MOBILE SYSTEM DESIGN

A Simple Framework for Mobile System Design Interviews

A complete guide for iOS & Android engineers

Alex Lementuev and contributors

Compiled edition · July 9, 2026

ABOUT THIS EDITION

This book was compiled automatically from the public repository github.com/weeeBox/mobile-system-design. The source files were read without modification; only internal links and image paths were remapped so the material works as a single document.

Generated on July 9, 2026 for personal, offline reading.

Contents

PART I

The Interview Framework

The complete interview framework — from requirement gathering to the final Q&A — illustrated end-to-end with the "Design Twitter Feed" example.

IN THIS PART

CHAPTER 1

A Simple Framework For Mobile System Design Interviews (iOS & Android)

This guide provides a framework for approaching mobile system design interviews, specifically for iOS and Android roles. The core idea is that interviewers focus on assessing your thought process and communication skills rather than expecting a perfect, production-ready solution within a limited time. By showcasing your strengths, you enhance your overall evaluation as a candidate.

Join the "Mobile System Design" Discord server for general discussions and feedback.

Disclaimer

This framework is inspired by "Scalable Backend Design" resources. Understanding this guide doesn't guarantee success. Interview structure varies based on the interviewer's style. Active dialog is vital – ask clarifying questions and avoid assumptions.

This guide does not reflect the interviewing policies from Google (or any other company).

Interview Process (45-60 min)

- 2–5 min: Introductions
- 5 min: Task definition and requirement gathering
- 10 min: High-level discussion
- 20-30 min: Detailed discussion
- 5 min: Questions for the interviewer

Introductions

Briefly share your relevant experience. Example: "*Hi, I'm Alex. I've been developing for iOS/Android since 2010, focusing on frameworks and libraries. For the last 2.5 years, I led a team on the XYZ project, covering system design, planning, and technical mentorship.*" The purpose is to break the ice and highlight relevant skills.

Defining The Task

The interviewer presents the task, like "Design Twitter Feed." Determine the task's scope:

- **Client-side Only:** Design the app, given a backend and API.
- **Client-side + API:** Design both the client app and API. (Most Common).
- **Client-side + API + Backend:** Design the app, API, and backend. (Less Likely). Be transparent if your backend experience is limited. Focus on your mobile expertise and mention your high-level understanding of backend concepts from books, videos, etc.

Gathering Requirements

Categorize requirements into Functional, Non-Functional, and Out-of-Scope. Using "Design Twitter Feed" as an example:

Functional Requirements

Focus on high-value features (3-5).

- Users can scroll through an infinite feed of tweets.
- Users can "like" a tweet.
- Users can view comments for a specific tweet (read-only).

Non-Functional Requirements

These are critical for the product's overall success.

- Offline support: Users should be able to view cached tweets and interact with them even without internet connectivity.
- Real-time notifications: Notify users of new tweets, likes, and comments.
- Optimal resource usage: Minimize bandwidth, CPU, and battery consumption. Consider network optimization techniques and efficient data handling.

Out of Scope

Exclude features important for a real project but not in the interview scope.

- Login/Authentication

- Sending tweets
- Following/Retweeting
- Analytics

Providing the "Signal"

The interviewer evaluates your thought process. Show them:

- Your assumptions and how you stated them clearly.
- The rationale behind your feature choices.
- Relevant clarifying questions.
- Awareness of trade-offs and potential concerns.

Ask many questions and cover diverse aspects. Use a Breadth-First Search (BFS) approach, then ask the interviewer which area they want to explore in-depth. If they offer a choice, select a topic you know well.

Clarifying Questions

Ask these questions during the task clarification step:

- **Do we need to support emerging markets?**
  - Consider app size optimization due to low-end devices and expensive data. Prioritize caching and minimize network requests. On Android, look into using app bundles, on iOS investigate app thinning.
- **What number of users do we expect?**
  - High user count impacts backend load. Discuss Exponential Backoff and API Rate Limiting to prevent unintended DDoS.
- **How big is the engineering team?**
  - Smaller teams (2-4) require different considerations than larger teams (20-100). Focus on modularity and project structure to enable parallel development. Think about using a monorepo versus a multi-repo strategy.
- **What are the target OS versions and device types?**
  - This will drive decisions about architectural patterns (e.g. Compose/SwiftUI vs. older view-based systems), available APIs (e.g. for background tasks, data persistence), and necessary compatibility layers.
- **Are there any specific performance benchmarks we need to achieve?**
  - Understand acceptable latency for feed updates, image loading, and other key interactions. This will influence caching strategies, network optimization, and data serialization techniques.
- **What level of data consistency is required?**
  - Can we tolerate eventual consistency for likes and comments, or do we need strong consistency? This impacts decisions about offline behavior and data synchronization strategies.

- **How frequently is the data updated?**
  - If tweets are very frequent, then a pull-based refresh strategy might overwhelm the server. Push notifications or SSE might be more appropriate.
- **What are the expected types of media?**
  - Image sizes and formats will influence caching and networking strategies. Consider supporting modern formats such as webp or AVIF. Video will require different considerations, such as streaming protocols (HLS, DASH), transcoding, and CDN support.

High-Level Diagram

If the interviewer is satisfied, ask if they want to see a high-level diagram.

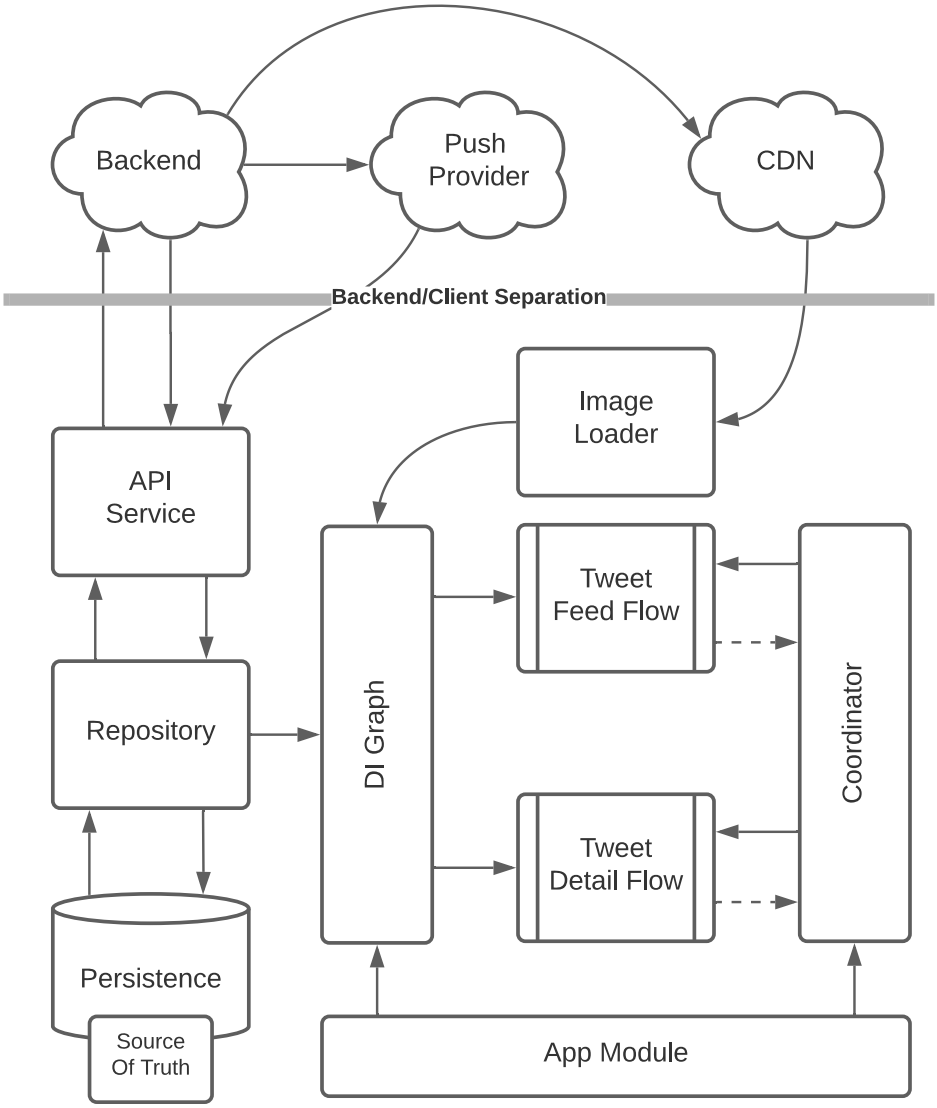


FIGURE 1.1 High-level Diagram

Server-side Components:

- **Backend:** The overall server infrastructure. Focus on client-side concerns.

- **Push Provider:** Infrastructure for mobile push notifications. Delivers payloads from the backend.
- **CDN (Content Delivery Network):** Delivers static content (images, videos) to clients efficiently.

Client-side Components:

- **API Service:** Abstract client-server communication. Provides a clean interface for interacting with the backend. Consider using a library like Retrofit (Android) or Alamofire (iOS).
- **Persistence:** The single source of truth for data on the device. Data is persisted locally first and then propagated to other components. This enables offline access and reduces network dependency.
- **Repository:** Mediator between API Service and Persistence. Decouples data sources from the UI.
- **Tweet Feed Flow:** Components responsible for displaying the scrollable tweet list.
- **Tweet Details Flow:** Components responsible for displaying a single tweet's details.
- **DI Graph:** Dependency injection graph. Enables loose coupling and testability (Dagger/Hilt on Android, Swinject/Typhoon on iOS).
- **Image Loader:** Loads and caches images (Glide/Coil on Android, SDWebImage/Kingfisher on iOS). Check out Image Loading Strategies Deep Dive.
- **Coordinator:** Manages navigation and flow logic between Tweet Feed and Tweet Details, decoupling the components. Check out Mobile Navigation Architecture Deep Dive.
- **App Module:** Executable part of the system that "glues" components together. Responsible for application lifecycle management.
- **Analytics Service:** A module responsible for collecting and transmitting analytics data, such as user actions and performance metrics.

Providing the "Signal"

The interviewer looks for:

- Ability to present the "big picture" without unnecessary details.
- Identification of major building blocks and their interactions.
- Consideration of app modularity and team collaboration.

Frequently Asked Questions

Why is a high-level diagram necessary? Can I skip it?

Not mandatory, but it has advantages:

- **Time management:** Quickly outlines the system and raises discussion points.
- **Modularity:** Each component can be a separate module for team collaboration.
- **Best practices:** Mirrors backend system design and the C4 model for visualizing architecture (https://c4model.com/).

Still not sure? Ask the interviewer if they want a diagram or prefer to jump into specific components.

Deep Dive: Tweet Feed Flow

The interviewer might focus on a specific component, like "Tweet Feed Flow". Discuss:

- **Architecture patterns:** MVP, MVVM, MVI, Clean Architecture, Redux. MVC is generally discouraged. Choose a well-known pattern for easier onboarding.
- **Pagination:** Essential for infinite scrolling. (See Pagination section).
- **Dependency injection:** Improves module isolation and testability.
- **Image Loading:** Discuss low-res placeholders, full-res loading, and scrolling performance.

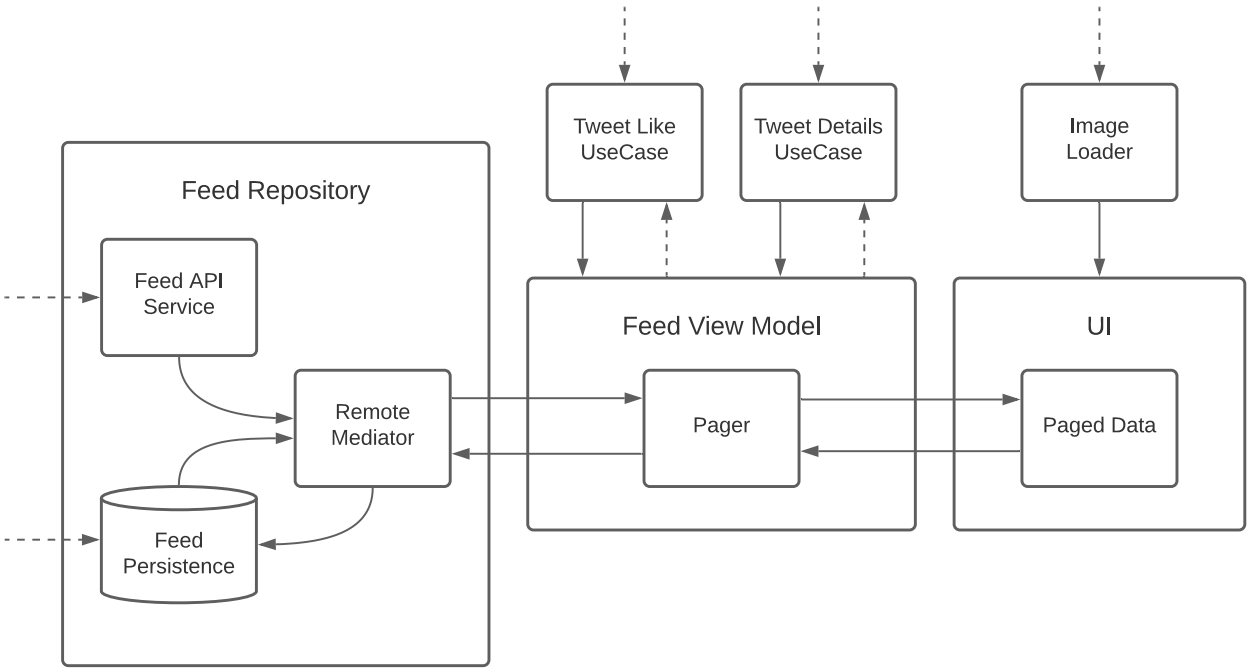


FIGURE 1.2 Details Diagram

Components

- **Feed API Service:** Abstracts the Twitter Feed API client, fetching paginated data. Injected via DI.
- **Feed Persistence:** Abstracts cached paginated data storage. Injected via DI.
- **Remote Mediator:** Fetches next/previous page data and saves it to the persistence layer.
- **Feed Repository:** Consolidates remote and cached responses into a `Pager` object using the `RemoteMediator`.
- **Pager:** Triggers data fetching from the Remote Mediator and exposes an observable stream of paged data to the UI.
- **"Tweet Like" and "Tweet Details" use cases:** Implement "Like" and "Show Details" operations. Injected via DI.
- **Image Loader:** Abstracts image loading from the image loading library. Injected via DI.

Providing the "Signal"

The interviewer assesses:

- Familiarity with common MVx patterns.
- Separation of business logic and UI.
- Understanding of dependency injection.
- Ability to design self-contained modules.

Frequently Asked Questions

How much detail should I provide?

No strict rule. Work with the interviewer; ask if they want more detail or to move on. Observe their body language. If they interrupt, stop and ask if they have questions. Collaboration is key.

Why no mention of specific classes (`RecyclerView`/`UICollectionView`) or vendors (Room, CoreData, Realm)?

- To keep the guide platform-agnostic. Libraries evolve rapidly. Abstracting functionality is more robust.
- Vendor selection is subjective and based on experience/trends.
- Large companies often build custom stacks.
- Implementation details are widely available online.

What drawing tool should I use?

Excalidraw and Google Drawings are popular choices. Some prefer collaborative editors and verbal discussion. Privacy policies may limit tool choices.

Client Library Design

If the interview task is to design a **Library** (e.g., Image Loader, Analytics SDK) rather than an App, the focus shifts to the **Public API Design**.

Providing the "Signal"

The interviewer assesses:

- **API Ergonomics:** Do you design interfaces that are intuitive and hard to misuse? (e.g., using `Result` types instead of `Exceptions`).
- **Stability:** Do you consider thread safety, idempotency, and graceful error handling?
- **Resource Management:** Do you ensure the library doesn't block the main thread or leak memory (avoiding long-lived references to Contexts/Views)?

Check out In-App API Design & Library Architecture Deep Dive for specific patterns (Builder, Singleton, Dependency Injection) and code examples for iOS and Android.

API Design

Cover as much ground as possible. Ask if the interviewer has preferences, or choose an area you know well.

Real-time Notifications

Discuss approaches for real-time updates:

Approach	Pros	Cons	Use Cases
Push Notifications	• **Easier to implement:** Requires less client-side and server-side engineering effort compared to persistent connections. • **Can wake the app in the background:** Allows delivery of updates even when the app is not actively running (within OS limitations). • **Battery efficient (when used correctly):** The OS manages the connection and delivery, optimizing for battery life.	• **Not 100% reliable:** Delivery is not guaranteed due to network conditions, device state, or platform limitations (e.g., Doze mode on Android). • **May have delays:** Delivery latency can vary depending on network conditions and the push notification provider. • **Relies on 3rd-party service:** Dependent on Apple's APNs (iOS) or Google's FCM (Android), introducing a point of failure and potential vendor lock-in.	Breaking news alerts, promotional offers, social media mentions, messaging apps (when app is backgrounded).
Short Polling	• **Simple to implement:** Relatively straightforward client-side and server-side logic. • **No persistent connection:** Avoids the complexities of managing and maintaining persistent connections. • **Compatible with firewalls and proxies:** Less likely to be blocked by restrictive network environments. • **Easy to scale horizontally:** The stateless nature of HTTP makes it easier to distribute load across multiple servers. • **Less expensive (if infrequent):** Lower server resource utilization if the polling interval is large. • **Can be useful for batch updates:** Polling can be used to retrieve multiple updates in a single request.	• **Potential for significant delays:** Notification delivery is limited by the polling interval. • **Overhead from TLS handshake and HTTP headers:** Repeated connections introduce significant overhead, especially for small updates. • **Inefficient bandwidth usage:** The client sends requests even when there are no updates available. • **High server load:** Frequent polling can strain server resources, especially with a large number of clients.	Dashboards with low refresh requirements (e.g., status checks every 30s), legacy systems, low-traffic apps.
Long Polling	• **Instant notifications (almost):** Server holds the connection open until an update	• **More complex, requires more server resources:** Requires more sophisticated server-side logic to	Web-based chat applications (fallback for WebSockets), simple notification systems

Approach	Pros	Cons	Use Cases
	is available, minimizing latency.	manage connections and track updates. • **Keeps a persistent connection (sort of):** Although not a *true* persistent connection like WebSockets, it simulates one, and managing timeouts is critical. • **Still introduces some latency:** Dependent on server response time even with updates readily available. • **Susceptible to connection timeouts and interruptions:** Requires robust error handling and reconnection logic. • **Can be more complex to scale:** Requires sticky sessions or other mechanisms to ensure that the client connects to the correct server instance.	where WebSockets are blocked or overkill.
SSE (Server-Sent Events)	• **Real-time updates with a single connection:** Establishes a unidirectional (server-to-client) stream for efficient delivery of updates. • **Simpler than WebSockets:** Easier to implement and manage compared to bidirectional communication protocols. • **Standard protocol:** Leverages standard HTTP protocol, making it easier to integrate with existing infrastructure. • **Lower overhead than long polling:** Avoids the overhead of repeatedly establishing and tearing down connections. • **Text-based protocol:** Easier to debug and inspect compared to binary protocols.	• **Keeps a persistent connection:** Requires server resources to maintain the connection. • **Unidirectional:** Only supports server-to-client communication, making it unsuitable for applications that require frequent client-to-server updates. • **Limited browser support (older browsers):** May require polyfills for older browsers. • **Connection limits:** May be subject to connection limits imposed by browsers or proxies.	Live sports scores, stock tickers, social media news feeds, status updates.
WebSockets	• **Can transmit binary and text data:** Supports both text-based and binary data, enabling a wide range of applications. • **Full-duplex communication:** Enables real-time, bidirectional communication between the client and the server.	• **More complex to set up:** Requires more complex server-side and client-side logic compared to HTTP-based protocols. • **Keeps a persistent connection:** Requires server resources to maintain the connection. • **More complex to scale:** Requires specialized infrastructure and tech-	Real-time chat, multiplayer games, collaborative editing (e.g., Google Docs), live location tracking.

Approach	Pros	Cons	Use Cases
	• **Lower latency than HTTP polling:** Avoids the overhead of repeatedly establishing and tearing down connections. • **Efficient use of bandwidth:** Reduces bandwidth consumption by maintaining a persistent connection.	niques to handle a large number of concurrent connections. • **Firewall issues:** May be blocked by firewalls or proxies that do not support WebSockets. • **Increased security concerns:** Requires careful attention to security to prevent attacks such as cross-site scripting (XSS) and denial-of-service (DoS). • **Higher battery consumption:** Maintaining a persistent connection can drain battery life on mobile devices. • **Harder to debug compared to HTTP:** Binary messages can be difficult to inspect.	

Choose an appropriate approach for the task. For "Design Twitter Feed", a combination of SSE (primary channel for "like" updates and potentially new tweet notifications if scale allows) and Push Notifications (for clients without active connections or less critical updates) could be suitable. Consider WebSockets for features like real-time chat or live video streaming (if in scope).

Protocols

REST (Representational State Transfer)

Text-based, stateless protocol for CRUD (Create, Read, Update, Delete) operations. Relies heavily on HTTP methods (GET, POST, PUT, DELETE).

Pros	Cons
• **Easy to learn, understand, and implement:** Widely adopted and well-documented, with numerous libraries and frameworks available. • **Easy to cache using HTTP caching:** Leverages built-in HTTP caching mechanisms (e.g., `Cache-Control` headers) for improved performance. CDNs are very effective with RESTful APIs. • **Loose coupling:** Client and server are independent and can evolve separately as long as the API contract is maintained. • **Stateless:** Each request contains all the information needed for the server to understand and process it, simplifying server-side logic and scalability. • **Widely supported:** Supported by virtually all clients (browsers, mobile apps, etc.) and servers.	• **Less efficient on mobile due to separate connections:** Each request requires a new TCP connection (although HTTP/2 mitigates this somewhat with connection multiplexing). Still can contribute to battery drain. • **Schemaless:** Difficult to validate data integrity and format on the client without custom validation logic or external schema definitions (e.g., OpenAPI/Swagger). This also makes refactoring more risky. • **Stateless:** Requires extra functionality to maintain session state, such as cookies or tokens, adding complexity to authentication and authorization. • **Overhead from metadata and headers:** Each request contains contextual metadata and headers, which can add significant overhead, especially for small payloads. • **Over-fetching/Under-fetching:** Clients often receive more data than they need (over-fetching) or need to make mul-

Pros	Cons
• **Human-readable:** Text-based messages are easy to debug and inspect.	tiple requests to retrieve all the required data (under-fetching).

Check out RESTful API Design Deep Dive for a deeper dive into best practices.

GraphQL

A query language for your API and a server-side runtime for executing those queries. Allows clients to request specific data from multiple resources using a single endpoint.

Pros	Cons
• **Schema-based, typed queries:** Clients can verify data integrity and format against a well-defined schema, improving developer experience and reducing errors. Enables introspection of the API. • **Highly customizable:** Clients can request only the specific data they need, reducing the amount of HTTP traffic and improving performance. Eliminates over-fetching. • **Bi-directional communication with GraphQL Subscriptions (WebSocket-based):** Supports real-time updates via WebSockets, enabling features like live feeds and notifications. • **Strong typing:** Allows for compile-time checks, reducing runtime errors. • **Versioning is less critical:** Because clients only request specific data, backend changes are less likely to break existing clients. • **Developer tools:** Excellent tooling for development, including GraphiQL for exploring and testing queries.	• **More complex backend:** Requires a more sophisticated server-side implementation compared to REST, including a GraphQL server and resolvers. • **"Leaky abstraction" – tight client-backend coupling:** Clients become tightly coupled to the backend schema, making it harder to evolve the API without breaking existing clients. Can complicate refactoring. • **Query performance depends on the slowest service (for federated data):** The performance of a query is bound to the performance of the slowest service on the backend, especially in a federated architecture where data is spread across multiple services. N+1 problem can occur without careful consideration. • **Complexity with caching:** HTTP caching is more difficult to implement compared to REST, requiring specialized solutions. • **Security considerations:** Requires careful attention to security to prevent malicious queries and unauthorized access.

WebSocket

Full-duplex communication over a single TCP connection.

Pros	Cons
• **Real-time, bi-directional communication:** Enables continuous communication between the client and the server, ideal for applications that require low-latency updates. • **Supports text and binary data:** Can transmit both text-based and binary data, making it versatile for various use cases.	• **Requires maintaining an active connection:** Requires server resources to maintain the connection, potentially leading to scalability challenges. • **Schemaless:** Difficult to validate data integrity and format on the client without custom validation logic. This can also make debugging more difficult. • **Limited number of active connections per server (65k limitation is theoretical and OS-dependent):** The actual limit depends on the server operating system and hardware, but managing a very high number of

Pros	Cons
• **Low latency:** Reduces latency by maintaining a persistent connection, avoiding the overhead of repeatedly establishing and tearing down connections. • **Efficient bandwidth usage:** Reduces bandwidth consumption by only sending data when there are updates.	concurrent connections can be challenging. Connection pooling and proper resource allocation are crucial. • **More complex to implement than HTTP polling or SSE:** Requires more sophisticated client-side and server-side logic. • **Stateful:** The server needs to maintain state about each client connection, which can complicate scaling and fault tolerance.

