APOGEE ASSIGNMENT TASK

Q1) Tell us something that is not already in your resume. This could be your hobbies, current internship search experience, your passion for sports, philosophical, mythology or any topic that you are interested in. Don't forget to share why this interests you.

Outside of what's listed on my resume, a big part of my current learning comes from my internship experience at Voibe, a startup building voice-first AI systems. What excites me most there is working at the intersection of messy real-world inputs and clean, usable software. I've been involved in experimenting with voice cloning pipelines and researching small language models (SLMs) that specialize in text structuring tasks like punctuation restoration, formatting spoken dictation into readable text, and minimizing hallucinations.

This work has taught me that good software isn't just about powerful models—it's about reliability, constraints, and user trust. For example, instead of chasing larger models, I've spent time benchmarking lightweight, purpose-built models that run fast and deterministically, because that's what actually improves user experience in production. I enjoy this kind of problem-solving: breaking vague human input into structured, actionable outputs. That mindset is something I find myself applying beyond AI as well—thinking about how systems can reduce friction, guide behavior, and unlock efficiency, which is why Apogee's approach to rethinking software sales strongly resonates with me.

Q2) You are building a simple mobile app that shows a list of items from an API.

1. Write the steps you would take to build this app. What considerations are you making?

To build a simple mobile app that displays a list of items from an API, I would start by understanding the basic requirements such as what kind of data needs

to be shown, how frequently it updates, and how the user is expected to interact with it.

After that, I would choose an appropriate mobile development framework like React Native or Flutter to build the app efficiently. I would then create a separate API service layer to fetch data from the backend and handle different states such as loading, success, and failure. Once the data is received, I would parse and validate it before displaying it in a list-based UI component, making sure the app performs smoothly even if the list grows large. I would also add basic error handling, loading indicators, and empty states so the user clearly understands what is happening in the app. Finally, I would test the app on different devices and network conditions to ensure reliability and a good user experience.

2. List assumptions you are making.

While building this app, I am assuming that the API is already available, well-documented, and returns data in a consistent format such as JSON. I am also assuming that authentication is either not required or handled separately, and that the number of items returned by the API is reasonable or supports pagination. Another assumption is that the API response time is suitable for a mobile application and does not cause major delays for the user.

3. List at least 3 things that could go wrong and how you'd debug them.

A few things could go wrong while building this app. One common issue is that the API might fail or return errors, in which case I would first test the API using tools like Postman to confirm whether the issue is on the backend or the client side. Another possible problem is the app crashing due to unexpected or missing fields in the API response, which I would debug by logging the raw response and adding proper null checks and validation. Performance issues such as slow list rendering can also occur, especially with large datasets, and I would debug this by profiling the app, implementing pagination or lazy loading, and optimizing how the list is rendered.

Q3) You have to build an app that manages all your action items. An action item could be to remind,

to send an email, to set up a calendar invite, to prioritise.

1. Explain how you would go about doing this.

2. Chalk out an implementation plan- what do you build first, how and why.

3. If these are to be managed by both web and mobile interface, list the changes you would

consider.

4. If you were to support offline mode, what could you cache and how would you invalidate

it? You might as well say, I don't like to cache anything with your reasoning.

From my perspective any system can be designed and get scaled by using these 5-step implementation plan.
1) Noting Functional & Non- functional Requirements
2) Selection of Database & making schema design

3)Data Flow & API Design
4) Drawing the high-level system design which satisfies only the functional requirements

5) Upgrading the existing design to satisfy the non-functional requirements.

1) Functional & Non-Functional Requirements

Functional requirements

- Users can create, edit, delete, and complete action items
- Action items support multiple types: reminders, emails, calendar events, and priority tasks
- Each action item can have metadata such as due date, priority level, recurrence, and status
- Notifications are triggered based on time and user preferences
- Actions can be reordered, filtered, and searched

Non-functional requirements

- Low latency for create/read/update actions
- High reliability for reminders and notifications
- Consistency across devices (web and mobile)
- Secure handling of user data and credentials
- Scalability to support increasing users and action volume

2) Database Selection, Schema & Design

I would choose a **relational database (MYSQL)** because action items have structured relationships, clear schemas, and require transactional consistency (especially for reminders and calendar events).

Core tables

- users (id, email, auth_provider, created_at)
- action_items (id, user_id, type, title, description, priority, status, due_at, created_at, updated_at)
- action_metadata (action_id, key, value) for extensibility
- notifications (id, action_id, trigger_time, delivery_status)

Indexes would be added on user_id, due_at, and status to support fast querying. This schema balances structure with flexibility, allowing new action types without major schema changes.

3) Data Flow & API Design

I would design a **backend-first, API-driven system**.

Data flow

- Client (web/mobile) sends requests to a centralized API layer
- API validates input, applies business rules, and persists data
- Background workers handle scheduled actions like reminders and email sending
- Notification services push updates back to users

API design

- REST for CRUD operations on action items
- Webhooks or async workers for time-based triggers
- Auth via JWT or OAuth for secure cross-platform access

This separation ensures that both web and mobile clients remain thin and consistent.

4) Building the System to Satisfy Functional Requirements

I would build the system in **progressive, high-impact stages**.

Phase 1: Core action management

- CRUD for action items
- Basic prioritization and status handling
- This unlocks immediate user value

Phase 2: Time-based actions

- Reminders and scheduled triggers
- Background job processing using queues
- Ensures reliability independent of client state

Phase 3: Integrations

- Email sending and calendar invites
- Abstracted integration layer so providers can be swapped easily

Phase 4: UX enhancements

- Search, filters, bulk actions
- Improves usability without changing core logic

This approach minimizes risk while delivering value early.

5) Satisfying Non-Functional Requirements (Including Web, Mobile & Offline)

Web & Mobile considerations

- Shared backend and business logic
- Platform-specific UI adaptations (gestures, notifications, offline storage)

- Idempotent APIs to handle retries and flaky networks

Offline support

I would cache:

- Recently accessed action items
- Pending user actions (creates/updates/deletes)

Caching would be done using a local database (SQLite) or If possible I would use Redis or Kafka

Each cached record would have a last_synced_at timestamp and a version hash.

Invalidation strategy

- On reconnect, sync local changes first
- Fetch server updates based on timestamps
- Resolve conflicts using last-write-wins or user prompts

I would avoid caching anything related to notifications or triggers, as correctness matters more than availability in those cases.

Github Link : https://github.com/SagarMaddela/Action-Items-Manager

Written by
Venkata Sagar Maddela.