multiple trucks updating simultaneously

Truck1 → geocode request

Truck2 → geocode request ← Rate limited!

Truck3 → geocode request ← Rate limited!

Error: ERR_HTTP2_SERVER_REFUSED_STREAM

` **Solution - Queue System:** ` javascript

```
let geocodeQueue = [];
let isProcessing = false;
const DELAY = 1100; // 1.1 seconds
function queueGeocode(lat, lng, callback) {

const cacheKey = ${lat.toFixed(4)},${lng.toFixed(4)} ;

// Check cache first
if (locationCache[cacheKey]) {
callback(locationCache[cacheKey]);
return;
}
// Add to queue
geocodeQueue.push({ lat, lng, cacheKey, callback });
processQueue();
}
async function processQueue() {
if (isProcessing || geocodeQueue.length === 0) return;
isProcessing = true;
while (geocodeQueue.length > 0) {
const request = geocodeQueue.shift();
const result = await fetchGeocode(request.lat, request.lng);
locationCache[request.cacheKey] = result;
request.callback(result);
if (geocodeQueue.length > 0) {
await delay(DELAY);
}
}
isProcessing = false;
}
`
```

Q26: How do you handle offline scenarios?

Answer: Current Implementation (Partial):

User data cached in localStorage

JWT token persists across sessions

Last known state preserved

Limitations:

GPS updates require connectivity

Route completion needs upload capability

Real-time tracking pauses offline

Future Enhancement Plan: ` javascript

```
// Service Worker for offline support
self.addEventListener('fetch', event => {
if (!navigator.onLine) {
// Queue requests for later sync
offlineQueue.push(event.request.clone());
}
});
// Background sync when online
self.addEventListener('sync', event => {
if (event.tag === 'sync-tracking') {
event.waitUntil(syncOfflineData());
}
});
```

`

9. Scalability

Q27: Can this system handle 100+ trucks and drivers?

Answer: Current Capacity:

Designed for ~10-20 trucks (Mati City scale)

MongoDB Atlas free tier: 512MB storage

Scaling to 100+ Trucks:

Component	Current	Scaled Solution
Database	Atlas Free	Atlas M10+ cluster
GPS Updates	HTTP Polling	WebSocket connections
Geocoding	Nominatim	Self-hosted or paid API
Caching	In-memory	Redis cluster
File Storage	Base64 in DB	AWS S3/CloudFlare R2
Server	Vercel Serverless	Dedicated server/K8s

Code Changes Needed:

WebSocket implementation for real-time

Database connection pooling optimization

Horizontal scaling with load balancer

CDN for static assets

Q28: What would you need to change for city-wide deployment?

Answer: Infrastructure:

- **Dedicated OSRM Server**
    - Self-hosted routing engine
    - No rate limits
    - Local road network data
- **Geocoding Service**
    - Paid API (Google Maps, Mapbox)
    - Or self-hosted Nominatim
- **Database Scaling**

    - MongoDB Atlas dedicated cluster
    - Read replicas for reporting
    - Sharding by region
- **Real-time Architecture**
    - WebSocket server (Socket.io)
    - Redis for pub/sub
    - Message queue for async tasks

Features to Add:

Multi-zone management

Shift scheduling

Maintenance tracking

Citizen reporting integration

Analytics dashboard

---

10. Business Value

Q29: What problem does Kolek-Ta solve for Mati City?

Answer: Problems Addressed:

Problem	Current State	Kolek-Ta Solution
Fleet Visibility	Paper logs, phone calls	Real-time GPS tracking
Accountability	Trust-based completion	Photo proof required
Fuel Costs	Manual estimation	Automatic tracking
Route Efficiency	Fixed routes	Data for optimization
Record Keeping	Paper documents	Digital database
Communication	Radio/phone	In-app notifications

Stakeholder Benefits:

Administrators: Real-time oversight, data-driven decisions

Drivers: Clear assignments, digital completion

Municipality: Cost savings, better service

Citizens: Reliable collection schedules

Q30: How does this system save money for the municipality?

Answer: Cost Savings:

- **Fuel Efficiency (15-20% savings)**
  - Track actual consumption vs estimates
  - Identify inefficient routes
  - Monitor driver behavior (speeding, idling)
- **Administrative Efficiency**
  - Digital records eliminate paper
  - Automated notifications reduce phone calls
  - One dashboard replaces multiple systems
- **Fraud Prevention**
  - Photo proof prevents false completions
  - GPS verifies actual routes taken
  - Fuel tracking catches discrepancies
- **Maintenance Planning**
  - Mileage tracking per truck
  - Predictive maintenance scheduling
  - Reduce breakdown costs

Example Calculation: `

10 trucks × 100 km/day × 30 L/100km = 300 L/day

300 L × ₱65/L = ₱19,500/day fuel cost

15% savings = ₱2,925/day = ₱87,750/month

`

Q31: What's the ROI potential?