Learn more:

- WebSocket Tutorial - How WebSockets Work

gRPC (gRPC Remote Procedure Calls)

A modern open source high performance Remote Procedure Call (RPC) framework that runs on top of HTTP/2. Supports bi-directional streaming using a single physical connection.

Pros	Cons
• **Lightweight binary messages (smaller than text-based):** Uses Protocol Buffers (Protobuf) for message serialization, resulting in smaller message sizes and improved performance. • **Schema-based:** Built-in code generation with Protobuf, ensuring type safety and reducing errors. • **Supports event-driven architecture:** Provides support for server-side streaming, client-side streaming, and bi-directional streaming, enabling a wide range of use cases. • **Multiple parallel requests:** Leverages HTTP/2 multiplexing to send multiple requests over a single connection. • **High performance:** Optimized for low latency and high throughput. • **Strong typing:** Provides strong typing for both requests and responses, reducing runtime errors. • **Good performance on unreliable networks:** Protocol Buffers are designed to be resilient to data corruption and network disruptions.	• **Limited browser support:** Requires gRPC-Web for browser clients, which adds complexity. While support has improved, it's still not as seamless as REST or GraphQL. • **Non-human-readable format:** Binary messages are difficult to debug and inspect without specialized tools. • **Steeper learning curve:** Requires familiarity with Protocol Buffers and gRPC concepts. • **Increased complexity compared to REST:** More complex to set up and configure compared to REST. • **Heavier on client:** The client library required for gRPC is generally larger compared to REST, which may impact app size. • **Not as widely understood as REST:** REST is much more widely understood, and developers are more likely to be familiar with it.

MQTT (Message Queuing Telemetry Transport)

A lightweight, publish-subscribe network protocol that transports messages between devices. Often used for IoT applications but can be suitable for mobile in specific scenarios.

Pros	Cons
• **Lightweight:** Minimal bandwidth usage and low overhead, ideal for constrained networks. • **Publish-Subscribe:** Decouples message producers and consumers.	• **Binary protocol:** Difficult to debug and inspect. • **Not as widely used as HTTP-based protocols:** Less mature ecosystem compared to REST, GraphQL, and WebSockets.

Pros	Cons
• **Real-time communication:** Low latency and quick message delivery. • **QoS Levels:** Provides different Quality of Service (QoS) levels to ensure message delivery based on importance (at most once, at least once, exactly once).	• **Stateful:** Requires managing client connections and subscriptions. • **Security Considerations:** Requires careful configuration to secure the MQTT broker and client connections. • **Broker Dependency:** Relies on a central MQTT broker for message routing, which can be a single point of failure.

Select an appropriate protocol. REST is suitable for "Design Twitter Feed" due to its simplicity for many of the initial tasks, though consider GraphQL for more complex data fetching requirements in the future. For truly real-time scenarios (if in scope), explore WebSockets or SSE. gRPC may be overkill for the initial feature set but beneficial if significant performance improvements are needed with more complex features. Consider MQTT only if you specifically need to interact with IoT devices or have very constrained network conditions.

Pagination

Essential for endpoints returning lists. Prevents excessive network and memory usage.

Types of Pagination

Strategy	Pros	Cons
Offset Pagination (also known as Limit-Offset Pagination)	• **Easy to implement:** Parameters can be passed directly to a SQL query (e.g., `SELECT * FROM tweets LIMIT 20 OFFSET 100`). • **Stateless on the server:** The server doesn't need to maintain any state about previous requests. • **Simple to understand:** Intuitive for both developers and API consumers.	• **Poor performance with large offsets:** The database needs to scan through a large number of rows before returning the requested page, leading to increased latency, especially for large tables. • **Inconsistent when adding/deleting rows (Page Drift):** If new items are inserted or deleted before the requested offset, the same item might appear on multiple pages, or items might be skipped entirely. This is also known as the "missing item" or "duplicate item" problem. • **Not suitable for real-time feeds:** Page drift makes it unsuitable for rapidly changing datasets. • **Vulnerable to Parameter Manipulation:** Clients can easily manipulate the offset parameter, potentially accessing unauthorized data or causing performance issues.
Keyset Pagination (also known as Seek Method or Time-	• **Translates easily to SQL:** Can be implemented efficiently using `WHERE` clauses in SQL (e.g., `SELECT * FROM tweets WHERE created_at > '2021-05-25T00:00:00' ORDER BY created_at ASC LIMIT 20`).	• **"Leaky abstraction" – pagination aware of the database:** The API exposes details about the underlying data storage (the ordering field), increasing the risk of breaking changes if the database schema changes.

Strategy	Pros	Cons
Based Pagination)	• **Good performance with large datasets:** Avoids the performance issues of offset pagination, as the database can use an index on the ordering field. • **Stateless on the server:** The server doesn't need to maintain any state about previous requests. • **More resilient to insertions:** Insertions after the "after" value won't cause items to be skipped or duplicated (though insertions *before* will affect the total number of pages).	API consumers need to know *what* field to paginate on. • **Only works on fields with natural ordering (timestamps, IDs, etc.):** Requires a field that can be used to consistently order the results. Doesn't work well if there's no clear ordering. • **Can have issues with ties:** If multiple items have the same value for the ordering field, the pagination might not be consistent (addressed using compound keyset pagination). • **Sensitive to data modifications:** Deletions of items within a page can still cause issues. • **Less flexible for random access:** Difficult to jump to a specific page without iterating through all the preceding pages.
Cursor/Seek Pagination (also known as Token-Based Pagination)	• **Decouples pagination from SQL:** The API doesn't expose any details about the underlying data storage or ordering. • **Consistent ordering when inserting/deleting new items:** More robust to data modifications than offset pagination. • **More flexible:** Can support various ordering criteria and complex queries. • **Hides implementation details:** Clients don't need to know which field is being used for sorting/pagination.	• **More complex backend:** Requires generating and managing cursors, which involves encoding and encrypting the pagination state. Also requires decoding and validating the cursor on subsequent requests. • **Does not work well if cursors expire or become invalid:** Requires careful management of cursor lifetime and invalidation. Deletions of *all* items between the 'after_id' and the next item can invalidate the cursor. • **Still potentially sensitive to item deletions:** If items are deleted and IDs re-used. • **Less transparent:** Can be harder to debug issues, as the cursor is an opaque token. • **Not easily bookmarkable:** Cursors might expire, making bookmarking difficult.
Page Number Pagination	• **Easy for users to understand:** Intuitive for users familiar with traditional pagination. • **Simple to implement (initially):** Can be implemented using offset pagination on the backend.	• **Inherits all the cons of offset pagination:** Poor performance with large page numbers, page drift issues, etc. • **Not suitable for infinite scrolling:** Doesn't scale well for applications with a large number of pages. • **Not suitable for real-time data:** Page drift issues.

Choose an approach. Cursor Pagination is suitable for "Design Twitter Feed" as it decouples pagination from the underlying data store and can handle insertions/deletions reasonably well. However, consider the complexity of implementation. If the dataset doesn't change frequently and the API is read-heavy, Keyset pagination might be a simpler and more performant alternative. Offset pagination is generally discouraged unless the dataset is small and doesn't change frequently.

Example API request:

```
GET /v1/feed?after_id=p1234xzy&limit=20
Authorization: Bearer <token>
{
  "data": {
    "items": [
      {
        "id": "t123",
        "author_id": "a123",
        "title": "Title",
        "description": "Description",
        "likes": 12345,
        "comments": 10,
        "attachments": {
          "media": [
            {
              "image_url": "https://static.cdn.com/image1234.webp",
              "thumb_url": "https://static.cdn.com/thumb1234.webp"
            },
            ...
          ]
        },
        "created_at": "2021-05-25T17:59:59.000Z"
      },
      ...
    ]
  },
  "cursor": {
    "count": 20,
    "next_id": "p1235xzy",
    "prev_id": null
  }
}
```

Additional Information

- Evolving API Pagination at Slack
- Everything You Need to Know About API Pagination

Check out Mobile Pagination Strategies & Deep Dive for more details.

Mention HTTP authentication (Authorization header, 401 Unauthorized) and Rate-Limiting (429 Too Many Requests). Keep it brief and simple. Also, discuss potential API versioning strategies if the pagination scheme needs to be updated in the future.

Providing the "Signal"

The interviewer assesses:

- Awareness of network challenges and expensive traffic.
- Familiarity with communication protocols.
- Understanding of RESTful API design.
- Familiarity with authentication and security.
- Understanding of network error handling and rate-limiting.

- Understanding of the trade-offs associated with different pagination strategies.
- Ability to choose a pagination strategy that is appropriate for the specific use case.

Data Storage

Data Storage Options

Local device storage options:

Option	Pros	Cons
Key-Value Storage (UserDefaults/ SharedPreferences/Property List/MMKV)	• **Easy to use built-in API:** Simple APIs for storing and retrieving data. • **Fast for simple operations:** Efficient for small amounts of data. • **Low overhead:** Minimal memory footprint.	• **Insecure:** Data is stored in plain text (use `EncryptedSharedPreferences` on Android, third-party wrappers or `Keychain` on iOS for sensitive data). • **Not for large data:** Performance degrades significantly with large datasets. • **No schema or querying:** Limited querying capabilities. No data integrity constraints. • **No data migration:** Difficult to manage schema changes. • **Poor performance for complex operations:** Inefficient for complex data structures or operations. • **Not ACID compliant:** No guarantee of atomicity, consistency, isolation, or durability.
Database/ORM (SQLite/Room/ Core Data/Realm/ ObjectBox)	• **Object-relational mapping (ORM):** Simplifies data access and manipulation using object-oriented paradigms. • **Schema and querying:** Supports structured data with defined schemas and powerful querying capabilities using SQL or ORM-specific query languages. • **Data migration:** Provides mechanisms for managing schema changes and data migration. • **ACID properties:** Guarantees atomicity, consistency, isolation, and durability. • **Efficient for complex operations:** Optimized for complex data structures and operations.	• **More complex setup:** Requires defining schemas, setting up database connections, and managing ORM configurations. • **Potentially Insecure:** (use SQLCipher or other encryption libraries, properly configured) • **Bigger memory footprint:** Database files can consume significant storage space. • **Performance overhead:** ORM can introduce performance overhead compared to raw SQL queries. • **Steeper learning curve:** Requires familiarity with SQL or ORM concepts.
Custom/Binary (DataStore/ NSCoding/Cod-	• **Highly customizable:** Allows fine-grained control over data serialization and storage.	• **No schema/migration:** Requires manual management of schema changes and data migration. • **Lots of manual effort:** Requires writing custom serialization and deserialization code.

Option	Pros	Cons
able/Protocol Buffers)	• **Performant:** Can be optimized for specific data structures and operations. • **Potentially smaller footprint:** If you carefully control serialization and avoid ORM overhead.	• **Increased complexity:** Requires a deep understanding of data structures and serialization techniques. • **Error-prone:** Serialization and deserialization code can be complex and prone to errors. • **Harder to debug:** Binary formats are difficult to inspect and debug without specialized tools.
On-Device Secure Storage (Keystore/Key Chain/Android Biometrics)	• **Secure:** Provides hardware-backed or OS-level encryption for sensitive data. Protects against unauthorized access and data breaches. • **Integrates with device security:** Leverages device-level security features such as biometrics and PIN/password authentication.	• **Not optimized for general storage:** Designed for storing small amounts of sensitive data such as encryption keys and user credentials. • **Encryption/decryption overhead:** Can introduce performance overhead for encryption and decryption operations. • **No schema/migration:** Limited querying capabilities. Difficult to manage schema changes. • **Complex API:** Can be complex to use correctly. • **Limited storage:** Has restrictions on the amount of storage space available. • **User Dependency:** Reliant on the user configuring a lockscreen.
File Storage (Internal/External Storage)	• **Simple to implement.** • **Suitable for storing large, unstructured data.**	• **No schema.** • **Limited querying capabilities.** • **Can be slow for random access.** • **Requires careful management of file paths and permissions.**

Storage Location

- **Internal:** Sandboxed, private to the app, and not readable by other apps (with few exceptions). Data is deleted when the app is uninstalled.
- **External:** Publicly visible and accessible to other apps (subject to permissions). Data persists even after the app is uninstalled. Generally discouraged for sensitive data.
- **Media/Scoped:** A restricted subset of external storage specifically for media files. Provides enhanced privacy and security compared to traditional external storage. Requires proper permissions and follows specific access patterns.

Storage Type

- **Documents (Automatically Backed Up):** User-generated data that cannot be easily re-generated (e.g., user profiles, documents, settings). Automatically backed up to cloud storage (e.g., iCloud, Google Drive) unless explicitly excluded.
- **Cache:** Regeneratable data that can be downloaded again or recreated (e.g., downloaded images, cached API responses). Can be deleted by the system to free up space. Should not contain critical data.

- **Temp:** Temporary data that is only used for a short period and should be deleted when no longer needed (e.g., temporary files created during processing). Not backed up and can be deleted by the system at any time.

Best Practices

- Store as little sensitive data as possible.
- Use encrypted storage for sensitive data (e.g., `EncryptedSharedPreferences`, `Keychain`, or SQL-Cipher).
- Prevent uncontrolled storage growth. Implement cache cleanup policies and manage file sizes.
- Use appropriate storage locations based on data privacy requirements (internal vs. external storage).
- Consider data backup and restore strategies for user-generated data.
- Use modern data storage APIs (e.g., `DataStore` on Android) for improved performance and security.
- Avoid storing secrets or API keys directly in the code or configuration files.
- Implement proper error handling and logging for data storage operations.
- Test data storage logic thoroughly to ensure data integrity and prevent data loss.

Approach

Select an approach and discuss pros/cons. For "Design Twitter Feed":

Feed Pagination Table

Use Room (Android) or CoreData (iOS) for storing paginated feed data. The schema could look like this:

```kotlin
// Kotlin (Android - Room)
@Entity(tableName = "feed")
data class FeedItem(
    @PrimaryKey val itemId: String,
    val authorId: String,
    val title: String,
    val description: String,
    val likes: Int,
    val comments: Int,
    val attachments: String, // comma separated list of URLs or JSON
    val createdAt: Date,
    val cursorNextId: String?,
    val cursorPrevId: String?
)
```

```swift
// Swift (iOS - Core Data)
// Add attributes for: itemID, authorID, title, description, likes, comments,
// attachments, createdAt, cursorNextId, cursorPrevId
```

- Limit the total number of entries (e.g., 500) to control local storage size. Implement a Least Recently Used (LRU) eviction policy to remove older items.

- Flatten attachments into a comma-separated list of URLs or store them as JSON within a text field. Alternatively, create a separate `attachments` table and join it with the `feed` table on `itemId` for better normalization.
- Store `cursorNextId` and `cursorPrevId` with each item to simplify paged navigation. Use nullable types for the cursor IDs.

Attachments Storage

- Store attachments (images, videos) as files in Internal Cached storage.
- Use attachment URLs as cache keys. Implement an ImageLoader library like Coil (Android) or Kingfisher (iOS) to manage downloading, caching, and displaying images.
- Delete attachments when the corresponding feed items are deleted from the `feed` table. Implement a mechanism for deleting orphaned attachments.
- Limit the cache size (e.g., 200-400 MB) to prevent excessive storage usage. Use an LRU cache eviction policy to remove less frequently used attachments.

For sensitive media data (e.g., private accounts), selectively encrypt image files using encryption keys stored in the Keystore/KeyChain. Use Android Biometrics API or Touch ID/Face ID on iOS for user authentication to access the encrypted data.

Providing the "Signal"

The interviewer assesses:

- Awareness of mobile storage types, security, limitations, and compatibility.
- Ability to design a storage solution for common scenarios, considering performance, security, and scalability.
- Understanding of data modeling and database design principles.
- Familiarity with data persistence libraries and frameworks on Android and iOS.
- Awareness of data encryption and secure storage practices.
- Understanding of cache management and eviction policies.

Additional Topics

Major Concerns For Mobile Development

Key concerns to keep in mind:

- **User Data Privacy:** Leaks can damage business reputation and result in legal penalties. Ensure compliance with privacy regulations such as GDPR and CCPA.
- **Security:** Protect against reverse engineering (more important for Android), data breaches, and unauthorized access. Implement security best practices such as data encryption, secure authentication, and authorization.

- **Target Platform Changes:** New OS releases may limit functionality and tighten privacy. Stay up-to-date with platform changes and adapt the app accordingly. Use compatibility layers or conditional compilation to handle differences between OS versions.
- **Non-reversibility of released products:** Assume releases are final. Use staged rollouts and server-side feature flags to safely deploy new features and minimize the impact of bugs.
- **Performance/Stability:**
  - Metered data usage: Cellular data can be expensive. Optimize network requests and minimize data transfer. Use data compression techniques.
  - Bandwidth usage: Constant radio usage drains battery. Batch network requests and use efficient network protocols.
  - CPU usage: High computation drains battery and causes overheating. Optimize algorithms and use background threads for computationally intensive tasks.
  - Memory Usage: High usage increases the risk of background termination. Use memory profiling tools to identify and fix memory leaks.
  - Startup Time: Slow startup creates a poor user experience. Optimize app initialization and lazy-load resources.
  - Crashers/ANRs: UI freezes and crashes lead to poor app ratings. Implement crash reporting and use error tracking tools to identify and fix bugs.
- **Battery Consumption:** Minimize battery drain by optimizing network requests, CPU usage, and background processing. Use battery profiling tools to identify and fix battery-related issues.
- **Geo-Location Usage:**
  - Don't compromise privacy with location services.
  - Prefer the lowest possible accuracy. Request increased accuracy progressively.
  - Provide a rationale before requesting location permissions. Use coarse location when fine location is not required. Obfuscate precise locations if possible.
- **3rd-Party SDKs Usage:**
  - SDKs can cause performance regressions, security vulnerabilities, and outages.
  - Guard each SDK with a feature flag.
  - Use A/B testing or staged rollout for new SDK integrations.
  - Have a long-term support and upgrade plan. Regularly review and update SDKs to address security vulnerabilities and performance issues.
- **App Responsiveness:** Ensure the app remains responsive to user input even when performing complex tasks. Use background threads and asynchronous operations to avoid blocking the main thread.
- **Testing on Real Devices:** Test the app on a variety of real devices to ensure compatibility and performance. Use cloud-based device testing services to access a wide range of devices.
- **Licensing and Legal Compliance:** Comply with all applicable licensing terms and legal regulations, including open source licenses, privacy policies, and data security laws.

Privacy & Security

Privacy and security should be a central consideration throughout the entire development lifecycle, not an afterthought.

- **Minimize user data collection:** Only collect data that is absolutely necessary for the functionality of the app.
  - Avoid device identifiers (use resettable, pseudo-anonymous IDs). Consider using Instance IDs instead.
  - Anonymize collected data whenever possible. Aggregate data and remove personally identifiable information (PII).
  - Explain what data you are collecting, how it is used, and who it is shared with, in easy to understand terms.
  - Provide users with transparency and control over their data. Allow users to access, modify, and delete their data.
  - Implement data retention policies to ensure that data is only stored for as long as it is needed.
- **Minimize permission usage:** Request only the permissions that are necessary for the app's functionality.
  - Be prepared for permission denials and respect the user's choice. Handle permission denials gracefully and provide alternative functionality if possible.
  - Be prepared for auto-reset permissions. Implement a mechanism for requesting permissions again if they are auto-reset by the system.
  - Be prepared for app hibernation of unused apps.
  - Delegate functionality to system apps (Camera, Photo Picker, File Manager) to reduce the need for permissions.
- **Secure Data Storage:**
  - Assume on-device storage is not secure, even when using KeyStore/KeyChain functionality. Implement encryption to protect sensitive data.
    - Encrypt all sensitive data at rest using strong encryption algorithms.
    - Store encryption keys securely in the KeyStore/KeyChain or a hardware security module (HSM).
    - Use secure data storage APIs such as EncryptedSharedPreferences on Android and the Keychain on iOS.
    - Implement proper access controls to restrict access to sensitive data.
  - Assume backend storage is also not secure.
    - Use end-to-end encryption to protect data in transit and at rest. Implement server-side encryption to protect data stored in the backend.
    - Use secure communication protocols such as HTTPS.
    - Implement proper authentication and authorization mechanisms to restrict access to backend resources.

- **Secure Communication:**
  - Use HTTPS for all network communication.
  - Validate server certificates to prevent man-in-the-middle attacks.
  - Implement proper authentication and authorization mechanisms.
  - Use secure data transfer protocols.
- **Code Security:**
  - Protect against reverse engineering by obfuscating the code and using anti-tampering techniques.
  - Validate user input to prevent injection attacks.
  - Use secure coding practices to avoid common security vulnerabilities.
  - Perform regular security audits and penetration testing.
- **Privacy by Design:**
  - Implement privacy considerations at every stage of the development process.
  - Conduct a Data Protection Impact Assessment (DPIA) to identify and mitigate privacy risks.
  - Use privacy-enhancing technologies (PETs) such as differential privacy and homomorphic encryption.
- **Regular Security Updates:**
  - Stay up-to-date with the latest security vulnerabilities and patches.
  - Regularly update dependencies and third-party libraries.
  - Implement a security incident response plan.
- **Privacy Compliance:**
  - Comply with all applicable privacy regulations, such as GDPR and CCPA.
  - Obtain user consent before collecting or processing personal data.
  - Provide users with the right to access, modify, and delete their data.
  - Implement data breach notification procedures.
- **Transparency and User Education:**
  - Assume platform security/privacy rules will change. Control critical functionality with remote feature flags to quickly adapt to new regulations.
  - User *perception* of security is crucial. Discuss how you would educate users about data collection, storage, and transmission.
    - Be transparent about data collection practices.
    - Provide users with clear and concise information about how their data is used.
    - Educate users about security threats and best practices.
    - Implement a user-friendly privacy policy.

Cloud vs On-Device

Choose between running functionality on a device or in the cloud. Especially relevant for On-Device AI, Machine Learning, and complex processing.

Cloud Execution

Pros	Cons
• **Device-independent:** Customers are not limited by device capabilities (CPU, GPU, memory). Runs consistently across all devices. • **Better usage of client system resources:** Offloads intensive computation, saving device battery and preventing overheating. • **Fast pace of changes:** Updates and improvements can be deployed server-side without requiring app updates, enabling rapid iteration. A/B testing and experimentation are easier. • **Bigger computational resources:** Access to virtually unlimited compute power, enabling complex tasks such as large-scale data processing and complex AI models. • **Better security:** Server-side code is generally more secure than client-side code, as it is not exposed to the user. Easier to protect against reverse engineering and tampering. Secure environments and more sophisticated security tools are usually available. • **Easier analytics and offline data analysis:** Centralized data collection enables comprehensive analytics and insights into user behavior and app performance. Easier to perform batch processing and machine learning on aggregated data. • **Simplified development:** Can simplify mobile development by offloading complex logic and processing to the cloud. Developers can focus on building the user interface and user experience.	• **Increased Latency:** Requires network communication, which can introduce latency and negatively impact the user experience. Can be a major problem for real-time or interactive applications. • **Network Dependency:** Requires a reliable network connection. Users may not be able to access certain features when they are offline or have a poor network connection. • **Cost:** Cloud resources can be expensive, especially for computationally intensive tasks. Need to carefully manage cloud costs and optimize resource utilization. • **Security Risks:** Requires careful attention to security to protect data in transit and at rest. Data breaches can have serious consequences. • **Compliance Challenges:** May be subject to data privacy regulations such as GDPR and CCPA. Requires careful consideration of data residency and data transfer requirements.

On-Device Execution

Pros	Cons
• **Better privacy:** Data does not leave the user's device, providing enhanced privacy and security. Reduces the risk of data breaches and unauthorized access. No data is transmitted over the network. • **Real-time functionality:** Certain operations can run much faster on a user's device, providing a more responsive and interactive user experience. Suitable for applications that require low latency. • **Lower bandwidth usage:** Reduces bandwidth consumption by avoiding the need to send data over the network. Can be important for users with limited data plans or poor network connections. • **Offline functionality:** Allows users to access certain features even when they are offline. Improves the user experience in areas with poor network coverage. • **Lower backend usage:** Reduces the load on the backend servers, potentially saving costs. Distributes the processing load across all client devices. No server costs for computations performed on-device.	• **Device Limitations:** Performance is limited by the device's CPU, GPU, and memory. May not be suitable for computationally intensive tasks or large datasets. • **Inconsistent Performance:** Performance can vary significantly depending on the device. Requires careful optimization to ensure consistent performance across a range of devices. • **Larger App Size:** Can increase the app size due to the inclusion of machine learning models or other large libraries. • **Difficult to Update:** Requires app updates to deploy new features or bug fixes. Can be difficult to ensure that all users are running the latest version of the app. • **Security Risks:** Client-side code can be more vulnerable to reverse engineering and tampering. Requires careful attention to security to protect sensitive data. • **Battery Consumption:** Can drain battery life, especially for computationally intensive tasks.

Pros	Cons
• **Reduced dependency on external services:** Makes the application more resilient to outages or disruptions of external services.	