Answer: Investment:

Development: Student project (no cost)

Hosting: Vercel free tier + MongoDB Atlas free

Maintenance: Minimal ongoing cost

Returns (Annual Estimates):

| Benefit | Estimated Savings |

|---------|------------------|

Fuel Optimization	₱500,000 - 1,000,000
Administrative Time	₱200,000
Fraud Prevention	₱100,000
Maintenance Prediction	₱150,000
Total	**₱950,000 - 1,450,000**

Intangible Benefits:

Improved public trust

Better service reliability

Data for future planning

Environmental reporting compliance

11. Future Improvements

Q32: What features would you add with more time?

Answer: Priority 1 (High Impact): □ Mobile app (React Native/Flutter) □ WebSocket real-time updates □ SMS notifications for drivers □ Analytics dashboard with charts **Priority 2 (Medium Impact):** □ IoT bin fill-level sensors □ Route optimization algorithm improvement □ Predictive maintenance alerts □ Driver performance scoring **Priority 3 (Nice to Have):** □ Citizen complaint integration □ Weather-based scheduling □ Multi-language support □ Voice commands for drivers

Q33: How would you implement bin fill-level sensors?

Answer: Hardware:

Ultrasonic distance sensors in bin lids

LoRaWAN or NB-IoT connectivity

Solar-powered battery

Software Integration: ` javascript

```
// API endpoint for sensor data
router.post('/api/bins/:binId/level', async (req, res) => {
const { fillPercentage, batteryLevel } = req.body;
await Bin.updateOne(
{ binId: req.params.binId },
{
currentLevel: fillPercentage,
lastReading: new Date(),
sensorBattery: batteryLevel
}
);
// Trigger alert if bin is full
if (fillPercentage > 80) {
await notifyAdminFullBin(req.params.binId);
}

res.json({ success: true });

});
```

` **Route Optimization:**

Prioritize full bins

Skip empty bins

Dynamic route adjustment

Q34: Could this integrate with existing city systems?

Answer: Integration Points:

- **Accounting/Finance System**

`

Export: Fuel costs, mileage reports

Format: CSV, Excel, API

`

- **HR/Payroll System**

`

Export: Driver attendance, routes completed

Integration: API webhook on route completion

`

- **Citizen Portal**

`

Import: Collection complaints

Export: Service status updates

`

- **GIS/Mapping System**

`

Export: Route geometries (GeoJSON)

Import: Updated road networks

`

API Design:

RESTful endpoints for all data

JWT authentication for system-to-system

Webhook support for real-time events

Export endpoints for batch data

---

12. Testing & Quality

> **Q35: How did you test your application?**

Answer: Testing Approach:

- **Unit Tests (Jest)**

```
tests/unit/
├── auth.test.js - Authentication logic
├── fuel.test.js - Fuel calculation algorithm
├── routeOptimizer.test.js - Route optimization
└── tripData.test.js - GPS data processing
```

- **End-to-End Tests (Playwright)**

```
tests/
├── auth.spec.js - Login/logout flows
├── api.spec.js - API endpoint testing
├── mobile.spec.js - Mobile responsiveness
└── accessibility.spec.js - WCAG compliance
```

- **Manual Testing**
  - Cross-browser (Chrome, Firefox, Safari)

- Mobile devices (Android, iOS)
- Different screen sizes
- GPS accuracy testing (outdoor)

Q36: What's your test coverage?

Answer: Covered Areas:

Authentication flows (login, logout, token refresh)

API endpoints (CRUD operations)

Fuel calculation algorithm

Route optimization functions

Form validation

Error handling

Test Commands: ` bash

npm test # Run Jest unit tests

npm run test:e2e # Run Playwright E2E tests

` **Future Improvements:**

Increase coverage to 80%+

Add integration tests

Implement CI/CD pipeline

Add load testing

---

13. Code Quality

Q37: Why vanilla JavaScript instead of React/Vue?

Answer: Reasons:

- **Simpler Deployment:** No build step required
- **Faster Initial Load:** No framework overhead
- **Learning Curve:** Team familiarity with vanilla JS
- **Sufficient for Scope:** Project doesn't need complex state management
- **Direct DOM Control:** Better for map integrations

Trade-offs:

Larger app.js file (10,000+ lines)

Manual DOM manipulation

No component reusability (could refactor)

If Starting Over:

Would consider React for:

Better code organization

Component reusability

State management

Testing ecosystem

Q38: How do you organize 10,000+ lines in app.js?