Things you should never run on a device:

- **New "resource" creation (sensitive):** Generating coupons, tickets, promo codes, or other sensitive resources should always be done server-side to prevent fraud.
- **Transactions and Payment verification:** Never process payment information directly on the device. Delegate to a 3rd-party SDK (e.g., Stripe, Braintree) or redirect the user to a secure payment gateway. Sensitive financial calculations should be performed on the server.
- **Complex Business Logic or sensitive Algorithms:** Sensitive algorithms, user authentication, or core business rules should be kept on the server for security and control. This protects valuable IP and allows for updates without app releases.

Offline and Opportunistic State

Offline state ensures app usability without a network connection:

- User can like/comment/delete tweets offline.
- UI updates "opportunistically" assuming requests will be sent when online.
- App displays cached results.
- App notifies users about its Offline State, and any pending actions.
- State changes are batched and sent when the network goes back online.

Check out Offline-First Architecture & Data Synchronization Deep Dive for a deep dive into synchronization strategies and conflict resolution.

Request de-duplication

Ensure retrying requests won't create duplicates on the server (idempotence). Use unique client-side generated request IDs and server-side de-duplication. The server should store a record of processed request IDs for a certain period.

Use the Idempotency-Key HTTP header to ensure a client can safely retry a request without accidentally performing the same operation twice. The client generates a unique key for each request and includes it in the header. If the server receives the same key again, it knows that the request has already been processed and can return the previous response.

Example (Kotlin):

```kotlin
// Client-side request
val idempotencyKey = UUID.randomUUID().toString()
val request = Request.Builder()
    .url("https://example.com/api/like")
    .header("Idempotency-Key", idempotencyKey)
    .post(requestBody)
    .build()
```

Synching Local and Remote States

Handle state synchronization across multiple devices. Implementing conflict resolution is crucial.

Strategy	Pros	Cons
Local Conflict Resolution (Last Write Wins with Timestamps - often NOT recommended)	• **Easy to implement** (but often incorrect).	• **Insecure** - gives a local device authority over the backend. • **Fails if multiple devices send updates simultaneously** (the last update "wins," potentially losing data). • **Requires app update for merging logic changes.** • **Can lead to data loss and inconsistencies.**
Remote Conflict Resolution (Recommended)	• **Moves conflict resolution authority to the backend.** • **Does not require client updates for resolution logic.** • **More reliable and consistent data synchronization.**	• **More complex backend implementation.** • **Potentially overwrites local, un-synced changes on the client**, requiring careful UX design to manage.

Conflict Resolution Strategies (on the server, using Remote Conflict Resolution):

- **Last Write Wins with Timestamps (Not recommended in most cases):** The server simply accepts the latest update based on the timestamp. This is the simplest approach, but it can lead to data loss if clocks are not synchronized or if updates are received out of order.
- **Conflict Detection and Resolution UI:** The server detects a conflict and returns an error to the client, prompting the user to manually resolve the conflict through a UI. This approach provides the most control over conflict resolution but can be cumbersome for the user.
- **Operational Transformation (OT):** A more sophisticated approach that transforms operations based on the history of changes. This allows for concurrent edits to be merged seamlessly. OT is more complex to implement but provides a better user experience.
- **Conflict-Free Replicated Data Types (CRDTs):** Data structures that are designed to be merged automatically without conflicts. CRDTs are a good choice for applications where data consistency is critical.

More Information

Offline functionality for Trello's mobile applications: Airplane Mode: Enabling Trello Mobile Offline

Caching

Caching is essential for offline support, reducing network usage, and improving performance. It involves storing data in memory (for immediate access) or on disk (for persistence).

Check out Caching Strategies Deep Dive for details on strategies (Read-Through, Stale-While-Revalidate), eviction policies, and security.

Quality Of Service (QoS)

Implementing Quality of Service (QoS) for network operations is critical for providing a smooth user experience while conserving device resources. It involves prioritizing network requests based on their importance (User-Critical vs Background).

Check out Quality of Service Deep Dive for details on concurrency limits, request prioritization, and power management.

Resumable Uploads (Chunked Uploads)

Resumable uploads break down large files into smaller chunks, allowing the upload to be paused and resumed. This improves reliability on unstable networks.

Check out Resumable Uploads Deep Dive for the initialization process, chunk appending, and advantages/disadvantages.

Prefetching

Prefetching proactively fetches data before it is explicitly requested (e.g., preloading the next page of a feed). This reduces latency and improves UX but consumes more resources.

Check out Prefetching Deep Dive for types of prefetching, implementation strategies, and how to mitigate negative impacts.

Conclusion

Navigating system design interviews can feel like an overwhelming task. Remember that there's inherent randomness – the process can differ significantly across companies and even between interviewers within the same company. The key is to focus on what you *can* control and learn from what you can't.

Things You Can Control:

- **Your Mindset:** Approach each interview with a positive and curious attitude. Be enthusiastic about solving the problem and demonstrate a willingness to learn. Be respectful and collaborative, even under pressure. A good attitude leaves a lasting impression.

- **Your Preparation:** Thorough preparation is your strongest weapon. Practice mock interviews with peers, mentors, or online services. The more you practice, the more comfortable and confident you'll become.
- **Your Knowledge Base:** Continuously expand your knowledge of mobile technologies, architectural patterns, and design principles. Stay updated with the latest trends and best practices in iOS and Android development.
  - Immerse yourself in open-source projects: iOS, Android
  - Curated List of Real-world Design Blog Posts
- **How you articulate your thought process:** Clearly and concisely explain your reasoning behind design choices, and be prepared to justify your decisions with evidence and trade-offs. Walk the interviewer through your process step-by-step.
- **Your Resume:** Craft a compelling resume that highlights your accomplishments and demonstrates the impact you've made in your previous roles. Quantify your achievements whenever possible. Tailor your resume to match the specific requirements of the position you're applying for.

Things Beyond Your Control:

- **The Interviewer's Style and Bias:** You might encounter interviewers with different personalities, preferences, or even unconscious biases. Focus on presenting your best self and don't take any perceived negativity personally.
- **The Competition:** There might be other highly qualified candidates with more experience or skills. Focus on showcasing your unique strengths and value proposition.
- **The Hiring Committee's Decision:** The final decision rests with the hiring committee, which considers various factors beyond your control, such as budget constraints, team dynamics, and organizational priorities.

Embrace the Process, Learn from Every Experience:

Remember that every interview, regardless of the outcome, is a valuable learning opportunity. Analyze your performance, identify areas for improvement, and seek feedback from others. Don't be discouraged by rejections – view them as stepping stones toward your ultimate goal.

Failure is a part of the process. Even the most experienced engineers face rejection at some point in their careers. What matters is how you respond to setbacks. Use each experience to refine your skills, expand your knowledge, and strengthen your resolve.

Keep practicing, keep learning, and keep pushing yourself. The right opportunity will eventually come your way. Believe in your abilities, and never give up on your dreams. Good luck on your journey!

Frequently Asked Questions

Q: How do you know this approach works? Why is this guide necessary?

- A: There's no one-size-fits-all approach to system design interviews. The structure of the interview heavily depends on the interviewer's style and the company's specific needs. However, having a structured framework like this helps both the candidate and the interviewer focus on the *content* of the discussion rather than getting bogged down in the organizational aspects of the interview. It provides a common language and a starting point for exploration.

Q: Can you go a bit deeper into [specific technology/pattern/component]?

- A: This guide aims to provide a broad overview of key concepts. It doesn't delve into extreme detail on any specific technology or implementation. The goal is to equip you with a foundational understanding, encouraging you to draw upon your personal experience and expertise to provide concrete solutions during the interview. Remember, the interviewer is assessing your problem-solving skills and your ability to adapt and apply your knowledge.

Q: I'm an interviewer - won't this guide encourage candidates to memorize solutions and "cheat" during the interview?

- A: System design is far more nuanced than coding challenges. Memorizing a specific solution is rarely sufficient, as interviewers can easily adapt the problem or introduce new constraints. This guide aims to provide a foundational understanding, but success hinges on a candidate's ability to reason critically, apply their knowledge creatively, and articulate their thought process clearly. It's usually quite apparent when a candidate is relying solely on memorization rather than demonstrating genuine understanding. Exposure to common patterns and approaches can reduce interview anxiety and allow the candidate to showcase their problem-solving skills more effectively.

Q: What if the interviewer wants to focus on a technology I'm not very familiar with?

- A: Be honest about your strengths and weaknesses. If the interviewer veers into an area where your knowledge is limited, acknowledge that and explain your general understanding of the concepts involved. Emphasize your willingness to learn and your ability to quickly acquire new knowledge. Suggest alternative solutions using technologies you are more comfortable with.

Q: How important is it to draw a diagram? What if I'm not a visual thinker?

- A: Visualizing the system architecture through diagrams can be extremely helpful for both you and the interviewer. It provides a clear and concise way to communicate the overall design and the interactions between components. If you're not naturally inclined to draw diagrams, practice beforehand! There are many online tools that can help. However, if you feel a diagram is not the best way to express yourself, communicate clearly why and offer an alternative representation, such as a well-structured verbal explanation.

Q: How should I handle disagreements with the interviewer?

- A: It's perfectly acceptable to have different opinions than the interviewer. However, approach the discussion respectfully and professionally. Clearly articulate your reasoning and be willing to consider alternative viewpoints. Avoid being argumentative or dismissive. If you strongly disagree, you can politely acknowledge the interviewer's perspective while reiterating your own rationale. The goal is to demonstrate your critical thinking skills and your ability to engage in constructive dialogue.

Q: What if I get stuck and can't think of a solution?

- A: Don't panic! It's okay to pause and take a moment to gather your thoughts. Communicate your thought process out loud. Explain what you're considering and what challenges you're facing. Ask clarifying questions to help you better understand the problem. If you're still stuck, ask the interviewer for a hint or guidance. The interviewer is more interested in seeing how you approach a difficult problem than in finding the perfect solution.

Q: How much detail should I provide when discussing specific components?

- A: The level of detail should depend on the interviewer's interest and the time available. Start with a high-level overview and then dive deeper into specific aspects as prompted by the interviewer. Pay attention to their body language and ask if they want you to elaborate or move on to another topic. Don't get bogged down in minute details unless the interviewer specifically requests it.

Q: How do I choose between different architectural patterns (MVP, MVVM, MVI, etc.)?

- A: There's no single "best" architectural pattern. Each pattern has its own advantages and disadvantages, and the best choice depends on the specific requirements of the project. Be prepared to explain the pros and cons of different patterns and justify your choice based on factors such as complexity, testability, and maintainability. Familiarity with common patterns such as MVVM with a unidirectional data flow (like MVI or Redux) is generally favored.

Q: How can I improve my communication skills for system design interviews?

- A: Practice explaining complex technical concepts in a clear and concise manner. Use diagrams and visual aids to illustrate your ideas. Practice active listening and ask clarifying questions to ensure that you understand the interviewer's perspective. Seek feedback from others on your communication style. Record yourself explaining technical concepts and analyze your performance. Focus on being clear, concise, and confident in your communication.

Q: Should I memorize specific code snippets or API calls for the interview?

- A: While having a general understanding of common APIs and libraries is helpful, it's generally not necessary (or advisable) to memorize specific code snippets. The focus should be on demonstrating your understanding of the underlying concepts and your ability to apply them to solve problems. If you need to refer to a specific API or code example, explain why you're using it and what it accomplishes.

Q: Is it better to focus on a single area of mobile development (e.g., iOS or Android) or have broad knowledge across both platforms?

- A: The ideal approach depends on the specific role and the company's needs. Some roles may require deep expertise in a single platform, while others may value broad knowledge across both platforms. Be honest about your skills and experience. If you have broad knowledge, highlight your ability to learn new technologies and adapt to different platforms. If you have deep expertise in a single platform, emphasize your ability to contribute immediately to the team and your willingness to learn about other platforms as needed.

Additional Information and Resources

Explore these resources to further enhance your preparation:

More System Design exercises can be found here!

Level-Specific Expectations in System Design Interviews

The focus and complexity of system design interviews will vary depending on the target engineering level. The key is to demonstrate skills and thinking appropriate for that level. An approximate engineering level breakdown can be found here (though leveling structures can differ between companies).

> *NOTE: Years of experience are only a rough guide and don't directly translate to a specific level. Individual skills, contributions, and impact weigh much more heavily.*

Junior Engineer

- The system design round is often optional.
- Focus is on foundational knowledge, understanding basic concepts, and implementing small, well-defined components.
- Expect questions about data structures, algorithms, and basic design patterns.
- Limited or no experience with large-scale system design is expected.
- Demonstrate a willingness to learn and a capacity for growth.

Mid-Level Engineer

- The system design round emphasizes practical implementation and the application of architectural patterns.
- Expect discussions about building specific components using platform-specific libraries and frameworks.
- Demonstrate proficiency in coding, testing, and debugging.
- Ability to translate high-level requirements into concrete implementation details.
- Understand the trade-offs between different implementation approaches.

Senior Engineer

- The system design round shifts to a more high-level perspective.
- Focus on system architecture, component interactions, and scalability.
- Expect discussions about multiple components, their communication, and overall system behavior.
- Implementation details are less critical, unless they significantly impact performance.
- Ability to select appropriate technologies and justify the choices.
- Understand the trade-offs between different architectural patterns and design choices.
- Think about modularity, testability, and maintainability.

Staff Engineer

- The system design round transitions from technical to strategic considerations.
- Expect discussions about the target audience, available resources (computational and human), expected traffic, and deadlines.
- Prioritize business needs over technical implementation details.
- Ability to explain how to reduce time-to-market, safely roll out features, and handle large-scale outages.
- Deep understanding of user privacy and its legal implications.
- Consider the long-term impact of design decisions.
- Focus on building consensus and influencing technical direction.
- Expert in scaling systems, choosing the right databases, and understanding networking principles.
- Ability to identify and mitigate risks.
- Think about cost optimization and resource allocation.

Looking for more content?

Interview Template

Use the TEMPLATE.md to structure your notes during mock interviews.

System Design Exercises

Check out the collection of mobile system design exercises.

Common System Design Interview Mistakes

Check the guide here.

Mock Interviews

Check out the mock interview archive or sign-up to be a candidate yourself!

Consider "starring" the repository to help other people discover the guide! Thank you!

PART II

Technical Deep Dives

Focused chapters that expand the framework's core technical topics, presented in the order the framework references them.

IN THIS PART

CHAPTER 2

Image Loading Strategies Deep Dive

Image loading is a foundational system design topic. While "downloading a file" and "loading an image in a list" may appear similar, their technical constraints and performance requirements are distinct.

Objective: Demonstrate an understanding of **memory management**, **scrolling performance**, and **caching hierarchies**.

1. Image Loading vs. File Downloading

Candidates often conflate these two concepts. It is important to clarify the distinction early in the discussion.

- **File Downloader (e.g., PDF, Zip):**
  - **Goal:** Reliability and Data Integrity.
  - **Constraint:** Completing the download regardless of app backgrounding or network instability.
  - **Storage:** Disk (Permanent).
  - **UI:** Typically restricted to progress indicators.
- **Image Loader (e.g., Feed, Gallery):**
  - **Goal:** Responsiveness and Frame Rate.
  - **Constraint:** Decoding speed and RAM efficiency.
  - **Storage:** Multi-tier Cache (RAM + Disk).
  - **UI:** Tightly bound to the UI lifecycle (view recycling).

2. The "Build vs. Buy" Decision

The interviewer will likely ask: "*Would you build this from scratch?*"

- **The Strategic Answer:** "For a production application, I would leverage an industry-standard library (e.g., Glide/Coil for Android; Kingfisher/Nuke for iOS)."
- **The Rationale:**
  - **Edge Case Management:** Libraries handle complex scenarios (Exif rotation, color spaces, progressive JPEGs) that are resource-intensive to implement manually.
  - **Performance:** They utilize highly optimized caching engines.
  - **Resource Allocation:** Using a library allows the engineering team to focus on *product features* rather than maintaining low-level infrastructure.

2.1 Selecting a Library (Evaluation Criteria)

If you choose to use a library, you must justify **how** you selected it. Avoid subjective reasoning.

- **Maintenance:** "I prioritize libraries with active maintenance and rapid resolution times for security vulnerabilities."
- **Community:** "A robust community (evidenced by GitHub stars and StackOverflow activity) suggests long-term viability and easier debugging."
- **Performance:** "Does the library support modern optimizations like Memory Pooling and hardware-backed bitmaps?"
- **Size:** "For a lightweight application, a smaller, modern library may be preferable to a monolithic framework."

2.2 Abstraction and Future-Proofing

Senior engineers understand that libraries evolve or become obsolete.

- **The Problem:** Direct implementation (e.g., hardcoding library calls across hundreds of files) creates high coupling. Migrating to a new library would require extensive refactoring.
- **The Solution:** Implement an **Abstraction Layer** (Interface).

```kotlin
// Generic Interface
interface ImageLoader {
    fun loadImage(url: String, target: View)
}
```

- **The Benefit:**
  - **Maintainability:** Implementation changes are isolated to a single wrapper class.
  - **Testability:** Enables the use of a `FakeImageLoader` for UI tests, loading local assets instantly and eliminating network flakiness.

3. Best Practices

3.1 Caching Hierarchy (The "L1/L2" Strategy)

The most critical architectural decision is the **sizing** of caches. Undersized caches lead to excessive network calls, while oversized caches risk Out Of Memory (OOM) errors.

Memory Cache (L1: Fast & Expensive)

- **Content:** Decoded Bitmaps/Images (ready for rendering).
- **Sizing Strategy:** Avoid hardcoded constants (e.g., "50MB"). Calculate size as a percentage of *available* application memory.
  - **Heuristic: 15% - 25%** of available RAM.

    - **Calculation:** `TargetMemoryCacheSize = (Screen Width * Screen Height * BytesPerPixel) * 4 Screens`. This ensures memory capacity for approximately 3-4 screens of images.
- **Library Defaults:** Most libraries default to roughly 2 screens worth of images or ~25% of available memory.

Disk Cache (L2: Slow & Cheap)

- **Content:** Encoded bytes (JPEG/PNG).
- **Sizing Strategy:** Disk space is abundant but finite.
    - **Heuristic: 250MB - 1GB**.
- **Expiration:** Most libraries use LRU (Least Recently Used) eviction or time-based expiration (e.g., 7 days).

Key Insight: Demonstrate awareness that "Available RAM" varies drastically between low-end and high-end devices, requiring dynamic sizing logic.

3.2 View Lifecycle Integration

- **Problem:** During rapid scrolling, views are recycled. The view originally assigned to "Image A" may be reused for "Image B" before Image A finishes loading.
- **Solution: Request Cancellation**.
    - Before initiating a request for a View, cancel any pending requests attached to that View (using Tags or View IDs).
    - This prevents incorrect images from flickering into view cells after the user has scrolled past them.

3.3 Sampling & Downsampling

- **Problem:** Loading a 4K image (12MB decoded) into a small thumbnail view (requiring only 40KB).
- **Solution:** Decode bounds first (read image dimensions without allocating memory), calculate the sample size, and decode only the necessary pixels.
- **Key Insight:** This demonstrates a proactive approach to preventing OOM crashes.

3.4 Optimizations in Major Libraries

Modern libraries implement complex optimizations. Mentioning these demonstrates deep domain knowledge.

- **Memory Pooling:** Allocating memory for an image is expensive. Instead of garbage collecting old images, they are placed in a "Pool" and reused for subsequent images. This reduces Garbage Collection (GC) frequency and UI stutter.
- **Prefetching:** Initiating image fetches for the *next* screen (or upcoming list items) before they enter the viewport.

- **Background Decompression:** Decompressing JPEG data is CPU-intensive. Performing this on the main thread freezes the UI. Images should always be decompressed on a background thread before being passed to the view.
- **Pause on High-Velocity Scroll:** During rapid scrolling, pause new request submissions and resume only when scrolling speed decreases. This prevents network and CPU congestion caused by requests that will likely be canceled before display.

4. Common Pitfalls ("Red Flags")

- **Decoding on the Main Thread:** A critical error. This blocks the UI thread, causing dropped frames (jank).
- **Caching Raw Responses in Memory:** Inefficient. The cache should store the *decoded* image to avoid wasting CPU cycles re-decoding the image upon every viewing.
- **Ignoring Request Coalescing:** If multiple widgets request the same URL simultaneously, the system should issue a *single* network call and distribute the result to all listeners.
- **Neglecting Priorities:** Visible thumbnails must have higher priority than prefetching images for off-screen content.

5. Real-World Case Studies

- **Glide (Android):** (GitHub)
  - *Feature:* Aggressive memory caching and bitmap pooling.
  - *Signal:* Mentioning `LruBitmapPool` demonstrates understanding of Android's Garbage Collection struggles.
- **Kingfisher (iOS):** (GitHub)
  - *Feature:* `ImageProcessor` pipeline.
  - *Signal:* Shows you know how to handle post-processing (rounding corners, blurring) *after* downloading but *before* caching to save CPU on subsequent reads.
- **Pinterest App:** (Engineering Blog)
  - *Strategy:* Uses a "Staggered Grid" with variable aspect ratios.
  - *Signal:* Discussing how to pre-calculate layout height based on metadata *before* the image loads prevents UI layout jumps.

6. Summary Checklist for the Interview

1. **Differentiation:** Focus on RAM usage and Frame Rate, not just data transfer.
2. **Architecture:** Request Manager -> Memory Cache -> Disk Cache -> Network.
3. **Optimization:** Emphasize Downsampling, Request Coalescing, and Lifecycle Cancellation.

4. **Modern Context:** Acknowledge existing libraries but clearly explain the underlying *concepts* they abstract.

CHAPTER 3

Mobile Navigation Architecture Deep Dive

Navigation is one of the most critical and complex aspects of mobile system design. As apps grow, tight coupling between screens ("Feature A imports Feature B") creates a monolithic spaghetti code structure that is hard to test, refactor, and modularize.

The Challenge: How do you design a navigation system that decouples feature modules, supports deep linking, handles complex flows (e.g., authentication), and maintains state across configuration changes?

1. Core Requirements

- **Decoupling:** Module A should navigate to Module B without knowing Module B exists.
- **Deep Linking:** The app must be able to open any specific screen from a URL (e.g., `app://profile/123`).
- **Result Handling:** Screen B needs to return data to Screen A (e.g., selecting a contact).
- **Back Stack Management:** Handling the hardware back button (Android) or swipe-to-back (iOS) correctly.
- **Context/State Preservation:** Restoring the navigation stack after a process death.

2. Evolution of Navigation Patterns

2.1 Direct Coupling (The "Junior" Approach)

- **Concept:** Screen A directly imports and instantiates Screen B.
  - *Example:* `Navigator.push(new DetailScreen(id))`
- **The Problem: Tight Coupling.** The "Feed" module must compile the "Detail" module. You cannot split them into separate build modules easily. Circular dependencies arise quickly (Profile -> Feed -> Profile), leading to a monolithic architecture that scales poorly.

2.2 The Coordinator Pattern

A dedicated object handles the flow logic, removing it from the UI components.

- **How it works:**
  1. `FeedScreen` calls `coordinator.showDetail(id)`.
  2. `FeedCoordinator` knows how to assemble the `DetailScreen` and its dependencies.
  3. `FeedCoordinator` pushes it onto the navigation stack.
- **Pros:** Isolates navigation logic from UI logic. Excellent for unit testing navigation flows.

- **Cons:** Coordinators can become "God Objects" if not split properly. Still requires the Coordinator to know about the destination classes unless combined with Dependency Injection.

2.3 The Router / URI-Based Navigation (The "Scalable" Approach)

Every screen is a URL.

- **How it works:**
  1. `FeedModule` asks: `Router.navigate("app://detail?id=123")`.
  2. `Router` looks up the registration map.
  3. `Router` instantiates the target screen and pushes it.
- **Pros:** Zero compile-time coupling. Modules only know the Router interface. Identical logic for internal navigation and external Deep Links.
- **Cons:** Loss of type safety (passing strings instead of objects).

3. Deep Dive: Modern Navigation Architecture

3.1 The Declarative Navigation Graph

- **Concept:** A centralized file (JSON/XML/DSL) defines all possible destinations and paths ("Actions").
- **The Signal:** This treats navigation as a **State Machine**. You define the states (Screens) and transitions (Actions).
- **Type Safety:** Modern tools generate code from this graph to ensure type safety for arguments, fixing the main downside of loose URI-based navigation.
- **Modularization:** Large graphs can be split into "Nested Graphs" (e.g., `LoginGraph`, `CheckoutGraph`), allowing different teams to own different parts of the navigation flow.

3.2 Feature-Based Navigation (Multi-Module)

In a modularized app where `FeedModule` and `SettingsModule` are separate binaries that don't know about each other, how do they navigate?

Option A: Interface Injection (The Clean Architecture Way)

1. Define an interface in a shared core module: `interface FeedNavigator { fun goToSettings() }`.
2. `FeedModule` depends on `FeedNavigator`.
3. The main `AppModule` (which composes everything) implements `FeedNavigator` and injects it into `FeedModule`.

Option B: Implicit Deep Links (The Loose Coupling Way)

1. `FeedModule` asks the Router to navigate to a generic URI: `app://settings`.
2. The Router resolves this string to the `SettingsScreen` at runtime.
3. *Trade-off:* Loss of compile-time safety (typos in strings cause runtime failures).

3.4 The Back Stack & Deep Linking

A common interview question: "*If a user opens a Deep Link to `app://profile/settings`, what happens when they press Back?*"

- **The Wrong Answer:** "The app closes." (This feels broken to the user).
- **The Right Answer (Synthetic Back Stack):** The navigation system must recognize that `Settings` belongs to a hierarchy. It should synthesize the parent screens (`Home` -> `Profile` -> `Settings`) onto the back stack so the user navigates "Up" the hierarchy, preserving the expected user flow.

4. Common Pitfalls ("Red Flags")

- **The "God" Router:** Handling all navigation logic in a single file or class. Split logic into sub-routers or coordinators per feature.
- **Circular Dependencies:** Feature A importing Feature B directly, creating a monolithic build graph that prevents parallel compilation.
- **Ignoring Process Death:** Failing to save and restore the navigation stack state. If the OS kills the app to save memory, the user must return to the exact same screen hierarchy, not the home screen.
- **Stringly-Typed Navigation:** Using raw strings (`"app://detail"`) everywhere without a centralized constant file or builder pattern. This leads to runtime crashes from simple typos.

5. Summary Checklist for the Interview

1. **Decoupling Strategy:** "I will use a Router pattern (or Coordinators) to ensure Feature A doesn't depend on Feature B."
2. **Deep Linking:** "I will treat internal navigation and external deep links uniformly using a centralized URL resolver."
3. **Type Safety:** "I will use code generation (like SafeArgs or a custom struct generator) to ensure parameter type safety between decoupled modules."
4. **State Restoration:** "I will ensure the Router can serialize the back stack to handle process death."

6. Further Reading

- **Square:** The blind leading the blind (Coordinators)
- **Uber:** RIBs Architecture (Router-Interactor-Builder)
- **Android:** Navigation Component Guide
- **Airbnb:** Deep Link Dispatch

CHAPTER 4

In-App API Design & Library Architecture Deep Dive

In mobile system design interviews, you may be asked to design a **client-side library** rather than a full app. Examples include:

- "Design an Image Loading Library" (like Glide/Kingfisher)
- "Design an Analytics SDK" (like Segment/Firebase)
- "Design a Networking Library" (like Retrofit/Alamofire)

The interviewer is testing your "**API Ergonomics**" - can you design a tool that other developers will love to use and find difficult to break?

1. The Core Philosophy: "Easy to Learn, Hard to Misuse"

Joshua Bloch (author of *Effective Java*) defined the gold standard for API design. In an interview, explicitly mentioning these two goals provides a strong signal.

Discoverability: Is it easy to use?

The "Hello World" of your library should be achievable with a single line of code.

- **Zero Configuration:** Users should not be required to manually define cache sizes, thread pools, or timeouts for standard requests. Sensible defaults are essential for a positive developer experience.
- **Fluent Interfaces:** Chaining methods guides the developer through configuration, allowing IDE autocomplete to serve as a form of "living documentation."

Example: Image Loading

- *Bad Design:*

```kotlin
val loader = ImageLoader(context, cacheSize = 100, threadPool = Executors.newFixedTh
readPool(4))
loader.loadImage(url, imageView, null, null) // What are those nulls?
```

- *Good Design (Glide/Picasso style):*

```kotlin
Glide.with(context)
    .load(url)
    .placeholder(R.drawable.loading)
    .into(imageView)
```

Safety: Is it hard to misuse?

If a developer provides invalid parameters, the library should handle the error gracefully or fail predictably.

- **Fail Fast:** In development environments, the library should surface configuration errors immediately with descriptive messages rather than failing silently or causing inconsistent states.
- **Type Safety:** Utilize `Enums` or `Sealed Classes` instead of raw Strings or Integers to restrict inputs to valid, compile-time checked values.

Example: Network Methods

- *Bad Design:* `request("GET", url)` -> Developer can typo "get".
- *Good Design (Alamofire/Retrofit):* `request(.get, url)` -> Compiler ensures validity.

---

2. Key Topics for the Interview

2.1 Error Reporting: How do you tell the developer something went wrong?

In mobile development, I/O operations (Network/Disk) fail frequently. How you expose these errors defines the quality of your API.

- **Approach 1: Exceptions (Avoid for Async)**
  - *Why to avoid:* In async callbacks, thrown exceptions often crash the app unexpectedly or get swallowed by the thread pool.
- **Approach 2: Nullable Returns (Avoid)**
  - *Why to avoid:* Returning `null` forces the developer to guess *why* it failed. Was it network? Parse error? Disk full?
- **Approach 3: Callback Hell (Common but Dated)**
  - `onSuccess(data)` / `onError(exception)`
- **Approach 4: Result Types / Sealed Classes (Best Practice)**
  - Forces the developer to handle both success and specific failure cases.

Android Example (Kotlin Sealed Class):

```kotlin
sealed class NetworkResult<out T> {
    data class Success<T>(val data: T) : NetworkResult<T>()
    data class Error(val exception: Exception, val isTransient: Boolean) :
NetworkResult<Nothing>()
}

// The compiler forces you to handle the Error case
when(result) {
    is Success -> showData(result.data)
    is Error -> if (result.isTransient) retry() else showError()
}
```

iOS Example (Swift Result Type):

```swift
func fetchUser(completion: (Result<User, NetworkError>) -> Void) { ... }

// Usage
switch result {
case .success(let user): display(user)
case .failure(let error): handle(error) // Error is strongly typed
}
```

2.2 Threading: The "Main Thread" Rule

Your library **must never** block the UI thread.

- **Mistake:** Doing I/O in the constructor or `init` block.
- **Best Practice:** All I/O should happen on a background dispatcher, and results should be delivered to the **Main Thread** (for UI libraries) or the caller's thread (for generic libraries).

---

3. Common Pitfalls ("Red Flags")

The "God Config" Object

Passing a massive configuration object with 20 optional parameters into a constructor.

- **Fix:** Use the **Builder Pattern**. It separates the construction of a complex object from its representation.

Leaking Implementation Details

Exposing internal libraries (e.g., exposing `OkHttp` types in your custom networking library's public API).

- **Why it's bad:** If you switch from `OkHttp` to `Cronet` later, you break every app using your library.
- **Fix:** Wrap internal objects in your own Domain Models.

Static Mutable State

Avoid global variables that control state. It makes testing impossible (tests can't run in parallel) and causes race conditions. Use Dependency Injection instead.

Swallowing Errors

Avoid catching exceptions without appropriate handling. At a minimum, errors should be logged or surfaced to the caller to prevent silent failures in production.

"Magic" Behavior

Avoid implementing hidden side effects, such as exhaustive automatic retry loops or implicit lazy initialization that may affect the caller's thread. These behaviors can mask underlying performance issues and lead to unpredictable execution states. Favor deterministic logic that requires explicit configuration for resource-intensive or state-altering operations.

---

4. Real-World Case Studies (For "Signal")

Mentioning these libraries shows you study the ecosystem.

- **Retrofit (Android):** (GitHub)
  - *Why it's great:* It uses **Annotations** (`@GET`, `@POST`) to turn an API definition into a declarative Interface. It abstracts away the complexity of parsing and request creation.
- **Alamofire (iOS):** (GitHub)
  - *Why it's great:* It uses **Parameter Encoding** and **Response Validation** chaining.
  - `AF.request(url).validate().responseJSON { ... }`
- **Room (Android):** (Documentation)
  - *Why it's great:* **Compile-time verification** of SQL queries. It catches "Safety" issues (syntax errors) before the app even runs.

5. Summary Checklist for the Interview

1. **Define the Interface:** Write the code you *wish* you could write as a developer using your library. (Read-Eval-Print Loop approach).
2. **Choose the Scope:** Singleton (Global) vs. Instance (Scoped).
3. **Handle Lifecycle:** How does the client lifecycle affect your API? (Auto-cancellation).
4. **Mockability:** Can a user inject a test-double into your library for their own unit tests?

CHAPTER 5

RESTful API Design Deep Dive

This cheatsheet provides a concise list of design principles and mobile-specific optimizations for building robust RESTful APIs. In a system design interview, demonstrating these patterns shows your ability to build scalable systems that thrive under mobile constraints (high latency, limited bandwidth, and flaky reliability).

Your Goal: Proactively suggest these optimizations to show you prioritize the mobile user experience.

1. Resource-Oriented Design

REST APIs are based on resources (nouns), not actions (verbs).

- **Do Use Nouns, Not Verbs:** URIs should represent entities.
  - **Good:** `GET /users`, `POST /orders`
  - **Bad:** `GET /getUsers`, `POST /createOrder`
- **Do Use Plural Nouns:** Keep resource names consistent, typically plural.
  - `GET /users/123` (Read user 123)
  - `GET /users` (List all users)
- **Do Use Hierarchy:** Use nesting to indicate relationships, but avoid deep nesting.
  - **Good:** `GET /users/123/orders`
  - **Avoid:** `GET /users/123/orders/456/items/789` (Too deep; prefer `GET /orders/456/items`).
- **The Signal:** Shows you understand standard REST architectural patterns and can design intuitive, predictable interfaces.

2. HTTP Semantics (The Protocol Layer)

Use standard HTTP methods and status codes for their intended purpose.

2.1 HTTP Methods

Method	Action	Safety	Idempotency	Description
GET	Read	Safe	Yes	Retrieve data. Should never modify state.
POST	Create	Unsafe	No	Create a new resource. Not idempotent by default.
PUT	Replace	Unsafe	Yes	Replace a resource entirely.
PATCH	Update	Unsafe	No*	Partial update. (*Can be designed to be idempotent).

Method	Action	Safety	Idempotency	Description
DELETE	Delete	Unsafe	Yes	Remove a resource.

2.2 Standard HTTP Status Codes

Avoid inventing custom error codes. **Do** use standard status codes to control client flow.

- **2xx (Success):** `200 OK`, `201 Created` (for POST), `204 No Content` (for DELETE).
- **3xx (Redirection):** `301 Moved Permanently`, `304 Not Modified` (Critical for mobile caching).
- **4xx (Client Error):** `400 Bad Request`, `401 Unauthorized` (auth missing), `403 Forbidden` (auth present but insufficient), `404 Not Found`, `429 Too Many Requests`.
- **5xx (Server Error):** `500 Internal Server Error`, `503 Service Unavailable`.
- **The Signal:** Demonstrates you know how to leverage the HTTP protocol correctly, which is crucial for caching and intermediate proxies.

3. Mobile-Specific Optimizations (The "Mobile" Signal)

Demonstrate your mobile expertise by explicitly optimizing for the realities of cellular networks: limited bandwidth, high latency, and battery constraints.

3.1 Versioning

APIs evolve, but mobile apps are not updated instantly.

- **Strategy:** URI Versioning (`GET /v1/users`) is recommended for mobile.
- **Pros:** Explicit, easy to test, easy to cache.
- **Cons:** "Pollutes" the URI.
- **The Signal:** You understand that old app versions will exist in the wild for years and must continue to function.

3.2 Resource Management (Over-fetching vs. Under-fetching)

Finding the balance between "one request for everything" and "many small requests" is key.

- **Over-fetching:** Getting too much data.
  - *Solution:* **Field Selection**. Allow clients to specify exactly what they need: `GET /users/123?fields=id,name,avatar_url`.
- **Under-fetching:** Getting too little data, requiring subsequent requests (N+1 problem).
  - *Solution:* **Resource Embedding**. Allow clients to include related resources: `GET /orders/123?embed=items,payment_details`.

- **The "Backend for Frontend" (BFF) Pattern:** Ideally, mention creating a dedicated API layer specifically for the mobile app that aggregates data from multiple microservices into a single, mobile-optimized response.

3.3 Filtering, Sorting, and Searching

Keep the base resource URL clean and use query parameters.

- **Filtering:** `GET /users?role=admin`
- **Sorting:** `GET /users?sort=-created_at` (`-` for descending)
- **Searching:** `GET /users?q=alex`

4. Reliability & Resilience

4.1 Idempotency for Non-Idempotent Operations

Mobile networks are flaky. A `POST` request (e.g., "Pay $10") might time out. The user retries.

- **How to Generate Keys:** The client should generate a unique key (typically a UUID v4) for every critical mutation request.
- **Where to Include:** Pass this key in a custom HTTP header, e.g., `Idempotency-Key` or `X-Request-ID`.
- **Server Logic:**
  1. Check shared cache/DB for the key.
  2. If **Seen**: Return the cached response.
  3. If **New**: Execute logic and store result.
- **The Signal:** You know how to build resilient systems that handle network failures gracefully without duplicating transactions.

4.2 Rate Limiting

Protect your backend from abuse and spikes.

- **Response Headers:** Return `X-RateLimit-Remaining` and `X-RateLimit-Reset` so the app can back off intelligently.
- **Status Code:** Return `429 Too Many Requests`.

4.3 Date and Time

- **Do** always use **ISO 8601** strings: `2023-10-27T10:00:00Z`.
- **Do** always store and return dates in **UTC**. Let the mobile client handle local time formatting.

5. Security & Privacy

- **HTTPS:** Mandatory. No exceptions.
- **Authentication:** Use `Authorization: Bearer <token>` (e.g., JWT). Avoid passing tokens in URL parameters.
- **Input Validation:** Validate all input on the server side. Never trust the client.

5.1 API Key Placement

When using API keys to identify the client application:

- **Do Use Headers:** `X-API-Key`. Headers are less likely to be inadvertently logged than URLs.
- **Avoid Query Parameters:** `/resource?api_key=xyz`. URLs are logged in cleartext by proxies.
- **Mobile Reality:** Treat any key embedded in the app as **public**. Use them for rate limiting, not sensitive permissions.

5.2 Protecting Against Scraping

- **Rate Limiting:** Aggressive limiting per IP/User.
- **App Attestation:** Use Google Play Integrity / Apple App Attest to ensure requests come from your genuine app binary.
- **WAF:** Use a Web Application Firewall to block suspicious traffic patterns.

6. Error Handling

Return consistent, structured errors.

```json
{
  "error": {
    "code": "USER_ALREADY_EXISTS", // Machine-readable
    "message": "A user with this email already exists.", // Human-readable
    "details": [
      { "field": "email", "issue": "duplicate" }
    ]
  }
}
```

- **The Signal:** You prioritize the developer experience (DX). Structured errors allow the mobile app to parse specific issues and show localized, helpful UI messages.

7. Common Pitfalls ("Red Flags")

- **Tunneling:** Sending everything via `POST`. (Breaks caching).
- **Returning 200 OK for Errors:** Breaks HTTP-aware clients and monitoring tools. Use 4xx/5xx.
- **Verbs in URIs:** `POST /createOrder`. (Use `POST /orders`).

- **Chatty APIs:** Designing endpoints that require N+1 requests. Fix with Compound Documents.
- **Inconsistent Naming:** Mixing `camelCase` and `snake_case`. Pick one.
- **Leaking Internals:** Returning raw SQL exceptions to the client.
- **Missing Pagination:** Returning all records in a single response (`GET /users`). Always paginate lists.

8. Real-World Case Studies

- **Stripe (Idempotency):** Uses `Idempotency-Key` header to safely retry payment creation requests. (Source)
  - *Signal:* Mentioning this shows you understand financial-grade reliability.
- **Twitter (Versioning):** Uses distinct API versions (`/1.1/`, `/2/`) to allow radical architectural changes (like moving from Monolith to Microservices) without breaking clients. (Source)
- **GraphQL (Facebook):** While REST is standard, mentioning GraphQL for "Mobile-Specific Data Fetching" (solving over-fetching) shows breadth of knowledge. (Source)

9. Summary Checklist for the Interview

1. **Define Resources:** Identify the nouns (Users, Tweets).
2. **Select Methods:** Map actions to HTTP verbs (GET, POST).
3. **Optimize for Mobile:** Add Versioning, Pagination, and Partial Response (BFF).
4. **Handle Failure:** Define Error formats and Idempotency keys.
5. **Secure It:** HTTPS, Auth headers, Rate Limiting, and Attestation.

CHAPTER 6

Mobile Pagination Strategies & Deep Dive

In a system design interview, "How do you handle the feed?" is a guaranteed question for any content-heavy application. This cheatsheet outlines the strategies and trade-offs you should discuss to demonstrate your understanding of mobile-specific constraints.

Your Goal: Show that you can choose the **right architecture** for the specific user experience (Infinite Scroll vs. Page Navigation) while considering network, battery, and memory usage.

1. Step 1: Clarify the Use Case

The most common mistake candidates make is jumping straight to a solution (e.g., "I'll use cursors!") without defining the problem.

Why is this important? The user experience (UX) dictates the underlying engineering architecture. You cannot choose the right pagination strategy without knowing how the user interacts with the data.

- **Infinite Scroll** implies **Dynamic Data**. Users want to see *new* content without duplicates.
- **Static Pages** implies **Random Access**. Users want to jump to specific records (e.g., "Page 10 of Order History").

How it guides the solution:

- If you choose **Offset** for a Feed, users will see duplicate items when new content is posted (Page Drift).
- If you choose **Cursor** for a static list, users can't jump to specific pages, breaking expected navigation patterns.

Scenario A: Infinite Scroll (Dynamic Content Streams)

Recommendation: Cursor-Based Pagination

- **The "Why":** Users endlessly scroll down. Content is dynamic (items are created constantly).
- **The Problem it Solves:** "Page Drift".
  - *Scenario:* User loads Page 1 (Items 1-10). While reading, 5 new items are added to the stream. User requests Page 2.
  - *Offset Failure:* Page 2 would return Items 6-15. Items 6-10 are duplicates of what the user already saw!
  - *Cursor Success:* The cursor points to "Item 10". The server asks "Give me 10 items *after* Item 10". No duplicates, no missed items.
- **Mobile Benefit:** Smoother UX, no janky list updates to deduplicate items.

- **The Signal:** Demonstrates an understanding of data integrity (no duplicates) and a prioritization of user experience over simpler implementations. Shows an understanding of the eventual consistency of distributed systems.

Scenario B: Static Lists / Admin Panels (Order History, Logs)

Recommendation: Offset/Page-Number Pagination

- **The "Why":** User wants to jump to a specific record or date. "Go to Page 5". Data is relatively static.
- **The Mobile Benefit:** Familiar navigation patterns (Previous/Next buttons). Easier implementation.
- **The Signal:** Demonstrates a pragmatic approach that avoids over-engineering for simple problems.

Scenario C: Bi-directional Lists (Chat Apps)

Recommendation: Cursor-Based (Two-way)

- **The "Why":** Users jump to a specific message (e.g., from Search or Push Notification) and need to scroll *up* to see history and *down* to see newer messages.
- **The Problem:** Standard "Endless Scroll" only appends to the bottom. Chat requires prepending and appending.
- **The Signal:** Identifying this unique constraint distinguishes senior mobile engineers. It requires managing two cursors (`before_id` and `after_id`) and maintaining scroll position stability during prepend operations.

2. API Design: What the Signal Looks Like

Show the interviewer you know how to design a clean, robust API contract.

The Request

Avoid simply stating that a GET request is sent. **Do** specify parameters and explain default safety mechanisms.

```http
GET /v1/feed?cursor=dXNlcjpVMEc5V0ZXTjI=&limit=20
```

- `cursor` : The opaque token from the previous response.
- `limit` : **Crucial Signal:** Mention that you would enforce a **server-side max limit** (e.g., 100) to prevent a malicious client from crashing the server with excessive data requests.

The Response

Avoid returning a raw array. **Do** wrap data in an envelope with metadata.

```json
{
  "data": [ ... ], // List of item objects
  "pagination": {
    "next_cursor": "dXNlcjpVMEc5V0ZXTjI=", // Opaque string
    "has_more": true // Helps the client UI know whether to show a spinner
  }
}
```

Advanced Signal: HATEOAS (Hypermedia)

You might suggest including full navigation links (HATEOAS style) instead of just the cursor.

```json
"links": {
   "next": "https://api.example.com/v1/feed?cursor=dXNlcjpVMEc5V0ZXTjI=",
   "self": "https://api.example.com/v1/feed?cursor=..."
}
```

- **Pros:** The client doesn't need to know *how* to construct the next URL (loose coupling). If you change the parameter name from `cursor` to `token`, the client app doesn't break.
- **Cons:** Increases payload size (string bloat).
- **Verdict:** Mention it as a "pure REST" option, but acknowledge that for mobile, sending just the lightweight cursor is often preferred to save bandwidth.

Interview Tip: Explicitly mention "**Opaque Cursors**".

- *Interviewer:* "What's inside that string?"
- *You:* "It's a Base64 encoded string containing the ID/Timestamp of the last item. We encode it so clients don't try to perform math on it (like `id + 1`), allowing us to change the backend logic (e.g., switching from Integer IDs to UUIDs) without breaking the mobile app."

3. Mobile Client Considerations (The "Mobile" part of the interview)

This is where you differentiate yourself from a backend engineer by focusing on client-side resource constraints and UX patterns.

3.1 Prefetching & Thresholds

- **The Signal:** Demonstrates a proactive approach to loading content before the user reaches the end of the list.
- **Strategy:** Trigger the next page load when the user is **X items** (e.g., 5-10) away from the end.
- **Trade-off:** Too aggressive = wasted data (user might stop scrolling). Too lazy = user sees a spinner.

3.2 Network Reliability

- **Problem:** Request for Page 2 fails.
- **Solution:** Show a "Tap to Retry" footer. Do *not* automatically retry indefinitely (battery drain).

- **The Signal:** Demonstrates consideration for battery life and the graceful handling of failure states.

3.3 Memory Management & Recycling

- **Problem:** Infinite scrolling creates an infinitely growing list in memory.
- **Solution:**
  - **View Recycling:** Explain the pattern of reusing UI components. As a user scrolls, views that leave the screen are "recycled" and re-bound with new data for entering items. This ensures the memory footprint corresponds to the *screen size*, not the total *list size*.
  - **Diffing:** Discuss using background threads to calculate the difference between the old and new list states (Insertions, Deletions, Moves). This allows the UI thread to animate only the specific changes, preventing jank.
  - **Large Data Sets:** For massive lists (10k+ items), discuss "Windowing" - dropping off-screen pages from the backing data structure to prevent Out-Of-Memory (OOM) errors.

3.4 UX Patterns: Skeletons vs. Spinners

- **Problem:** How to indicate loading state for the initial page vs. subsequent pages.
- **Solution:**
  - **Initial Load:** Use **Shimmer/Skeleton** placeholders that mimic the cell layout. This reduces perceived latency compared to a generic spinner.
  - **Pagination:** Use a subtle **Spinner/Footer** at the bottom of the list.
- **The Signal:** You care about Perceived Performance and UI polish.

3.5 State Restoration (Process Death)

- **Question:** "What happens if the user backgrounds the app, the OS kills the process to save memory, and the user returns?"
- **Answer:**
  - The app must restore the **exact scroll position** and the **data** that was visible.
  - **Strategy:** Persist the current `cursor` and the `list_state` (scroll offset) to the system's saved state bundle. Re-fetch the data using the cursor or load from local persistence.
- **The Signal:** This is a deep mobile-specific cut that backend engineers will miss.

3.6 Caching & Offline Support

- **Question:** "How does pagination work offline?"
- **Answer:**
  - Persist the first X pages (e.g., 50 items) in a local database.
  - When offline, serve from the database.
  - When online, fetch fresh data, update the database, and notify the UI (Single Source of Truth pattern).

- **The Signal:** Demonstrates an understanding of "Offline-First" architecture and the creation of resilient applications.

3.5 Automatic Pagination (Auto-Pagination)

An advanced signal is discussing the abstraction of pagination logic through client libraries, often referred to as "Automatic Pagination."

- **The Concept:** Instead of the developer manually extracting the "next page token" and firing a new request, the client library treats the paginated resource as an **asynchronous stream or iterator**.
- **Reducing Boilerplate:**
  - **Token Management:** The library automatically captures the `next_page_token` from a response and populates the `page_token` in the subsequent request.
  - **Abstraction:** The developer simply iterates over the results (e.g., using a `for` loop or a stream observer); the library handles the network calls in the background.
- **Resource Management:**
  - **Lazy Resolution:** High-quality implementations are "lazy" - they only fetch the next page when the consumer reaches the end of the current buffer, preventing the "greedy" loading of massive datasets.
- **The Signal:** Demonstrates familiarity with high-level architectural patterns (AIP-4233) that reduce human error and improve code maintainability by hiding low-level network plumbing.

4. Common Pitfalls ("Red Flags")

- "**I'll just return all the data.**" -> Immediate fail. Shows lack of scalability awareness.
- "**I'll use `page` parameters for the real-time feed.**" -> Shows lack of awareness regarding real-time data shifts (Page Drift).
- "**I'll sort by Client Timestamp.**" -> Never trust the client clock. Always use Server Time.
- **Ignoring Empty States:** Always plan for the `{ "data": [] }` case.

5. Security & Privacy

- **Scraping:** Pagination makes it easy to scrape your entire database. Mention **Rate Limiting** (e.g., "Max 60 requests/minute").
- **Broken Access Control:** Ensure the user has permission to see *every* item in the page, not just the first one.
- **Predictable IDs:** If you use simple integers (`/users?after_id=50`), competitors can guess your growth rate. Opaque cursors hide this.
- **The Signal:** Demonstrates awareness of production risks and the importance of protecting business assets.