Answer: Current Organization: ` javascript

```
// ============================================

// SECTION: PAGE LOADING

// ============================================

// ============================================

// SECTION: TOAST NOTIFICATIONS

// ============================================

// ============================================

// SECTION: USER MANAGEMENT

// ============================================
```

// ... etc

` **Sections:**

- Page Loading & Utilities
- Toast/Modal Notifications
- Authentication
- Dashboard
- User Management
- Truck Management
- Route Management
- GPS Tracking
- Fuel Management
- Completion History

Future Refactoring: `

```
public/
├── js/
│ ├── app.js - Main initialization
│ ├── auth.js - Authentication
│ ├── dashboard.js - Dashboard functions
│ ├── trucks.js - Truck management
│ ├── routes.js - Route management
│ ├── tracking.js - GPS tracking
│ └── utils.js - Shared utilities
```

`

14. Specific Technical Questions

> ### Q39: How does the Haversine formula work for distance calculation?

Answer: **Purpose:** Calculate great-circle distance between two GPS coordinates **Formula:** `

```
a = sin²(Δlat/2) + cos(lat1) × cos(lat2) × sin²(Δlng/2)

c = 2 × atan2(√a, √(1-a))

distance = R × c

Where R = 6371 km (Earth's radius)
```

` **Implementation:** ` javascript

```javascript
function haversineDistance(lat1, lng1, lat2, lng2) {

const R = 6371; // Earth's radius in km

const dLat = (lat2 - lat1) * Math.PI / 180;

const dLng = (lng2 - lng1) * Math.PI / 180;

const a = Math.sin(dLat/2) * Math.sin(dLat/2) +

Math.cos(lat1 Math.PI / 180)

Math.cos(lat2 Math.PI / 180)

Math.sin(dLng/2) * Math.sin(dLng/2);

const c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));

return R * c; // Distance in km

}
```

` **Accuracy:** Very accurate for distances up to a few hundred kilometers

---

> ### Q40: What's the difference between your route optimization algorithms?

Answer: **1. Nearest Neighbor (Greedy)** `

```
Start at depot

While unvisited locations exist:

Go to nearest unvisited location
```

Return to depot

`

Fast: O(n²)

Simple to implement

Good enough for small routes

May not find optimal solution

2. 2-Opt (Local Search) `

Start with any route

For each pair of edges:

If swapping improves distance:

Swap edges

Repeat until no improvement

`

Better solutions than greedy

Still relatively fast

Can get stuck in local optima

3. Genetic Algorithm `

Create population of random routes

For each generation:

Select fittest routes

Crossover (combine) routes

Mutate (random changes)

Until convergence

`

Best solutions

Slower: O(generations × population × n)

Good for complex constraints

Used for production optimization

---

15. Defense Tips

Before the Defense

□ Test the application on the presentation device □ Ensure stable internet connection □ Have backup screenshots/video in case of technical issues □ Prepare demo accounts (admin + driver) □ Clear browser cache for clean demo

During Demo

Show These Features:

- Login flow (both admin and driver)
- Admin dashboard with truck overview
- Driver dashboard with assigned route
- GPS tracking (simulate with browser dev tools if needed)
- Route completion with photo upload
- Fuel Management page with data
- Completion History with trip stats

Answering Questions

Be honest about limitations

Acknowledge what could be improved

Explain trade-offs you made and why

Reference specific code locations when technical

Relate features back to solving real problems

Common Pitfalls to Avoid

Don't claim 100% accuracy for estimations

Don't say "it just works" - explain how

Don't dismiss scalability concerns

Don't forget to mention security measures

Don't rush the demo - let features speak

Quick Reference Card

Key Technologies

Backend: `Node.js, Express 4.18`

Database: `MongoDB with Mongoose 8.0`

Auth: `JWT (jsonwebtoken 9.0)`

Security: `bcryptjs 2.4`

Maps: `Leaflet.js 1.9 + OpenStreetMap`

Styling: `Tailwind CSS`

Deployment: `Vercel + MongoDB Atlas`

Key Files

server.js` - Main entry point`

public/app.js` - Frontend logic`

routes/completions.js` - Route completion + auto fuel`

routes/fuel.js` - Fuel management`

routes/tracking.js` - GPS tracking`

models/` - Database schemas

Key Algorithms

Haversine - GPS distance calculation

Fuel Estimation - Speed factor × distance × load

Rate Limiting - Queue-based geocoding

Document prepared for Kolek-Ta Capstone Defense Last updated: December 2024

Kolek-Ta Waste Collection Management System - Capstone Project

Prepared for Panel Defense - December 2024

38