6. Real-World Case Studies

6.1 Twitter (High Velocity Feed)

- **Problem:** High-velocity writes cause massive "Page Drift" in offset systems.
- **Solution:** Cursor-based (`next_token`).

```http
// Request
GET /2/tweets/search/recent?
query=system+design&max_results=10&next_token=b26v89c19zqg8o3fo7ggj03...

// Response
{
  "data": [ ... ],
  "meta": {
    "result_count": 10,
    "next_token": "b26v89c19zqg8o3fo7ggj03..."
  }
}
```

- **Design Pattern:** Metadata envelope (`meta`) separates stream state from content, allowing for stable "anchors" in a constantly changing timeline.
- **Reference:** Twitter API Pagination

6.2 Slack (Deep History Performance)

- **Problem:** Scanning huge offsets (`OFFSET 50000`) is slow (O(N)).
- **Solution:** Cursor-based.

```http
// Request
GET /conversations.history?channel=C012345&cursor=dTx0...

// Response
{
  "ok": true,
  "messages": [ ... ],
  "has_more": true,
  "response_metadata": {
    "next_cursor": "bmV4dF9tZXNzYWdl..."
  }
}
```

- **Design Pattern:** Namespaced `response_metadata` ensures the pagination token is discoverable even when the `messages` array is empty.
- **Reference:** Slack API: conversations.history

6.3 Stripe (Financial Consistency)

- **Problem:** Financial reconciliation requires zero missed items.
- **Solution:** Strict cursors (`starting_after` , `ending_before`).

```http
// Request
GET /v1/charges?limit=10&starting_after=ch_12345

// Response
{
  "object": "list",
  "data": [
    { "id": "ch_12346", ... },
    { "id": "ch_12347", ... }
  ],
  "has_more": true,
  "url": "/v1/charges"
}
```

- **Design Pattern:** Uniform `object: list` structure allows client libraries to polymorphically automate pagination across all API resources.
- **Reference:** Stripe API Pagination

6.4 Reddit (Highly Dynamic Rankings)

- **Problem:** Content rank (votes) changes constantly during a session.
- **Solution:** Fullname IDs as cursors.

```http
// Request
GET /r/all/hot.json?limit=25&after=t3_15bfi0

// Response
{
  "kind": "Listing",
  "data": {
    "children": [ ... ],
    "after": "t3_16cfj1",
    "before": null
  }
}
```

- **Design Pattern:** Standardized "Listing" wrapper ensures consistent cursor placement (`after` / `before`) for every dynamic feed in the app.
- **Reference:** Reddit API Documentation

7. Summary Checklist for the Interview

1. **Clarify the Use Case:** Is it infinite scroll (Feed), static navigation (History), or bi-directional (Chat)?
2. **Choose the Strategy:** Cursor (Feed) vs. Offset (Static).
3. **Define the API:** Request params (cursor, limit) and Response structure (envelope, metadata).
4. **Add Mobile Context:** Prefetching, Error Handling, Offline Caching, and Process Death.
5. **Defend against Edge Cases:** Rate limiting, Empty states, Malformed inputs.

8. Further Reading

- **Slack Engineering:** Evolving API Pagination at Slack - A classic case study on why a major tech company migrated from offset to cursor-based pagination.
- **Twitter API Docs:** Pagination in the Twitter API - See how one of the world's largest feeds handles pagination in production.
- **Stripe API Reference:** Pagination - Widely considered the "Gold Standard" for developer-friendly cursor pagination.
- **Google AIP-158:** Pagination - Detailed design patterns for standardizing list methods.
- **Google AIP-4233:** Automatic Pagination - Standards for how client libraries should abstract pagination logic.
- **Nielsen Norman Group:** Infinite Scrolling vs. Pagination - The definitive UX research on when to use which pattern.

CHAPTER 7

Offline-First Architecture & Data Synchronization Deep Dive

In mobile system design interviews, "Offline Support" is often a dedicated section or a major non-functional requirement. Unlike web apps, mobile apps *must* assume the network is unreliable.

Your Goal: Demonstrate how to architect an app that works seamlessly without internet, syncs efficiently when connectivity returns, and handles data conflicts gracefully.

1. Core Philosophy: The Single Source of Truth

The most critical architectural decision is defining the "Single Source of Truth" (SSOT).

The "Online-First" Mistake

Many candidates design apps that fetch data from the network and display it directly in the UI.

- **Problem:** If the network fails, the screen is empty. If the user navigates away and back, they wait for a loader again.
- **Result:** Poor UX and high data usage.

The "Offline-First" Solution (Repository Pattern)

The UI *only* observes the **Local Database**.

1. **Read:** UI subscribes to the Local DB (e.g., Room `Flow`, CoreData `NSFetchedResultsController`).
2. **Write:** User actions update the Local DB immediately.
3. **Sync:** A background "Sync Engine" synchronizes the Local DB with the Remote API.

The Signal: This decouples the UI from the Network. The app feels instant because it's reading from local disk, regardless of network latency.

2. Synchronization Strategies

How do you keep the Local DB and Remote Server in sync?

2.1 Full Sync vs. Delta Sync

- **Full Sync:** Download the entire dataset every time.
  - *Pros:* Simple to implement. Guaranteed consistency.
  - *Cons:* High bandwidth, slow, battery drain. Only acceptable for tiny datasets (e.g., User Settings).

- **Delta Sync (Incremental Sync):** Download only what changed since the last sync.
  - *Mechanism:* The client sends a **sync marker** to the server. The server returns only records modified after that point.
  - **Option A:** `last_synced_timestamp`
    - *Flow:*
      1. Client fetches all data. Server returns current server time (e.g., `2025-12-19T10:00:00Z`).
      2. Client saves this timestamp locally.
      3. Next sync, client requests: `GET /sync?since=2025-12-19T10:00:00Z`.
      4. Server queries DB for `updated_at > since`.
    - *Risk:* Vulnerable to **Clock Skew**. If server instances have different times, or an update happens in the exact same millisecond as the sync, data might be missed.
  - **Option B:** `sync_token` **(The "Opaque Cursor") - Recommended**
    - *Definition:* A string or number generated by the server (e.g., `v2_seq_98765`) that acts as a bookmark. The client stores it blindly without interpreting it.
    - *Flow:*
      1. **Response:** Server returns data + token: `{"data": [...], "sync_token": "v2_seq_98765"}`.
      2. **Request:** Next sync, client sends it back: `GET /sync?token=v2_seq_98765`.
      3. **Server Logic:** Server decodes the token (e.g., maps it to a Global Sequence ID) and returns newer items.
    - *Benefit:* **Stateless & Robust.** Avoids clock skew entirely by using monotonic sequence IDs. Allows the backend to change versioning logic without breaking the mobile app.
  - *Pros:* Efficient, fast, saves battery.
  - *Cons:* Complex backend logic (requires "Soft Deletes" to sync deletions).

2.2 Sync Direction

- **Pull (Down-sync):** Fetching updates from the server to the device.
- **Push (Up-sync):** Sending local changes (pending writes) to the server.

2.3 Advanced Sync Patterns

Operation Log (Event Sourcing)

- **The Concept:** Instead of syncing the *current state* (e.g., "Note Title is 'Groceries'"), you sync the *list of changes* (e.g., "User A changed title to 'Groceries'").
- **How it works:**
  - The server keeps an append-only log of all mutations.
  - The client says "Give me all operations starting from Offset 100."
  - The client "replays" these actions locally.

- **Pros:** Preserves intent (e.g., distinguishing between "User set value to 0" and "User decremented value"). Easier to resolve conflicts.
- **Cons:** If the client is very old (Offset 0), replaying the entire history is slow. Requires "Snapshots" to fix.
- **Read More:** Martin Fowler on Event Sourcing

Merkle Trees (Hash Trees)

- **The Concept:** Instead of tracking *when* data changed, you track the *signature* of the data.
- **How it works:**
  1. Both the Client and Server organize their data records into a tree structure.
  2. Each leaf node is a hash of a data record. Each parent node is a hash of its children.
  3. **The Sync Protocol:** The client sends the "Root Hash" of its tree to the server.
  4. **Comparison:** If `Client.RootHash == Server.RootHash`, they are perfectly in sync (0 bytes transferred). If different, the server compares children hashes to traverse down and pinpoint specifically *which* record is out of sync.
- **Use Case:** Blockchain, Git, Cassandra, and complex file syncing (like Dropbox).
- **Pros:** Extremely bandwidth-efficient for verifying consistency of large datasets.
- **Cons:** High computational cost (hashing) and complexity to maintain the tree.
- **Read More:** Merkle Tree (Wikipedia)

Version Vectors (Vector Clocks)

- **The Concept:** Instead of a single "Server Time", we track logical counters for *every* actor (device) that modifies data.
- **How it works:**
  - State is tracked as `[DeviceA: 5, DeviceB: 3, Server: 10]`.
  - This allows the system to distinguish between "Device A hasn't seen Device B's update" vs "Device A *overwrote* Device B's update."
- **Use Case:** Peer-to-Peer systems or truly distributed offline-first apps where devices might sync directly with each other (rare in typical mobile interviews, but good for "Signal").
- **Read More:** Vector Clocks (Wikipedia)

Hash-Based "Check-In"

- **The Concept:** A simplified "Merkle Tree" for a quick sanity check.
- **How it works:**
  - Client calculates a single hash of its entire dataset (e.g., `md5(all_ids + timestamps)`).
  - Client sends this hash to the server.
  - Server compares it with its own calculation. If match -> Done. If mismatch -> Trigger full sync or standard delta sync.
- **Pros:** Very easy to implement. Great for verifying consistency after a series of complex delta syncs.
- **Cons:** Requires hashing the entire dataset, which can be slow for large databases.

2.4 Soft Deletes

You cannot physically delete a row on the server in a Delta Sync system, because the client won't know it's gone.

- **Solution:** Use a `is_deleted` (tombstone) column.
- **Flow:**
  1. Server marks item as `is_deleted = true`.
  2. Client requests changes since `T`.
  3. Server sends the "deleted" item.
  4. Client sees the flag and removes it from the Local DB (or keeps it hidden if undo is allowed).

3. Handling Local Writes (The "Pending" Queue)

When a user performs an action (e.g., "Like Tweet") while offline:

1. **Optimistic Update:** Immediately update the UI to show the "Like" state (red heart).
2. **Persist Action:** Store the action in a **Persistent Queue** (not just memory).
   - *Why Persistent?* If the app is killed before the network returns, the action must not be lost.
3. **Background Sync:** When the network returns (via `WorkManager` on Android or `BackgroundTasks` on iOS), process the queue.
   - *Success:* Remove item from queue.
   - *Failure (Transient):* Retry with Exponential Backoff.
   - *Failure (Permanent):* Remove from queue and notify user (e.g., "Could not like tweet"). Revert the Optimistic Update.

4. Conflict Resolution Strategies

This is the hardest part of offline architecture. What happens if the user edits a note offline, but someone else edits the same note on the server?

Strategy A: Last Write Wins (LWW)

- **Logic:** The system looks at the timestamp. The most recent update overwrites the other.
- **Pros:** Easy to implement.
- **Cons:** Data loss. (If I edit offline at 10:00, and you edit online at 10:05, my changes are wiped out when I eventually sync).

Strategy B: Server Authority (The "Git Push -f" approach)

- **Logic:** The server's version is the truth. If the client tries to upload a stale version, the server rejects it (HTTP 409 Conflict).
- **Client Handling:** The client must download the new server version and ask the user what to do.

- **Pros:** Safe, prevents silent data loss.
- **Cons:** Annoying UX ("Conflict detected, please resolve").

Strategy C: Field-Level Merging

- **Logic:** Merge non-conflicting fields automatically.
- **Example:** User A updates `Title`. User B updates `Description`. Both changes are kept.
- **Pros:** Reduces conflicts significantly.

Strategy D: CRDTs (Conflict-Free Replicated Data Types)

- **Logic:** specialized data structures designed to *always* merge successfully mathematically.
- **Use Case:** Collaborative text editors (Google Docs), counters.
- **The Signal:** Mentioning CRDTs shows deep theoretical knowledge, but acknowledge they are complex to implement from scratch.

5. Mobile-Specific Components

- **Database:**
  - **Android:** Room (SQLite wrapper). Strongly typed, observable.
  - **iOS:** Core Data (Object Graph) or SwiftData.
  - **Cross-Platform:** Realm (NoSQL, easy sync), SQLite (Raw).
- **Job Schedulers:**
  - **Android:** `WorkManager`. The gold standard. Handles constraints (e.g., "Run only when on WiFi and Charging").
  - **iOS:** `BGAppRefreshTask` / `BGProcessingTask`. stricter limitations on execution time.

8. Real-World Case Studies

- **Trello:** (Engineering Blog)
  - *Strategy:* A complex "Command Queue" system that replays user actions.
  - *Signal:* Demonstrates how to handle "optimistic UI" with potentially thousands of offline edits.
- **Linear:** (Engineering Blog)
  - *Strategy:* "Sync Engine" that treats the local database as a cache of the entire dataset.
  - *Signal:* Shows how high-performance apps prioritize local-first reads for speed.
- **CouchDB / PouchDB:** (Website)
  - *Strategy:* Replication protocols built into the database layer.
  - *Signal:* Understanding "Replication" vs. "Custom Sync" trade-offs.

9. Summary Checklist for the Interview

1. **Define SSOT:** "I will use the Repository Pattern with a local database as the Single Source of Truth."
2. **Define Sync Strategy:** "I will implement Delta Sync using a `last_updated` cursor to minimize bandwidth."
3. **Handle Offline Writes:** "I will use a persistent operation queue and `WorkManager` to flush changes when connectivity returns."
4. **Address Conflicts:** "For this use case, [Last Write Wins / User Prompt] is appropriate because..."
5. **Mention UX:** "I will use Optimistic Updates to make the app feel responsive."

10. Common Pitfalls ("Red Flags")

- **"I'll use a boolean `isOffline` flag."** -> Bad code smell. Avoid building separate logic paths. Always write to DB, let the Sync Engine handle the rest.
- **In-Memory Queues:** -> Data loss if the app crashes. Always persist pending actions.
- **Infinite Retries:** -> Battery drain. Always use Exponential Backoff and jitter.
- **Blocking the UI:** -> Database and Network operations must happen on background threads.

11. Further Reading

- **Trello:** Sync Architecture - Excellent breakdown of a complex sync engine.
- **Linear:** Sync Engine - How a high-performance app handles real-time sync.
- **Google:** Offline-First Guide - Official Android architectural guidance.

CHAPTER 8

Caching Deep Dive

Caching is a fundamental concept in system design, crucial for improving performance, reducing latency, and ensuring offline capability. In mobile development, it bridges the gap between the fast, reliable local environment and the slow, unreliable network.

1. Why Cache?

- **Offline Support:** Allows users to interact with the app without an internet connection.
- **Performance:** Loading data from local storage is orders of magnitude faster than network requests.
- **Cost Efficiency:** Reduces data usage for the user and backend load for the server.
- **Battery Life:** Radios are battery-hungry; fewer requests mean longer battery life.

2. Caching Layers

Memory Cache (L1)

The fastest access, but volatile. Data is lost when the app is killed.

- **Use Case:** Bitmaps (Images) currently being displayed, frequently accessed small objects (Configuration).
- **Implementation:** `LruCache` (Android), `NSCache` (iOS).
- **Constraints:** Strictly limited by available RAM. Must handle low-memory warnings to avoid OOM (Out Of Memory) crashes.

Disk Cache (L2)

Slower than memory, but persistent. Data survives app restarts.

- **Use Case:** JSON responses, large images, video chunks, user profile data.
- **Implementation:**
  - **Structured:** SQLite (Room), Core Data, Realm. Good for complex queries and relationships.
  - **Unstructured/Key-Value:** DiskLruCache, DataStore (Android), simple File I/O. Good for raw blobs or simple preferences.
  - **HTTP Cache:** Standard HTTP caching handled by networking clients (OkHttp, URLSession).

3. Caching Strategies

Cache-Aside (Lazy Loading)

The application code is responsible for loading data.

1. App asks Cache for data.
2. **Hit:** Cache returns data.
3. **Miss:** App fetches data from API, writes to Cache, then returns data.

Read-Through / Write-Through

The application treats the cache as the main data source. The cache library handles the fetching logic.

- **Repositories:** In mobile architecture, the Repository pattern often acts as this layer.

Stale-While-Revalidate

Return stale data immediately from the cache (for UI responsiveness) while simultaneously fetching fresh data from the network in the background to update the view.

4. Cache Eviction Policies

Since storage is finite, we must decide what to delete when full.

- **LRU (Least Recently Used):** Discards the least recently used items first. Most common for images and general data.
- **FIFO (First In First Out):** Discards the oldest items first, regardless of usage.
- **TTL (Time To Live):** Data expires after a specific time period (e.g., 5 minutes). Critical for volatile data like stock prices or news feeds.

5. HTTP Caching

Leverage standard headers to let the server control caching behavior.

- `Cache-Control`: `max-age=3600` (valid for 1 hour), `no-cache` (must revalidate), `no-store` (never cache).
- `ETag` **(Entity Tag):** A fingerprint of the resource. Client sends `If-None-Match: "tag"`. Server returns `304 Not Modified` (empty body) if data hasn't changed, saving bandwidth.
- `Last-Modified`: Client sends `If-Modified-Since`. Similar to ETag but time-based.

6. Security Considerations

- **Sensitive Data:** Never cache PII (Personally Identifiable Information), auth tokens, or payment details in plain text on disk.
- **Encryption:** On Android, prefer `Keystore` for encryption key management. On iOS, use `Keychain` / `File Protection`.
- **Snapshots:** Be aware that the OS might take snapshots of the UI (which includes cached data) for the app switcher. Mask sensitive fields.

7. Common Pitfalls

- **Over-Caching:** Caching everything fills up disk space and complicates debugging.
- **Invalidation Logic:** The hardest problem in computer science. How do you know when to clear the cache? (User logout, explicit pull-to-refresh, push notification triggers).
- **Race Conditions:** Reading/Writing to cache from multiple threads.

CHAPTER 9

Quality Of Service (QoS)

Implementing Quality of Service (QoS) for network operations in mobile apps is critical to providing a smooth user experience while conserving device resources, especially battery life. QoS involves prioritizing network requests based on their importance to the user and the app's functionality.

- **Limit Concurrent Network Operations:** Restricting the number of simultaneous network requests is crucial for several reasons. Each active network connection consumes system resources (CPU, memory, radio), impacting performance and battery life. Aim for a limit of 4-10 concurrent operations, adjusting based on device state (battery level, charging status, network connectivity) and app usage patterns. Consider the network bandwidth available.
- **Adaptive Concurrency:** Monitor network conditions (Wi-Fi vs. cellular, signal strength) and adjust the concurrency limit dynamically. Reduce the limit on cellular networks or when the signal is weak.
- **Network Request Prioritization:** Assign a QoS class to each network request based on its importance. This allows the app to prioritize essential requests while deferring less critical ones.
- **QoS Classes and Examples:**
  - **User-Critical (High Priority):** Requests directly impacting the user's immediate experience and require minimal latency.
    - Fetching the next page of data for the Tweet Feed (infinite scrolling).
    - Requesting Tweet Details (when a user taps on a tweet).
    - Sending a direct message.
    - Authentication requests.
  - **UI-Critical (Medium Priority):** Requests contributing to the user interface's responsiveness, but can tolerate slightly higher latency.
    - Fetching low-resolution placeholder images for tweets while scrolling.
    - Loading thumbnails or previews.
    - Non-blocking UI updates.
  - **UI-Non-Critical (Low Priority):** Requests enhancing the UI but are not essential for its core functionality. Can be delayed or canceled if necessary.
    - Fetching high-resolution images for tweets on the Feed.
    - Preloading images for tweets that are not yet visible.
    - Downloading non-essential resources.

    - **Background (Lowest Priority):** Requests performed in the background without directly impacting the user experience. Can be deferred until the device is idle or charging.
        - Posting "likes" or comments.
        - Uploading analytics data.
        - Backing up data.
        - Syncing data with the server.

- **Priority Queue Scheduling:** Implement a priority queue to manage network requests based on their QoS class. Dispatch requests in the order of their priority, ensuring that user-critical requests are processed first.
- **Request Suspension and Resumption:** If the maximum number of concurrent operations is reached, suspend lower-priority requests to allow higher-priority requests to proceed. Resume suspended requests when resources become available. Use cancellation tokens to allow for quickly cancelling long-running tasks if needed.
- **Cancellation Support:** Implement cancellation support for network requests to prevent unnecessary data transfer and resource consumption. Cancel requests when they are no longer needed (e.g., when the user scrolls past a tweet).
- **Network Request Throttling:** Implement throttling mechanisms to prevent the app from overwhelming the network with too many requests. Use techniques such as exponential backoff and rate limiting.
- **Device State Awareness:** Consider the device's current state when scheduling network requests. Defer low-priority requests when the device is on battery or has a weak network signal.
- **Power Management Optimization:** Minimize the number of wake locks held by the app to prevent battery drain. Use the JobScheduler (Android) or BackgroundTasks framework (iOS) to schedule background tasks efficiently.
- **Network Change Monitoring:** Monitor network connectivity changes and adapt the app's behavior accordingly. Pause or cancel network requests when the device loses connectivity.
- **Error Handling and Retries:** Implement proper error handling and retry mechanisms for network requests. Use exponential backoff to avoid overwhelming the server with retries.
- **Testing and Monitoring:** Thoroughly test the app's network behavior under different network conditions and device states. Use network monitoring tools to identify performance bottlenecks and optimize network requests. Implement logging and analytics to track network usage and identify potential issues.

By implementing these QoS techniques, you can significantly improve the performance, responsiveness, and battery life of your mobile app, providing a better experience for your users.

CHAPTER 10

Resumable Uploads (Chunked Uploads)

Resumable uploads, also known as chunked uploads, break down a large file upload into smaller, manageable chunks. This technique allows the upload to be paused and resumed without losing progress, making it ideal for situations where network connectivity is unreliable or the upload size is significant. This also allows for more granular error reporting and potentially, faster recovery.

The resumable upload process typically involves these stages:

- **Upload Initialization:** The client initiates the upload by sending a request to the server, providing metadata about the file (name, size, content type). The server creates a temporary storage location and returns a unique upload ID or URL to the client.
- **Chunk Bytes Uploading (Appending):** The client divides the file into smaller chunks (e.g., 1MB, 5MB) and uploads each chunk separately to the server using the upload ID or URL. The client sends each chunk with a specific offset indicating its position in the overall file.
- **Upload Finalization:** Once all chunks have been successfully uploaded, the client sends a finalization request to the server. The server assembles the chunks into the complete file and performs any necessary processing (e.g., validation, transcoding).

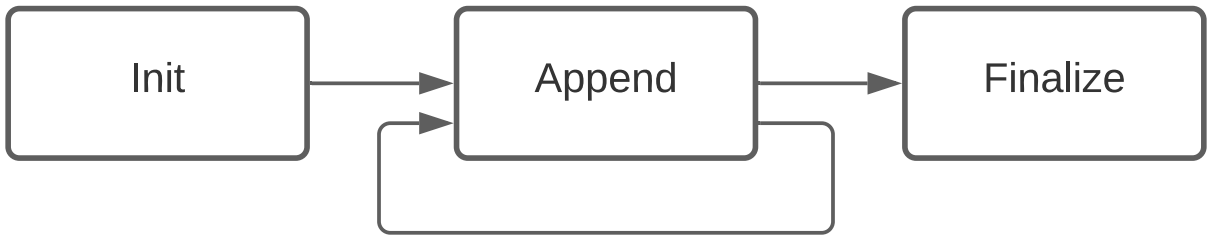


FIGURE 10.1 Resumable-Uploads

Advantages of Resumable Uploads:

- **Allows you to resume interrupted data transfer operations without restarting from the beginning:** This is the primary advantage, especially crucial for users with unreliable network connections or large file uploads. Saves time, bandwidth, and improves the user experience.
- **Improved Reliability:** By dividing the upload into smaller chunks, it reduces the impact of network interruptions. If a chunk fails to upload, only that chunk needs to be retransmitted, not the entire file.
- **Bandwidth Optimization:** Resuming from a specific point avoids resending already transferred data, conserving bandwidth.
- **Progress Monitoring:** Allows for more accurate and granular progress tracking during the upload process, providing users with better feedback. This also allows for better prioritization of bandwidth within the device.

- **Parallel Uploading (Potentially):** Some implementations allow for uploading multiple chunks in parallel, further speeding up the upload process (however, this can increase complexity). This requires server-side support for out-of-order chunk arrival and assembly.
- **Improved Error Handling:** Easier to pinpoint and handle errors at a chunk level rather than the entire upload.
- **Resource Management:** Can help manage server-side resources more effectively by receiving and processing data in smaller increments.
- **Suitable for Large Files:** Essential for uploading files exceeding size limits imposed by HTTP servers or mobile operating systems.

Disadvantages of Resumable Uploads:

- **An overhead from additional connections and metadata:** Each chunk requires a separate HTTP request, adding overhead to the overall upload process. Requires careful management of HTTP headers.
- **Increased Complexity:** Implementing resumable uploads requires more complex client-side and server-side logic compared to single-request uploads.
- **Server-Side Requirements:** Requires server-side support for handling chunked uploads, storing temporary data, and assembling the complete file.
- **State Management:** Both client and server need to maintain state about the upload process (e.g., upload ID, chunk offsets), which can add complexity.
- **Error Handling:** Requires robust error handling to deal with network interruptions, server errors, and other potential issues.
- **Potential for Data Corruption:** If chunks are not assembled correctly, it can lead to data corruption. Requires careful validation and integrity checks.
- **Storage Requirements:** The server needs temporary storage space to hold the uploaded chunks until the upload is finalized.

Additional Considerations:

- **Chunk Size:** The optimal chunk size depends on network conditions, file size, and server capabilities. Experiment with different chunk sizes to find the best balance between performance and overhead.
- **Encryption:** Encrypt the uploaded chunks to protect sensitive data in transit and at rest.
- **Concurrency:** Consider the number of concurrent uploads allowed to prevent overwhelming the device or server.
- **Timeout Handling:** Implement timeouts to handle stalled uploads.
- **Background Uploads:** If the upload can be interrupted (app closed), consider using background upload tasks with OS-level support (WorkManager/BackgroundTasks)
- **Idempotency:** Ensure that resuming a failed upload does not result in duplicate data on the server. Use unique identifiers for each chunk.

> **Note:** Resumable uploads are most effective with large uploads (videos, archives) on unstable networks. For smaller files (images, texts) and stable networks, a single-request upload should be sufficient. Carefully weigh the benefits and drawbacks before implementing resumable uploads. If targeting a specific cloud storage provider, utilize their SDK which may provide simplified upload mechanisms.

More Info:

- YouTube: Resumable Uploads
- Google Photos: Resumable Uploads
- Google Cloud: Resumable Uploads
- Twitter: Chunked Media Upload

CHAPTER 11

Prefetching

Prefetching is a technique used to improve application performance by proactively fetching data or resources that are likely to be needed in the future. By loading data in the background before it's explicitly requested, prefetching can significantly reduce latency and enhance the user experience. This creates the illusion of instant loading times and a more responsive application.

Prefetching can be applied to various types of content, including:

- **Data:** Pre-loading data for upcoming screens or features.
- **Images:** Caching images likely to be viewed by the user.
- **Code:** Downloading code modules or assets for features that might be used soon.
- **Configuration:** Fetching configuration updates in advance.

Types of Prefetching:

- **Predictive Prefetching**: Uses machine learning or heuristics to predict which data or resources the user will need based on their past behavior and patterns.
- **Cache-Aware Prefetching**: Takes into account what is already cached and prefetches only what is missing, optimizing bandwidth usage.
- **Just-In-Time (JIT) Prefetching**: Preloads resources just before they are needed. E.g. when the user scrolls close to the end of the page (but before making a request), prefetch the next page.

Advantages of Prefetching:

- **Improved Performance:** Reduced latency and faster loading times.
- **Enhanced User Experience:** Smoother and more responsive application.
- **Increased User Engagement:** Users are more likely to interact with an app that feels fast and fluid.
- **Hides network latency:** Creates a better user experience, even on slower connections.

Disadvantages of Prefetching:

- **Increased Bandwidth Usage:** Unnecessary prefetching can consume bandwidth, especially on metered connections.
- **Increased Battery Consumption:** Downloading data in the background can drain battery life.
- **Increased Storage Consumption:** Caching prefetched data can consume device storage.
- **Potential for Stale Data:** Prefetched data may become stale if it's not updated frequently.
- **Resource Wastage:** Resources might be fetched that are never actually used, wasting bandwidth and battery.
- **Complex Implementation:** Requires careful planning and implementation to avoid negative impacts.

Considerations for Implementing Prefetching:

- **Network Conditions:** Adapt prefetching behavior based on network connectivity (Wi-Fi vs. cellular). Disable prefetching on metered connections or when the signal is weak.
- **Device State:** Consider the device's battery level and charging status. Disable prefetching when the battery is low.
- **User Behavior:** Use analytics to track user behavior and identify patterns. Prefetch content that is most likely to be needed.
- **Data Staleness:** Implement a mechanism for invalidating and refreshing prefetched data to ensure that it's up-to-date. Use ETags to avoid downloading the same content repeatedly.
- **Cache Management:** Implement a cache eviction policy to remove less frequently used data and prevent excessive storage consumption. Use Least Recently Used (LRU) or Least Frequently Used (LFU) algorithms.
- **QoS (Quality of Service):** Prioritize prefetching requests based on their importance. Defer prefetching tasks when user-critical requests are in progress.
- **User Control:** Provide users with control over prefetching settings. Allow them to disable prefetching or adjust the amount of data that is prefetched.
- **Testing and Monitoring:** Thoroughly test the app's prefetching behavior under different network conditions and device states. Monitor network usage, battery consumption, and storage consumption to identify potential issues.
- **Respect User Privacy:** Only prefetch data that is relevant to the user's current activity. Avoid prefetching sensitive or private data without explicit consent.

Mitigating Negative Impacts:

- **Use ETags**: Send conditional GET requests with `If-None-Match` to check if the prefetched content has been updated. This avoids downloading the same content repeatedly if it hasn't changed.
- **Adaptive Prefetching**: Monitor network conditions and device state and adjust prefetching behavior accordingly.
- **Time-Based Invalidation**: Set a time-to-live (TTL) for prefetched data and invalidate it after a certain period to prevent staleness.
- **User-Initiated Refresh**: Provide a way for users to manually refresh the data if they suspect it's stale.
- **Stop on User Action**: If the user performs an action that indicates they no longer need the prefetched content, stop the prefetching process immediately.

More Info:

- Optimize downloads for efficient network access
- Improving performance with background data prefetching
- Informed mobile prefetching
- Understanding prefetching and how Facebook uses prefetching

PART III

Interview Exercises

Worked mock-interview exercises that apply the framework to typical library- and app-design questions.

IN THIS PART

CHAPTER 12

Typical Mobile System Design Interview Questions

The following exercises emulate typical interview scenarios using common system design questions. The solution to each question might not be accurate and should not be taken as the "ground truth". The purpose of the exercises is to demonstrate different interviewer-candidate interactions.

- Design "File Downloader Library"
- Design "Caching Library"
- Design "Image Library"
- Design "Location Sharing Library"
- Design "Network Library"
- Design "Json Parsing Library like Moshi/Gson"
- Design "Pagination Library"
- Design "Analytics Library"
- Design "A/B Experiment Library"
- Design "Photo App"
  - upload / sync photos with backend
- Design "Chat App"
- Design "Clock App"
- Design "Recipe App"
- Design "Photo Editing App"
- Design "Live Wallpaper Display App"
- Design "Instagram Post Feed/Facebook News Feed"
- Design "Facebook/Instagram story"
- Design "Flight booking system"
  - How to cache status
  - How to handle case like leaving a page and coming back without previous information loss
  - How to handle transaction failure
  - How to lock seat selection
- Design "Contact app with real time status"
- Design "Push notifications system"
  - broadcast receiver + notification manager + persistent connection
- Design "File uploader library"

Feel free to suggest your own topics.

CHAPTER 13

Mobile System Design Exercise: File Downloader Library (iOS & Android)

The task is defined as "Design a File Downloader Library." The definition is purposely vague and requires clarification. This guide aims to help you navigate this type of system design question, tailored for both iOS and Android platforms, using modern APIs.

Gathering Requirements

Clarifying requirements is crucial. Avoid proceeding to the solution before establishing a deep understanding of the problem. Here are some example questions and considerations for iOS and Android:

> **Candidate**: "Are we designing a part of an application or a general-purpose library?"
> **Interviewer**: "A general-purpose library."

> **Candidate**: "Are we downloading a file from the internet and saving it to the disk?"
> **Interviewer**: "Yes."

> **Candidate**: "What kind of files do we need to download? Any size constraints?"
> **Interviewer**: "Any kind-let's assume that we're working with binary files without specific format and no size limit."

> **Candidate**: "Should we support pausing/resume/canceling/listing downloads?"
> **Interviewer**: "Yes."

> **Candidate**: "Do we need to support simultaneous file downloads?"
> **Interviewer**: "I don't know-what do you think?"
> **Candidate**: "I think, there might be some good use-cases where simultaneous downloads would be useful. For example, downloading a video playlist."

> **Candidate**: "Should we limit the number of active simultaneous downloads?"
> **Interviewer**: "I don't know-what do you think?"
> **Candidate**: "I think having unrestricted parallel downloads might hurt application performance and quickly exhaust system resources (sockets, file handles). Also, without limits, we open ourselves up to DOS-style attacks or simply choking the bandwidth. A sensible default might be `4`, but it should be configurable."

> **Interviewer**: "Why do you think `4` is a good number for parallel downloads?"
> **Candidate**: "It's a heuristic. On mobile, we are often constrained by the network radio. Opening too many TCP connections can increase overhead. However, since these are I/O bound, we aren't limited by CPU cores. `4` is a conservative starting point to ensure we don't block the app's other network requests (like API calls)."

> **Candidate**: "What about network constraints? Should we allow downloads over Cellular, or restricted to WiFi?"
> **Interviewer**: "Good point. Let's make it configurable per download or globally."

> **Candidate**: "Do we need to ensure file integrity? e.g., verifying a checksum after download?"
> **Interviewer**: "Yes, that would be a great feature for a robust library."

> **Candidate**: "How robust should the persistence be? If the app crashes or is killed by the OS, should the downloads resume automatically next launch?"
> **Interviewer**: "Yes, they should survive app termination."

> **Candidate**: "Should we support progress reporting for active downloads?"
> **Interviewer**: "Might skip it for now and discuss it if we have time"

> **Candidate**: "Do we need to handle authentication?"
> **Interviewer**: "Let's skip it."

> **Candidate**: "Do we need to support HTTP ranges for resumable downloads?"
> **Interviewer**: "We can leave it out of scope"

Platform Specific Considerations:

- **iOS:**
  - **Background Download Considerations:** Ask about background download requirements early. iOS has strict limitations on background tasks. Using `URLSessionConfiguration.background(withIdentifier:)` allows downloads to continue even when the app is suspended, but it comes with limitations (e.g., the system decides when the download occurs). Consider the implications of using `BackgroundTasks.framework` for scheduled background tasks.
  - **File System Access:** Understand iOS's sandboxed file system. Know the common directories (`Documents`, `Library/Caches`, `tmp`) and their appropriate uses.

- **Android:**
  - **Background Download Considerations:** Android's background execution limitations have evolved. Consider using `WorkManager` for background tasks. `WorkManager` is a more general-purpose solution for deferrable, guaranteed execution tasks.
  - **DownloadManager**: While Android's `DownloadManager` is an option for background downloads, its usage is often discouraged due to potential security concerns and limited control over the download process. It can be prone to vulnerabilities related to file path injection and unintentional overwriting of existing files. Consider carefully whether the benefits outweigh the risks before using it.
  - **Permissions:** Android requires explicit permissions for accessing external storage. Ensure the design accounts for requesting and handling these permissions (`READ_EXTERNAL_STORAGE` , `WRITE_EXTERNAL_STORAGE` for older Android versions; scoped storage considerations for newer versions).
  - **Power Management:** Android has aggressive power-saving features (Doze mode, App Standby Buckets). Understand how these might impact background downloads and consider strategies to mitigate issues (e.g., using `WorkManager` with appropriate constraints).

Make sure not to overload the system requirements with unnecessary features. Think in terms of MVP (Minimum Viable Product) and pick features that have the biggest value. You can learn more about requirements gathering here.

Functional requirements

- Developers should be able to simultaneously download multiple files over HTTP to the disk.
- Developers should be able to pause/resume/cancel downloads.
- Developers should be able to list active downloads.

Non-functional requirements

- Limited active simultaneous downloads.
- Unlimited download file size.

Out of scope

- Login/Authentication.
- Resumable HTTP-downloads.

Client Public API (Optional)

Discussing the client-facing API helps demonstrate how developers will interact with the library. The example provided is a good starting point. Consider how it translates to platform-specific conventions:

```
FileDownloader:
+ init(config: FileDownloaderConfig)
+ download(request: FileDownloadRequest): FileDownloadTask
+ pauseAll()
+ resumeAll()
+ cancelAll()
+ activeTasks(): [FileDownloadTask]

FileDownloadRequest:
+ init(sourceUrl: Url, destPath: String)

FileDownloadTask:
+ addDownloadCallback(callback: FileDownloadCallback)
+ pause()
+ resume()
+ cancel()

FileDownloaderConfig:
+ init(maxParallelDownloads: Int)

FileDownloadCallback:
+ onComplete(request: FileDownloadRequest)
+ onFail(request: FileDownloadRequest, error: String)
+ onCancel(request: FileDownloadRequest)
```

Platform Specific Considerations:

- **iOS (Swift):**
  - Use Swift's concurrency features (async/await) for cleaner asynchronous code.
  - Leverage `Result` type for handling success/failure cases in callbacks.
  - Consider using Combine framework for reactive programming and managing download state.
- **Android (Kotlin):**
  - Use Kotlin coroutines for asynchronous operations.
  - Use `sealed class` or `Result` for representing success/failure outcomes.
  - Consider using Kotlin Flow (similar to Combine) for reactive data streams.
  - Leverage Android's `LiveData` or `StateFlow` for observing download progress and status updates in UI.

API Design Notes:

- **Error Handling:** Use specific error types in `onFail` callback to provide more context.
- **Progress Reporting:** Consider a `onProgress` callback to report download progress updates. Throttling these updates is crucial to avoid excessive UI updates.
- **Configuration:** The `FileDownloaderConfig` should include options for:
  - Maximum concurrent downloads
  - Timeout values
  - Retry policies (number of retries, backoff strategy)

    - Custom headers

- **FileDownloader**-represents a single file downloader instance; schedules and maintains active downloads.
- **DownloadRequest**-encapsulates a single download request (source, dest, headers, etc).
- **DownloadTask**-a handle to an asynchronous file downloading operation.
- **FileDownloaderConfig**-encapsulates file downloader configuration. Simplifies downloader instance creation and future refactorings. As an alternative, you may suggest the Builder pattern.
- **FileDownloadCallback**-encapsulate completion/failure callbacks.

High-Level Diagram

A high-level diagram visualizes the major components and their interactions. You can learn more about high-level diagrams here.

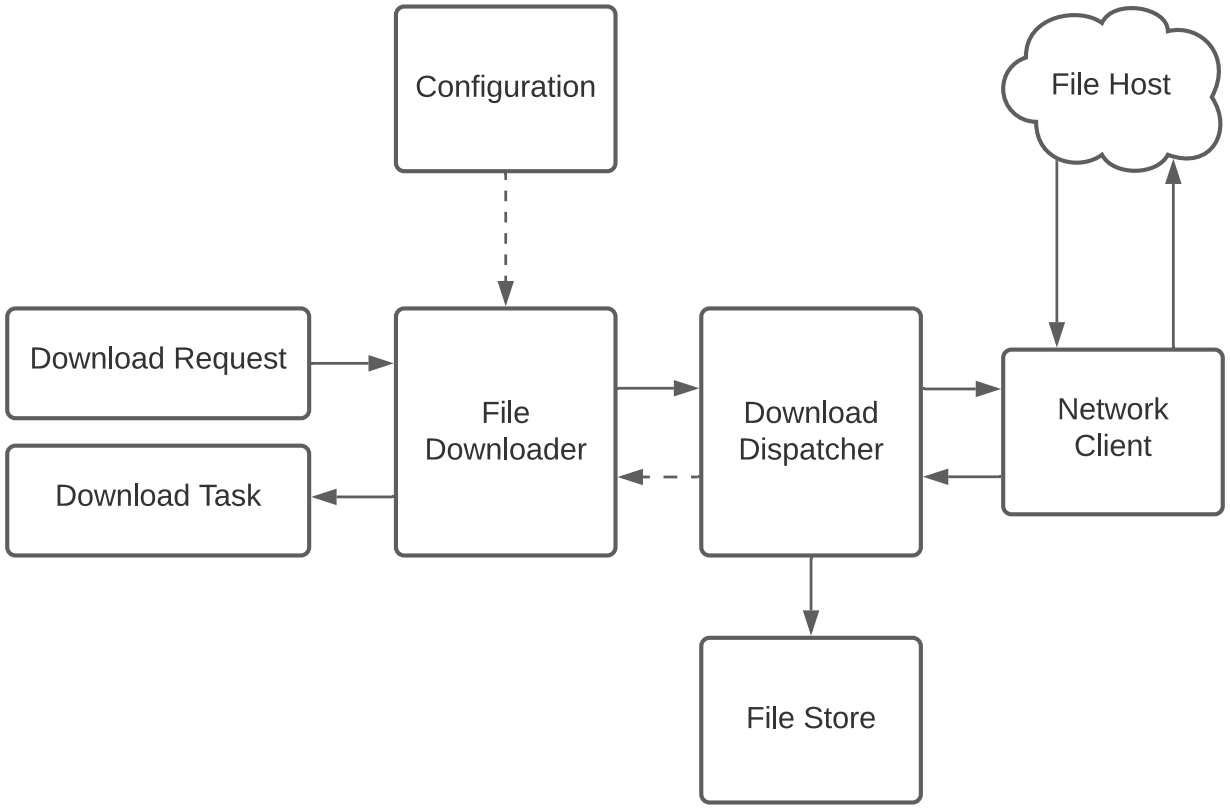


FIGURE 13.1 High-level Diagram

Components

- **File Downloader**-the central component which provides the client API and brings all components together.
- **Download Request**-encapsulates a single file downloading request; accepted by file downloader as input.

- **Download Task**-represents an asynchronous download operation; produced by the file downloader as output.
- **Download Dispatcher** - schedules and dispatches download operations.
- **Network Client**-handles receiving bytes over HTTP.
- **File Store**-writes file contents to the disk.

Deep Dive

After a high-level discussion, your interviewer might want to discuss some specific components of the system. Make sure to keep your explanation brief and don't overload it with details-let your interviewer guide the conversation and ask questions instead. You can learn more about deep-dive discussions here.

Deep Dive: Download Dispatcher

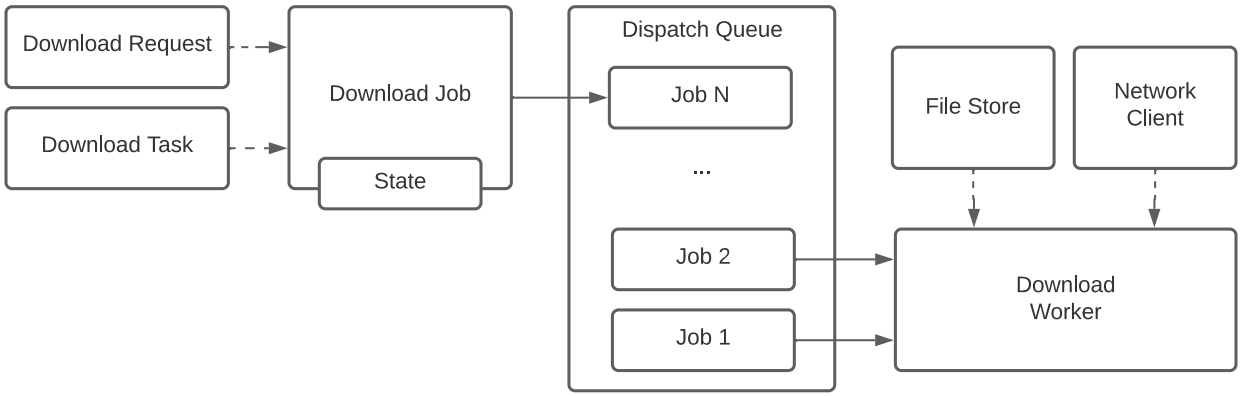


FIGURE 13.2 Download Dispatcher Diagram

> **Candidate**: "Download Dispatcher maintains a *concurrent* dispatch queue of jobs. Each job consists of a download request, a download task, and a state (PENDING, ACTIVE, PAUSED, COMPLETED, FAILED). The active jobs are dispatched by download workers."

> **Interviewer**: "Why do you need to maintain a job state?"
> **Candidate**: "To keep track of *pending*, *active*, and *completed* jobs: we limit the number of parallel downloads by design. Also, we need to maintain the `pause` state of the scheduled jobs."

> **Interviewer**: "What's the difference between a `Job` and a `Download Worker`?"
> **Candidate**: "A `Job` encapsulates a single downloading request from the user. A `Download Worker` is responsible for actual data transmission from the network."
> **Candidate**: "A `Job` is cheap and doesn't do anything. A `Download Worker` is expensive and handles blocking I/O."
> **Candidate**: "There might be an unlimited number of jobs and only a handful of workers (limited by

> the pool size)."
> **Candidate**: "There might be *multiple jobs* for the same URL but only *one worker* per download. For example, the same video file can be included in multiple playlists. The user can download these playlists in parallel - as a result, there might be different jobs for the same worker."
> **Candidate**: "If a single job gets canceled - its worker keeps downloading until done or all the associated jobs are canceled, too."

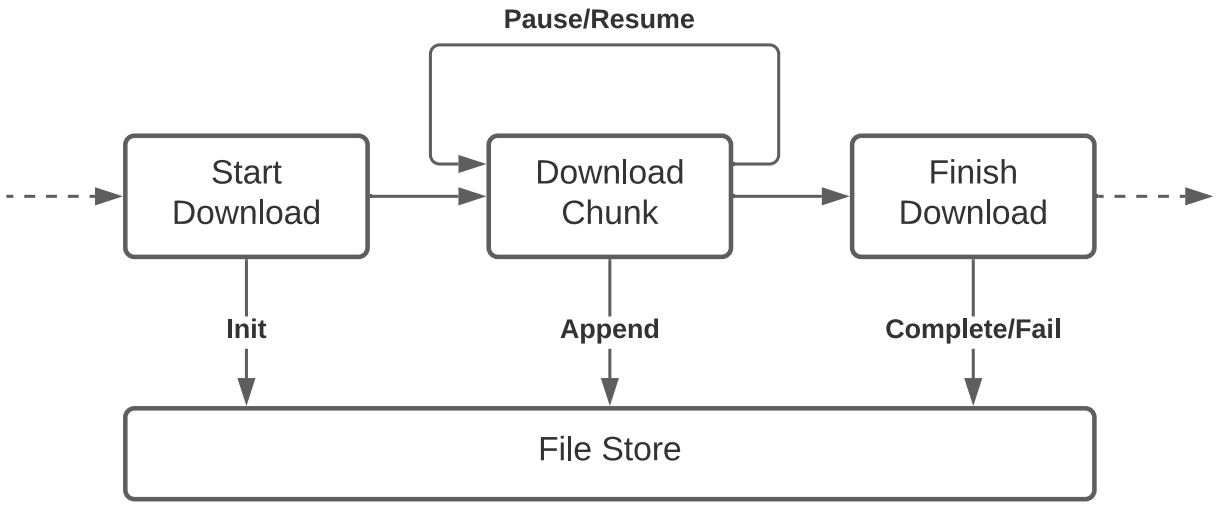


FIGURE 13.3 Download Job Diagram

> **Interviewer**: "Why do you need Init and Complete/Fail operations in the File Store?"
> **Candidate**: "This gives you an ability to pre-allocate disk space before downloading a file and perform post-processing/cleanup after a complete download."

Deep Dive: File Integrity & Security

> **Interviewer**: "How do we ensure the file downloaded correctly and wasn't tampered with?" **Candidate**: "We should implement a checksum validation. The server should provide a hash (MD5/SHA-256) in the response headers (e.g., `Content-MD5` or a custom header)." **Candidate**: "Once the download completes, but *before* we notify the client, the `File Store` component calculates the hash of the file on disk." **Candidate**: "If the hashes match, we rename the temporary file to the final destination. If not, we trigger a failure."

Platform Specific Considerations:

- **iOS:**
  - **Concurrency:** Use `DispatchQueue` with appropriate Quality of Service (QoS) levels for prioritizing downloads. Consider `OperationQueue` for more complex dependency management between tasks. Utilize `async/await` for structured concurrency.

    - **File Access:** Use `FileManager` for file system operations. Be mindful of error handling and potential exceptions.
- **Android:**
    - **Concurrency:** Use Kotlin coroutines with `Dispatchers.IO` for I/O-bound operations. Use `ExecutorService` or `ThreadPoolExecutor` for managing worker threads if more control over threading is required.
    - **File Access:** Use `java.io.File` and related classes for file system operations. Handle `IOException` appropriately.

Key Considerations:

- **Concurrency Control:** Implement proper synchronization mechanisms (locks, semaphores, concurrent collections) to prevent race conditions and data corruption when accessing shared resources (e.g., the job queue, file store).
- **Cancellation:** Implement a robust cancellation mechanism. This might involve interrupting network connections, deleting partially downloaded files, and updating job states. Utilize `CancellationToken` for more graceful and cooperative cancellation in both iOS and Android.
- **Retry Mechanism:** Consider adding a retry mechanism with exponential backoff for handling transient network errors.
- **Prioritization:** If supporting prioritized downloads, implement a priority queue in the Download Dispatcher. Ensure fair scheduling and prevent starvation of low-priority downloads.

Follow-up Questions

Prepare for questions that modify the original design.

> **Interviewer**: "How would you change your design if the library needs to keep track of downloaded files?"
> **Candidate**:
>
> - "We can add a component ("Progress Store") that maintains a structured record of scheduled jobs in a relational database."
> - "Each job would have an associated "progress" callback to signal every time the next chunk of bytes is received from the network."
> - "The Progress Store component will use the job callbacks to update the database."
> - "Once the job is complete-the corresponding record is deleted."

name	type
url	String
path	String

name	type
created_at	Date
total_bytes	Int
downloaded_bytes	Int
state	Int

> **Interviewer**: "Why won't you store files in the database?"
> **Candidate**: "It's easier to access a file on the disk than from the database. We can partially read BLOBs and provide the content as a stream but it might not be sufficient in the general case."

> **Interviewer**: "How would you handle downloading sensitive information?"
> **Candidate**: "We might introduce an "encrypted" File Store implementation and encrypt a fully-received file in post-processing. Consider using platform-specific encryption APIs (e.g., `CryptoKit` on iOS, Keystore for encryption key management on Android). Not sure if we can do it on the fly-don't have much experience working with encryption libraries."

> **Interviewer**: "How would you change your design to support download requests of different priorities? For example, user-critical, UI-critical, UI-non-critical, low-priority?"
> **Candidate**: "The first option: update Download Dispatcher to use a priority queue; the second option: configure on the File Downloader instance level and maintain different instances based on the priority (would also require synchronization between instances). The priority information could be included in a Download Request."

Foreground and Background Downloads (Optional)

Understanding the difference is important.

Foreground Download

Uses a worker thread in the host application process. Works best for short-running and urgent downloads.

pros:

- Immediate execution within the host app.
- Limited only by system resources.
- Easy to set up and troubleshoot.

cons:

- Stopped when the application is killed or suspended (works slightly different between iOS and Android).

Background Download

Might run in a separate process while the app is suspended. Best for long-running and nonurgent downloads.

pros:

- Guaranteed execution (iOS-survives backgrounding the app; Android-survives device restart).

cons:

- Hard to set up and debug.
- The host OS might wait for optimal conditions to perform the transfer, such as when the device is plugged in or connected to Wi-Fi.
- Must comply with Background Transfer Limitations.

Implementation Details

- A preferred download method is selected with the File Downloading request.
- The Download Dispatcher selects appropriate Download Worker implementation based on request configuration.

Platform Specific Considerations:

- **iOS:**
  - **Background Downloads:** Use `URLSessionConfiguration.background(withIdentifier:)` with `URLSessionDownloadTask`. Handle the `URLSessionDelegate` methods to receive progress updates and completion callbacks.
  - **Background Tasks Framework:** For tasks that don't fit the `URLSession` model, consider `BackgroundTasks.framework` for scheduling periodic tasks.
- **Android:**
  - **WorkManager:** Use `WorkManager` for scheduling reliable background tasks. It adapts to different Android versions and handles Doze mode and App Standby Buckets.

Major Concerns and Trade-Offs

Network/CPU/Battery Usage

- Having too many parallel downloads might negatively impact device battery life and increase cellular network usage. A possible workaround is to adjust the concurrency settings depending on the device

state. For example, limit the number of parallel downloads to `1` when using the cellular network or having a low battery charge, etc.

Disk Space Allocation

- Another major decision is whether to pre-allocate disk space or not. For example, the library can create place-holder files for every download in the queue to ensure enough space regardless of the file system state. It's hard to argue about this in a general case-as a workaround this can be configurable while initializing a File Downloader instance.

Additional Considerations:

- **Storage Limits:** Be mindful of device storage limitations. Implement mechanisms to handle low-storage situations gracefully (e.g., pausing downloads, deleting temporary files).
- **Data Usage:** Provide options for users to control data usage (e.g., only download over Wi-Fi). Respect the user's preferences and system settings.
- **Error Handling:** Implement robust error handling to handle network errors, file system errors, and other unexpected situations. Provide informative error messages to the user.

Conclusion

- Keep this in mind while preparing for a system design interview:
- Prioritize providing a "signal" over achieving perfection.
- Listen to your interviewer and keep track of the time.

Try to cover as much ground as possible without digging too much into the implementation details.

CHAPTER 14

Mobile System Design Exercise: Caching Library

The task might be simply defined as "Design Caching Library". The definition is purposely vague and requires some clarification.

Gathering Requirements

You are expected to ask clarifying questions and narrow down the scope of the task. This should not take more than 5 minutes.

> **Candidate**: "Are we designing a part of an application or a general-purpose library?"
> **Interviewer**: "A general-purpose library."

> **Candidate**: "What kind of items are we going to cache?"
> **Interviewer**: "Raw bytes."

> **Candidate**: "Do we have a size limit for each entry and an upper bound for the total number of items?"
> **Interviewer**: "Generally, no. But, for the sake of simplicity, let's assume image-like data up to 10 Mb per item. The max number of items is in order of thousands."

> **Candidate**: "Do we need to handle sensitive data? Should we support encrypted storage?"
> **Interviewer**: "Great question! What's your recommendation?"
> **Candidate**: "Having access to encrypted storage might be useful for specific use-cases. However, doing it in a general case might lead to an unnecessary overhead (encryption/decryption comes with CPU and memory cost)."
> **Candidate**: "We can allow turning encryption on/off on a library level or for certain specific items and let users decide based on their user-cases."
> **Interviewer**: "Sounds good. Let's leave it out of scope for now."

> **Candidate**: "How are we going to identify each cached item? Would string keys suffice?"
> **Interviewer**: "String keys are fine."

> **Candidate**: "Do we need to support an in-memory cache, a disk cache, or a hybrid memory-and-disk cache?"
> **Interviewer**: "Good question. How would the 'hybrid' approach work?"
> **Candidate**: "The cache would use the disk as a primary storage and keep a subset of items in memory for a quicker access."
> **Interviewer**: "Let's use the 'hybrid' approach."

> **Candidate**: "Should we limit the size of the cache?"
> **Interviewer**: "What's your recommendation?"
> **Candidate**: "I think, we should limit the number of bytes stored in memory and on the disk. Once the specific limit is reached - we should remove a portion of items according to the eviction policy."

> **Interviewer**: "What eviction policy are you going to use?"
> **Candidate**: "Since we're designing a general-purpose library-we should let the user select an eviction policy based on app-specific needs."
> **Candidate**: "We can start with pre-defined policies: Least Recently Used (LRU), Least Frequently Used (LFU), First In First Out (FIFO), Random, etc."
> **Candidate**: "We can also allow user-defined eviction policies using the Strategy design pattern. Although, we should probably leave it out of scope for now."
> **Interviewer**: "Sounds good. Let's leave user-defined policies out of scope."

> **Interviewer**: "Do we need to share the library between multiple platforms?"
> **Candidate**: "No. Let's leave it out of scope."

Make sure not to overload the system requirements with unnecessary features. Think in terms of MVP (Minimum Viable Product) and pick features that have the biggest value. You can learn more about requirements gathering here.

Functional requirements

- Users should be able to cache and retrieve raw byte data using strings as keys.
- Users should be able to configure disk and memory usage limits as a part of library initialization.
- Users should be able to configure the cache eviction policy as a part of library initialization.

Non-functional requirements

- The cached data should be persistent on the disk.
- A small subset of items should be kept in memory for quicker access.
- Once the cache is full - a portion of items should be deleted according to the eviction policy.

Out of scope

- User-defined eviction policies.
- Secure item storage.
- Cross-platform support.

Client Public API (Optional)

Your interviewer might want to discuss the client public API. In the simplest form it might look like this:

```
Cache:
+ init(config: CacheConfig)
+ set(key: String, value: [byte]): CacheTask
+ get(key: String): CacheTask
+ clear(key: String): CacheTask
+ clearAll()

CacheConfig:
+ init(maxMemoryCacheSize: Int, maxDiskCacheSize: Int)

CacheTask:
+ isSuccessful(): Bool
+ getCachedData(): [byte]?
+ getErrorMessage(): String?
+ addOnCompleteCallback(callback: (CacheTask) → Void)
```

> **Interviewer**: "Why do you return a `CacheTask` from `get` and `set` methods?"
> **Candidate**: "`get` and `set` calls may result in blocking I/O operations and should not be invoked on the main thread."
> **Candidate**: "We should make both of them async and let users specify completion callbacks."

NOTE: in this exercise, the API description is purposely left platform/language agnostic to reach a broader audience. During an actual interview, you would most likely design for either iOS or Android, and select the API style most suitable for the target platform. For example, you can advocate for coroutines instead of callbacks, suggest Rx-extensions, etc.

High-Level Diagram

A high-level diagram shows all major system components and their interactions (without implementation details). You can learn more about high-level diagrams here.

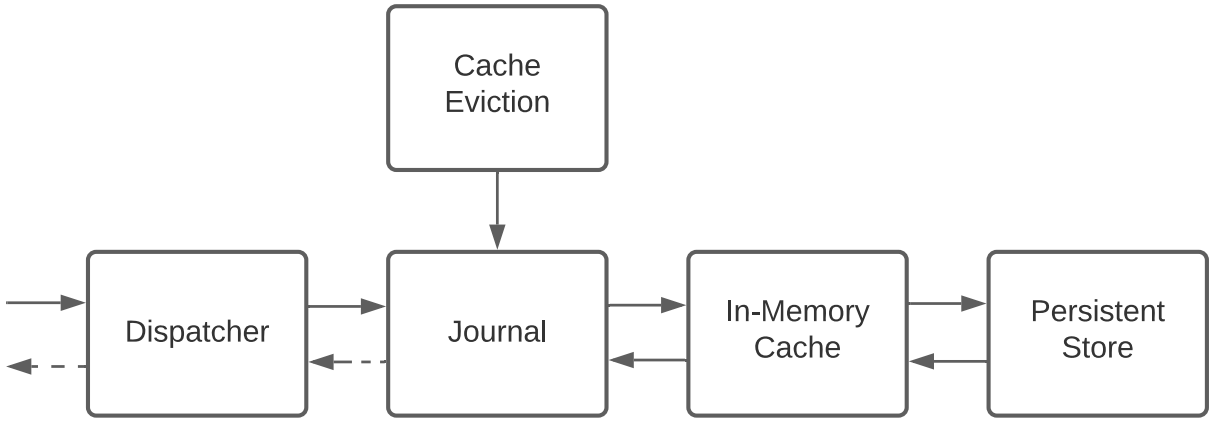


FIGURE 14.1 High-level Diagram

Components:

- **Dispatcher** - coordinates async read/write operations between the client and the library.
- **Journal** - maintains a structural record of metadata for cached items (access count, timestamps, size, etc).
- **In-Memory Cache** - handles fast in-memory item storage for quicker sequential access.
- **Persistent Store** - handles slow persistent disk item storage.
- **Cache Eviction** - manages cache overflow and item eviction.

Deep Dive

After a high-level discussion, your interviewer might want to discuss some specific components of the system. Make sure to keep your explanation brief and don't overload it with details-let your interviewer guide the conversation and ask questions instead. You can learn more about deep-dive discussions here.

Deep Dive: Dispatcher

> **Interviewer**: "I'm not sure what the Request Dispatcher component does…"
> **Candidate**: "It simplifies handling read/write concurrency."
> **Candidate**: "Using synchronous API calls might be a poor idea since item access might result in a blocking I/O operation. Doing so on the Main thread is not advisable and may result in jank, app termination, and *interview failures*."
> **Candidate**: "To solve this issue, we need to use async API. There might be a few options (depending on the platform and the programming language) but the callback-base approach is the simplest one and can be extended to coroutines, futures, promises, etc."

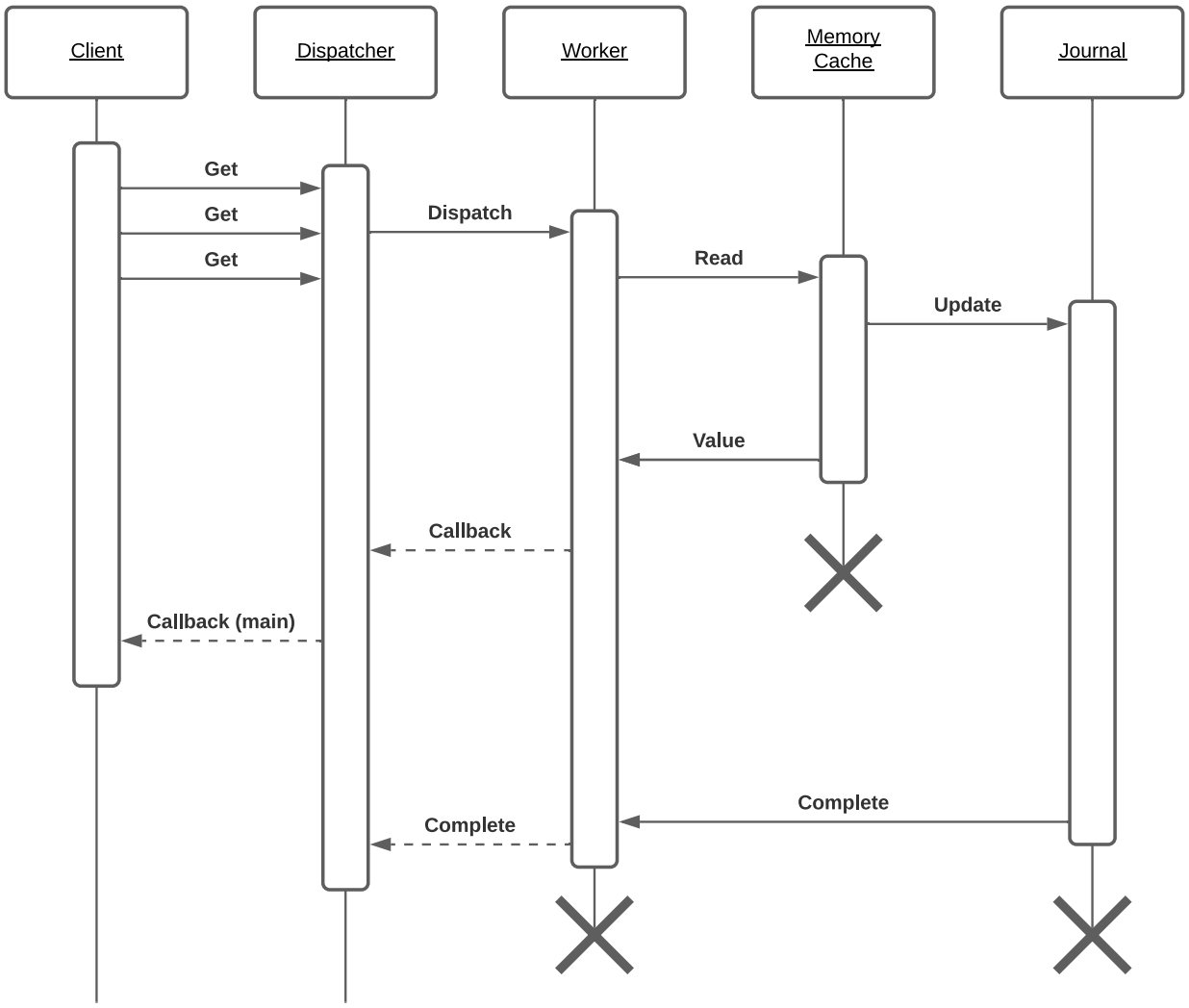


FIGURE 14.2 Dispatcher Sequence Diagram

> **Candidate**: "1) Dispatcher would maintain a *limited* pool of concurrent workers (via executor, dispatch queue, etc) to avoid creating too many background threads if the client makes a huge number of requests."
> **Candidate**: "2) Each worker would perform a cache access operation and notify its dispatcher via a callback when complete."
> **Candidate**: "3) After a worker completes - the dispatcher would notify the client by invoking the completion callback on the main thread (or user-specified executor/queue)."

Deep Dive: Journal

> **Interviewer**: "Would you explain how the Journal component works?"
> **Candidate**: "The Journal component creates and updates metadata records for cached items."

name	type
key	String
size_bytes	Int
access_count	Int
last_accessed	Date

> **Candidate**: "Every time a cached item is accessed - the Journal will increase the `access_count` and update the `last_accessed` timestamp. This metadata would allow selecting items for eviction and calculating the total cache size."
> **Candidate**: "The Journal itself could be stored in a text/binary file, property list, or a relational database (ORM). I think a relational database might be the best approach since it allows partial updates, querying, and provides data integrity."

> **Interviewer**: "So the Journal would only store the metadata - where would the actual bytes be stored?"

First Option: File System

> **Candidate**: "We can write binary files in the app 'cache' directory and use generated UUID's as file names. The item key won't work as a filename since the user may pass illegal path characters."
> **Candidate**: "One drawback of such approach - we would need to sync the Journal and the file system separately which might lead to race condition bugs. For example, the host app might crash after the Journal is updated but before the data is written to the disk."
> **Candidate**: "We may overcome this by introducing items states: `CLEAN`, `DIRTY`, `REMOVE`, etc. When the item is being created or updated - it will be marked with `DIRTY` state; once the process completes - it will be marked with `CLEAN` state; same for `REMOVE` state. The library may check the Journal upon initialization and prune dirty items and their files."

name	type
key	String
path	String
size_bytes	Int
access_count	Int
last_accessed	Date
state	Int

> **Candidate**: "The introduction of states makes it more reliable but won't solve all potential concurrency issues. We might still run in situations where different threads are trying to write the same item concurrently. We may try to add more synchronization but it would greatly complicate the design."

> **Candidate**: "The advantage of this option - the app cache (with all binary files) might be cleaned up by the user from the app settings when device storage is low. However, on mobile, the OS can also *automatically* delete files in the temporary/cache directory when disk space is critically low. This leads to a 'Zombie Journal' problem where the metadata exists, but the file is gone. Our `get()` method must handle 'File Not Found' errors gracefully by removing the orphan journal entry."

Second Option: BLOBs

> **Candidate**: "We can store the binary data in the same database where the Journal data is stored."

name	type
key	String
data	BLOB
size_bytes	Int
access_count	Int
last_accessed	Date

> **Candidate**: "This way we don't need to worry about synchronization and item states. We can also merge the Journal and the Persistent Store components."

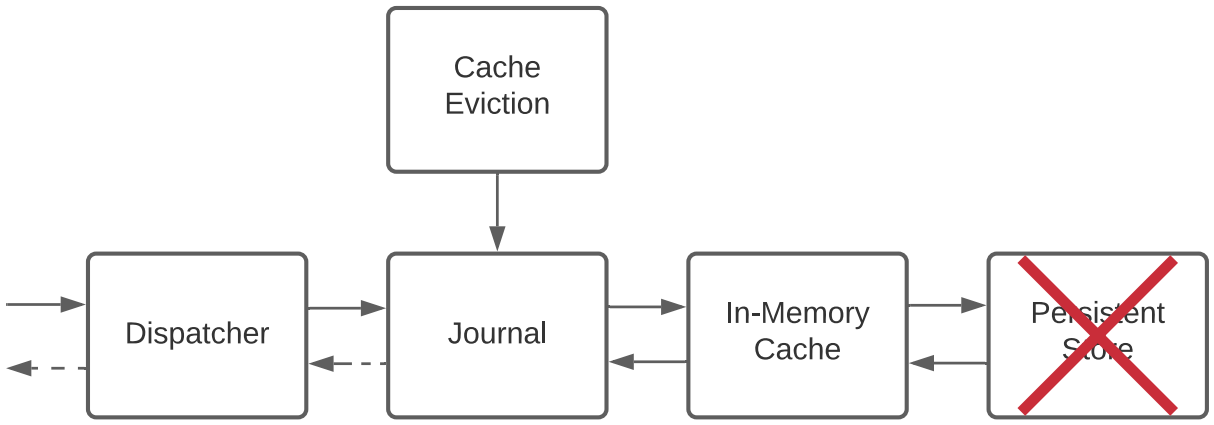


FIGURE 14.3 High-level Diagram

> *NOTE: It's ok to change some of your initial statements as the interview progresses - it's more about the thinking process and not pretty diagrams nor "making it right" on the first attempt.*

Candidate: "The disadvantage of this option - the binary data can't be cleaned from the app settings without removing the database itself and losing all its metadata."
Candidate: "There are lots of discussions around BLOBs vs file system. I don't have enough experience to take any side but I think the second option is an easier one."

Deep Dive: Cache Eviction

Interviewer: "How does the Cache Eviction component work?"
Candidate: "When a new cache item is created - the component checks the disk and in-memory storage sizes; if any of them exceed the maximum allowed size - the Cache Eviction runs a prepared query to purge a sub-set of items."

Interviewer: "Why do you need a separate component and won't include the logic directly into the Journal and the In-Memory Cache?"
Candidate: "I'm trying to follow a Single-responsibility principle: the Journal and the In-Memory Cache components should only perform a single function and should not be aware of cache eviction existence. It's also easier to test different components in isolation."

Interviewer: "You've mentioned *prepared queries* - can you provide a bit more details?"
Candidate: "For the Journal component, it could be a prepared SQL statement, a fetch/batch request, or any other ORM mechanism to delete multiple items. Alternatively, we can iterate using a cursor (if the vendor supports it) and delete items until the new size can accommodate an extra item."
Candidate: "The drawback of this approach is that the Cache Eviction components might *know* too much about other components implementation. At the same time - this is an efficient and a simple-to-build approach. Everything is a trade-off."

Interviewer: "What method would you choose?"
Candidate: "I think, a SQL cursor might be the most efficient since we don't need to load unnecessary items."

Interviewer: "Do you know how cursors work under the hood?"
Candidate: "Not really. I only worked with them and don't know the implementation details."

> **Interviewer**: "How about the In-Memory eviction?"
> **Candidate**: "For the In-Memory component, it could be a custom iterator over the collection which would perform a similar function."

> **Interviewer**: "What data structure would you use for the in-memory store?"
> **Candidate**: "I would use a **Doubly Linked List** combined with a **Hash Map**. The Hash Map provides `O(1)` access to items. The Doubly Linked List maintains the eviction order (e.g., LRU). When an item is accessed, we move it to the head of the list in `O(1)`. When we need to evict, we remove from the tail in `O(1)`."

Handling System Memory Warnings

> **Interviewer**: "What happens if the system runs low on memory?" **Candidate**: "If our cache consumes too much memory, the OS might kill the host application. We must listen for system memory warnings. When a warning is received, the library should aggressively clear the **In-Memory Cache** (e.g., reduce size by 50% or clear entirely) to free up resources and keep the app alive."

Follow-up Questions

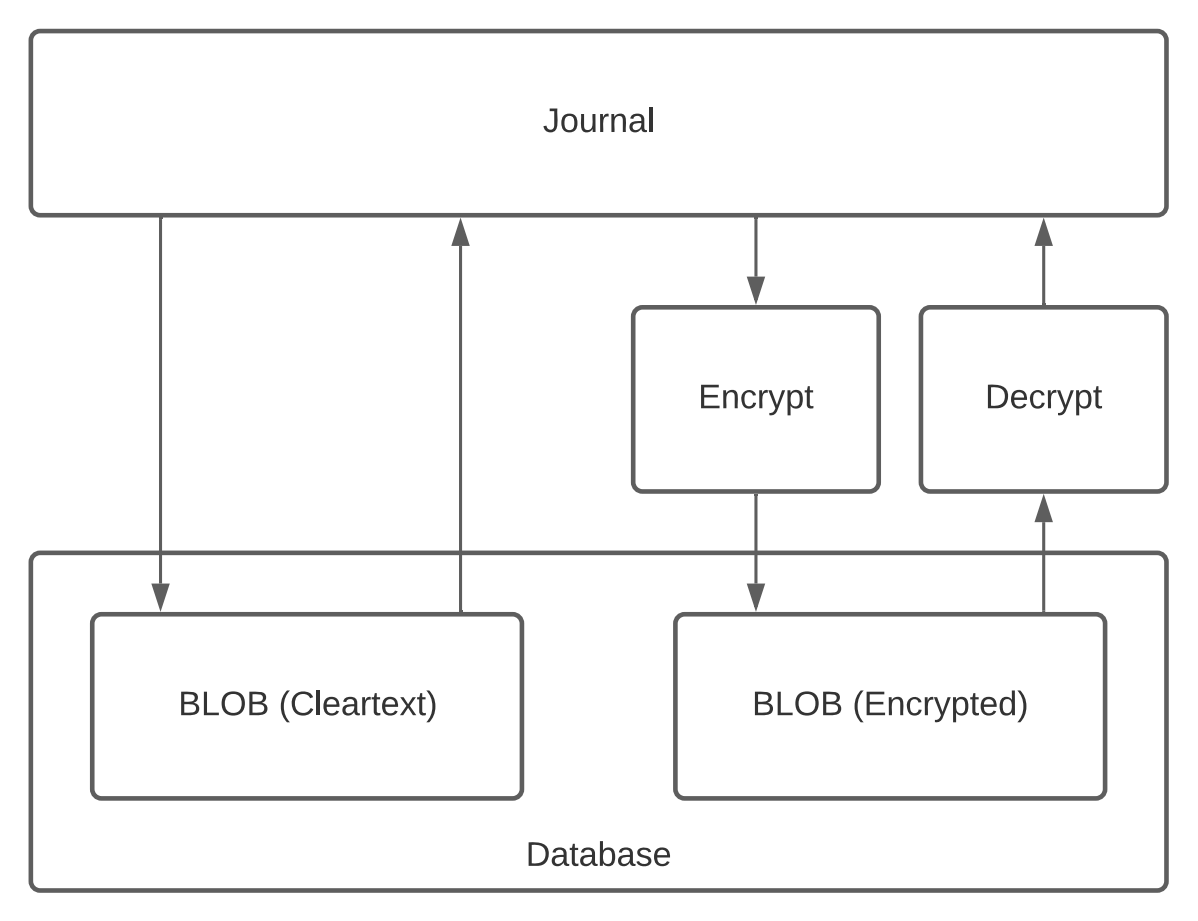
Some interviewers might ask follow-up questions that might change the original design and introduce new requirements.

Sensitive Information

> **Interviewer**: "How would you change your design to support handling sensitive information?"
> **Candidate**: "Is every item considered to be sensitive?"
> **Interviewer**: "Not necessary."
> **Candidate**: "In this case, we might want to provide users some flexibility. They can set the whole cache to be encrypted or only secure a subset of the items."

> **Candidate**: "To my best knowledge, iOS/Android platforms do not provide a built-in secure database at the moment. As a workaround, we could use third-party providers (like SQLCipher) to *encrypt the whole database* or only *encrypt data BLOB storage* in the database (under a strong assumption that cache keys do not contain any sensitive information)".
> **Candidate**: "For our general-case purpose, I would rather select the BLOB encryption since the vast majority of users won't store any sensitive information. Also, I'd be very cautious before adding a 3rd-

party dependency into the library: it might introduce licensing issues and binary/version incompatibility with the host app."



Candidate: "The encryption key would be generated and stored in the Keystore/Keychain."
Candidate: "We should also store an `encrypted` flag in the database to mark secure items."

name	type
key	String
data	BLOB
size_bytes	Int
access_count	Int
last_accessed	Date
encrypted	Bool

Candidate: "As a side note, many applications already have a mature encryption stack. For this case, we might provide them the ability to specify a custom encryption/decryption implementation."

```
CacheEncryption:
+ encrypt(data: [Byte]): [Byte]
+ decrypt(data: [Byte]): [Byte]

CacheConfig:
+ init(maxMemoryCacheSize: Int, maxDiskCacheSize: Int)
+ setCacheEncryption(encryption: CacheEncryption)
```

> **Candidate**: "This way they can better control user data privacy: for example, download encryption key from the backend upon login, etc."

Cross-platform Support

> **Interviewer**: "Imagine, you need to write a cross-platform library that should run on mobile and desktop platforms. How would you change your design?"
> **Candidate**: "Would we need to share the cached storage somehow?"
> **Interviewer**: "What do you mean?"
> **Candidate**: "Should the permanent cache storage be transferred between platforms?"
> **Interviewer**: "No."
> **Candidate**: "In this case, we don't need storage binary compatibility".
> **Candidate**: "I would separate the library into two modules: 1) common code (written in C/C++); 2) platform implementation/adapters (Java/Kotlin/Swift/Objective-C)."
> **Candidate**: "The Journal, the Cache Eviction, and the Database interface would be included in the common module; the rest would be included in the implementation module."
> **Candidate**: "However, I would be very cautious around using native code: 1) hard to debug; 2) hard to read crash logs; 3) need to compile for all supported architectures (arm7, arm64, x86, etc); 4) more likely to crash the host app (instead of just throwing an exception). 5) harder to develop compared to modern languages. 6) iOS/Android developers are more likely to submit pull-requests if the codebase built specifically for the target platform (assuming open-source)."

Major Concerns and Trade-Offs

Library responsiveness vs device resource usage

The biggest decision to make is Dispatcher's worker pool size. Having a larger amount of workers might make the library more responsive but would eventually spawn more threads.

Data security

- No data is considered secure as long as its encryption key is stored on the same device. Some devices provide hardware-backed key storage but we can't guarantee this in a general case.
- Encryption/decryption also comes with a CPU and memory cost: this is especially important for performance-critical cases and low-level devices.

Conclusion

- Keep this in mind while preparing for a system design interview:
- Prioritize providing a "signal" over achieving perfection.
- Listen to your interviewer and keep track of the time.
- Try to cover as much ground as possible without digging too much into the implementation details.

Looking for more content?

I'm thinking about creating an in-depth mobile system design course on top of the free articles. Please, fill out this form to express your interest!

CHAPTER 15

Mobile System Design Exercise: Image Library

The task might be simply defined as "Design Image Library". The definition is purposely vague and requires some clarification.

Gathering Requirements

You are expected to ask clarifying questions and narrow down the scope of the task. This should not take more than 5 minutes.

> **Candidate**: "Are we designing a part of an application or a general-purpose library?"
> **Interviewer**: "A general-purpose library."

> **Candidate**: "Are we designing a library that loads images or handles a collection of photos?"
> **Interviewer**: "The one that loads images."

> **Candidate**: "Where do we need to load images from?"
> **Interviewer**: "I don't know - you tell me."

> *NOTE: some interviewers may want you to "drive" the conversation and would "subtract points" if you try to ask them what you need to do.*

> **Candidate**: "Network, filesystem, and app resources would make sense. We can expose an API to create custom loaders but we could leave it out of scope for now."
> **Interviewer**: "Yes, let's leave it out of scope."

> **Candidate**: "Do we need to support UI-targets?"
> **Interviewer**: "Not sure what you mean."
> **Candidate**: "Do we want the library to load images directly to UI components (like ImageView, Button, etc)?"
> **Interviewer**: "Yes."

> **Candidate**: "I think, it might be beneficial to support non-UI targets - like saving the image to a file or providing a callback to access the image data when loading is complete."
> **Interviewer**: "Sounds good."

> **Candidate**: "I would also add a caching support to store downloaded images on the disk to preserve the bandwidth and make it easier to support offline state."
> **Interviewer**: "Sounds good."

> **Candidate**: "We should also provide view lifecycle management for UI-targets."
> **Interviewer**: "Not sure what you mean."
> **Candidate**: "That means that the library would track view hierarchy lifecycle events to decide when to stop loading and free resources. For example, you should interrupt image loading when the target view becomes detached from the view hierarchy."
> **Interviewer**: "Yes, it's a good feature."

> **Candidate**: "It would be good to provide thumbnails, customizable placeholders, loading indicators, and transition animation for UI-targets - but we should probably leave it out of scope."

> **Interviewer**: "What do you mean by `thumbnails`?"
> **Candidate**: "Many backends provide low-resolution versions of the images along with hi-resolution ones. It's useful for metered connections and slow networks. The image loader will prioritize the thumbnail loading over the high-resolution versions - this way, the host app UI would be able to do the initial rendering much faster."

> **Interviewer**: "Let's leave it out of scope."

Make sure not to overload the system requirements with unnecessary features. Think in terms of MVP (Minimum Viable Product) and pick features that have the biggest value. You can learn more about requirements gathering here.

Functional requirements

- Users should be able to load images from the network, filesystem, and app resources to UI/non-UI targets.

Non-functional requirements

- Cache loaded images in memory and on the disk.
- Image loading respects view hierarchy lifecycle events.

Out of scope

- Custom image loaders.
- Image placeholders, loading indicators, transition animations.

High-Level Diagram

A high-level diagram shows all major system components and their interactions (without implementation details). You can learn more about high-level diagrams here.

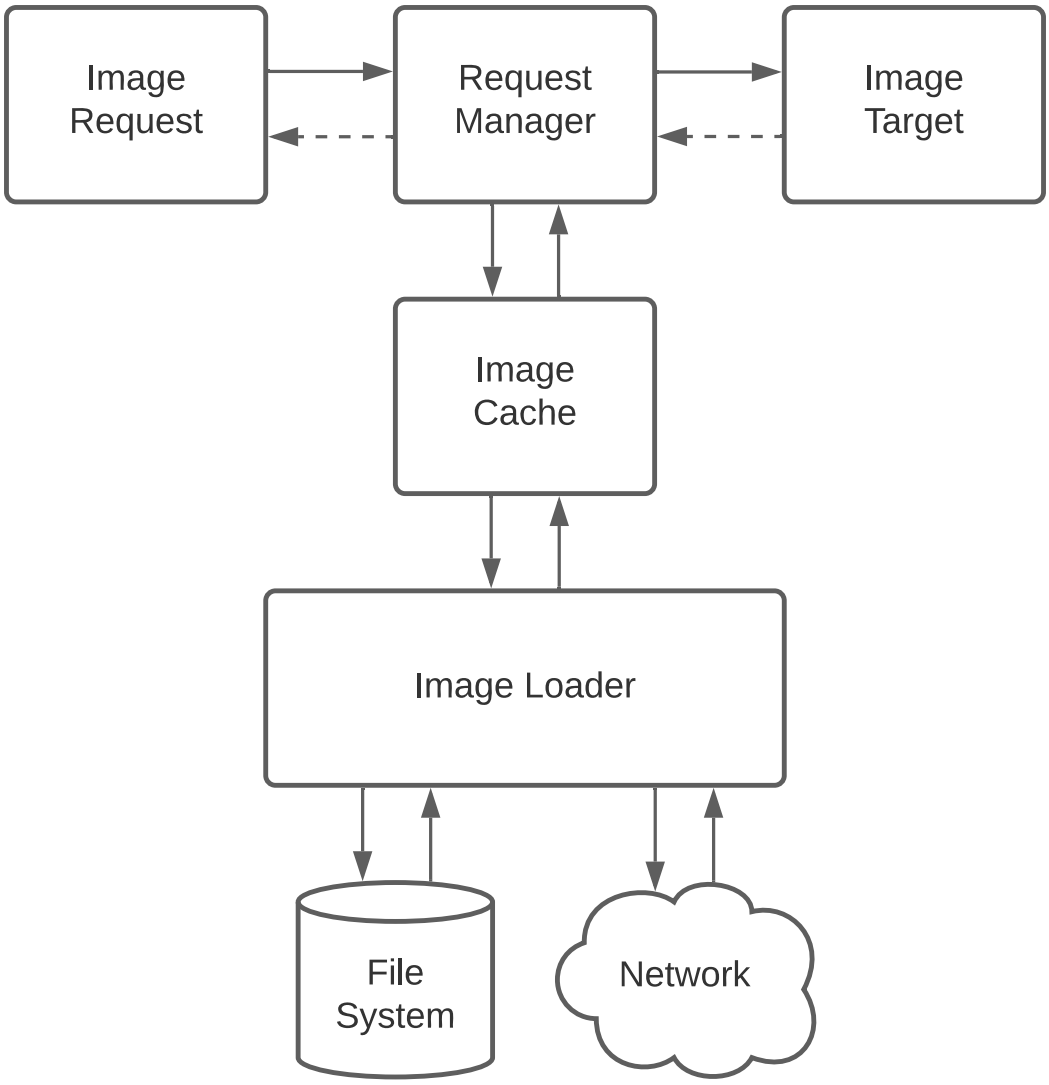


FIGURE 15.1 High-level Diagram

Components:

- **Image Request** - encapsulates a single image request; accepted by request manager as input.
- **Request Manager** - accepts and dispatches image requests to corresponding targets.
- **Image Cache** - fast in-memory image storage for quicker access.

- **Image Loader** - coordinates image loading from different source (filesystem, app resources, network).
- **Image Target** - incapsulate image target destination (UI-element, in-memory image data, etc).

Deep Dive

After a high-level discussion, your interviewer might want to discuss some specific components of the system. Make sure to keep your explanation brief and don't overload it with details-let your interviewer guide the conversation and ask questions instead. You can learn more about deep-dive discussions here.

Deep Dive: Image Cache

> **Interviewer**: "What is the purpose of the `Image Cache` component?"
> **Candidate**: "It's an in-memory LRU cache to keep a subset of loaded images for quicker access."

> **Interviewer**: "What would you use for the cache key?"
> **Candidate**: "It's a good question. The first option is to use the image URI. However, we might also want to distinguish between images with the same URI but different dimensions and formats. To accomplish this - we can create a new `ImageKey` type to encapsulate image parameters."

```
ImageKey:
+ init(source:Uri, width: Int, height: Int, format: ImageFormat)
```

> **Interviewer**: "Why would you need to include image dimensions?"
> **Candidate**: "Mostly to save memory and make drawing more efficient: it's better to re-scale the image to fit its target while loading."

> **Interviewer**: "What do you mean by `format`?"
> **Candidate**: "The image format describes the bit encoding for every pixel (for example, `RGBA8888`, `RGB888`, `RGB565`, etc). We can optimize our memory usage by reducing the bit depth or discarding the alpha channel."

> **Interviewer**: "Should this component also handle disk cache?"
> **Candidate**: "The disk cache only makes sense for the images downloaded over the network. I think the network client can handle it better."

Interviewer: "Why do you think so?"
Candidate: "Most of the modern HTTP clients have built-in disk caching mechanisms that respect `Cache-Control` directives for responses."
Candidate: "This way, we only need to specify the cache size and the client can handle content expiration for us."

Interviewer: "How would you select the size of the cache?"
Candidate: "I would start with some default value. For example, the Google Drive app has `250` Mb disk cache size by default."
Candidate: "Then I would provide a library setting to override this."

Interviewer: "That makes sense but I was asking about in-memory cache size 🙂"
Candidate: "This part is harder since we can't make any strong assumptions on the memory usage patterns of the host app."
Candidate: "A simple heuristics could be allocating a chunk proportional to the max device memory. For example, Google suggests `1/8` th."
Candidate: "We can also use low-memory callbacks to purge the cache when the device runs out of memory."

Interviewer: "Can you see any drawbacks of using a 3rd party library?"
Candidate: "Depending on a 3rd party library is tricky since it can lead to semantic and binary incompatibilities with the host app. On the other hand, it handles lots of complexity for us and encapsulates the experience of many developers who worked on it. I think it's a trade-off between time-to-the-market and code ownership."

NOTE: If you don't have much experience with network stack internals - it's ok to mention a well-known library as a workaround.

Deep Dive: Image Loader

Interviewer: "Can we talk more about the `Image Loader` component?"
Candidate: "The component is responsible for loading images from different sources such as file system, app resources, and network."
Candidate: "It accepts a `Loader Request` and produces an image in return."

Interviewer: "What would you include in the request?"
Candidate: "An `ImageKey` and a callback function."

```
LoaderRequest:
+ init(key: ImageKey, callback: (key: ImageKey, image: Image?, error: String?) → Void)
```

NOTE: in this exercise, the API description is purposely left platform/language agnostic to reach a broader audience. During an actual interview, you would most likely design for either iOS or Android and select the API style most suitable for the target platform. For example, you can advocate for coroutines instead of callbacks, suggest Rx-extensions, etc.

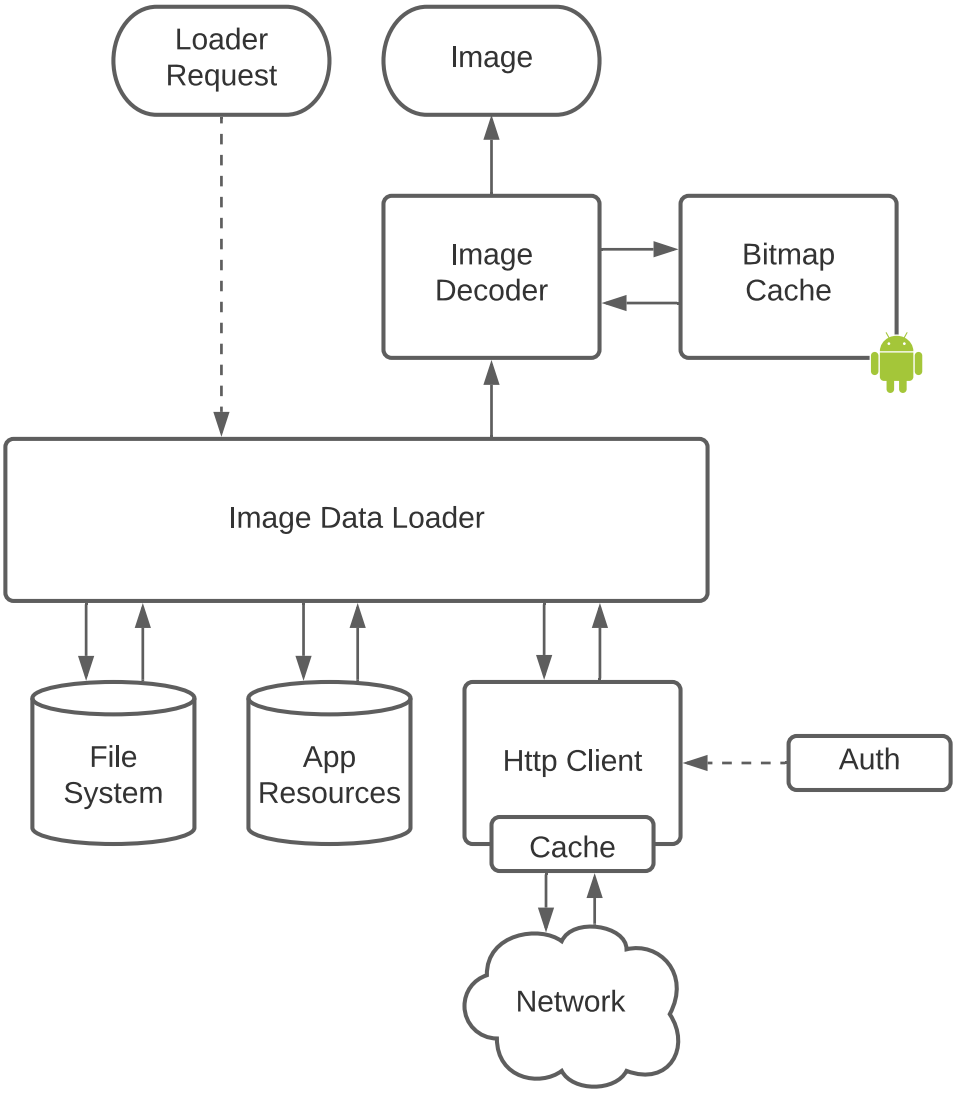


FIGURE 15.2 High-level Diagram

Candidate: "I would also decouple image data loading from the decoding. This way, we can easily add new decoders using the Strategy design pattern without changing the loader itself."

NOTE: Android engineers might also want to mention a special Bitmap cache to avoid excessive garbage collection with recycled bitmaps. For more information check BitmapPool from Glide.

Interviewer: "I'm a little confused by your caching approach - why there's no cache functionality in the Image Loader component?"
Candidate: "Mostly due to the Single-responsibility principle: the Image Loader should not know anything about caching - its single purpose is to ~~pass butter~~ load images."
Candidate: "However, the HttpClient component handles the disk cache for downloaded images. This is a trade-off between a 'clean' design and an 'ease of implementation' - might be a good short-term strategy but would scale poorly in a long run."

Interviewer: "What do you mean by `scaling` ? Are we talking about any backend issues?"
Candidate: "No. By `scaling` , I mean library compatibility and long-run support. Depending on the platform, a 3rd-party dependency may have version conflicts with other libraries from the host app."
Candidate: "You also need to have a good plan on supporting and updating the dependency over time: for example, you need to decide on the update schedule (each release, each major release, critical bugfix-only, etc) or if you want to maintain a private fork."
Candidate: "There might be licensing issues for the customers, too."

NOTE: See "Chapter 21. Dependency Management" from the "Software Engineering at Google" book.

Interviewer: "Sounds tough - why would you want to have the dependency in the first place?"
Candidate: "This a good example of `implement` vs `re-use` trade-off. Personally, I think we should prioritize time-to-market first and then refine in future releases."
Interviewer: "Makes sense."

NOTE: The system design interview is not all about technology and engineering - it's always good to keep business needs in mind, too.

Candidate: "I forgot to mention this but we can also easily add authentication support on the network client level."
Interviewer: "That's fine - we can leave it out of scope."

Deep Dive: Image Request

Interviewer: "Tell me more about `ImageRequest` ."
Candidate: "It encapsulates the image loading parameters and the image target. We can use fluent chaining to create a cleaner interface."

```
ImageRequest:
+ load(source: Uri): ImageRequest
+ load(format: ImageFormat): ImageRequest
+ into(target: ImageView): ImageRequest // we can overload this to support multiple
targets
```

> **Candidate**: "We can register lifecycle callbacks with UI-targets (ImageView, Button, etc) for easier control of resources. For example, we can stop loading when the target is detached from the view hierarchy, becomes invisible, or gets recycled as a part of list scrolling."

> *NOTE: We're not going to discuss the actual implementation here since it greatly depends on the UI framework. Make sure to discuss what mechanisms you would use and how to prevent memory leaks.*

Follow-up Questions

Some interviewers might ask follow-up questions that might change the original design and introduce new requirements.

> **Interviewer**: "How would you change your design if you can't use the HttpClient caching?"
> **Candidate**: "I would store image metadata in a relational database and store image bytes in the internal cache directory?"
> **Candidate**: "Also, I would set a disk usage limit for the cache and provide a simple LRU eviction policy."

> **Interviewer**: "Why would you select the internal cache directory? What other options do you have?"
> **Candidate**: "The cache directory can be automatically cleaned up when the device disk space runs low. Also, the cached contents won't be backed up with the app."
> **Candidate**: "I would prefer an internal storage for privacy reasons and to avoid using extra permissions (Android only)."
> **Candidate**: "I don't think if I know a better storage option."

> *NOTE: It's ok to tell the interviewer that you don't have much experience with the storage options. Better be honest than get caught in a lie.*

> **Interviewer**: "What if you need to store sensitive images?"
> **Candidate**: "I would support this as a flag in the `ImageRequest` and encrypt/decrypt corresponding files using encryption keys from Android Keystore or iOS Keychain."
> **Candidate**: "A **better** option is not to store sensitive images at all."

Major Concerns and Trade-Offs

Balancing Memory/Disk/CPU/Bandwidth usage

The biggest decision to make is how to properly handle system resources utilization:

- Using more memory would make the library more responsive but increases the risk of the host app being killed in the background.
- Using more disk space saves application bandwidth but frequent reads/writes can increase the device temperature and increase the chances for the host app to be uninstalled in low disk space conditions.
- Using more bandwidth may cost the user money and increase the battery drain.

Since you can't make any assumptions on the specific use-cases - making these parameters configurable might be the preferred approach. If you are not sure what default values to use - learn from the existing applications.

Low Data Mode (iOS) and Data Saver mode (Android)

We might want to respect the device settings for saving cellular data. Possible options might include:

- Block image downloading altogether.
- Introduce the level of priority for each image and only download UI-critical components. See Quality Of Service for more information.
- Provide support for hi-quality and low-quality downloads and only fetch low-quality images.
- Prefer cached data instead of fetching it from the network.

Conclusion

Keep this in mind while preparing for a system design interview:

- Prioritize providing a "signal" over achieving perfection.
- Listen to your interviewer and keep track of the time.
- Try to cover as much ground as possible without digging too much into the implementation details.

Looking for more content?

I'm thinking about creating an in-depth mobile system design course on top of the free articles. Please, fill out this form to express your interest!

CHAPTER 16

Mobile System Design Exercise: Chat Application

The task might be simply defined as "Design Chat Application". The definition is purposely vague and requires some clarification.

Gathering Requirements

You are expected to ask clarifying questions and narrow down the scope of the task. This should not take more than 5 minutes.

> **Candidate**: "Are we designing a general usage app (like WhatsApp) or something more specific (like an internal chat app for a company)?"
> **Interviewer**: "Something like WhatsApp."

> **Candidate**: "Do we have any information on the number of monthly active users?"
> **Interviewer**: "Why is this important for a mobile app?"
> **Candidate**: "We need to make sure that our clients won't accidentally create a DDoS attack on our backend services."

> **Interviewer**: "In what cases a DDoS *attack* is possible?"
> **Candidate**: "The most likely reason is a frequent HTTP-polling that produces large volumes of network traffic."
> **Candidate**: "Another reason - not using an exponential backoff while retrying failing requests. Million of clients retrying their requests at the same time can produce an unnecessary load to our backend services."
> **Interviewer**: "Got it. Let's say we're expecting to have 1,000,000 monthly active users."

> **Candidate**: "Do we know anything about our target market and the audience?"
> **Interviewer**: "Why do you think this is important?"
> **Candidate**: "An app targetted primarily for North America and Europe might have a different set of performance requirements compared to a similar app targetted to developing countries. The users in developed countries usually have more powerful phones compared to their peers in developing countries. The cost of cellular traffic in Europe and North America is much lower than in India."
> **Interviewer**: "Let's design an app for North America."

> **Candidate**: "Do we want to be able to edit/delete messages?"
> **Interviewer**: "Let's leave it out of scope."

> **Candidate**: "Do we want to support group chats?"
> **Interviewer**: "No, let's leave it out of scope."

> **Candidate**: "Do we need to support sign-up and log-in?"
> **Interviewer**: "We can leave it out of scope."

> **Candidate**: "Are there any constraints on media attachments, like file size or types?"
> **Interviewer**: "Let's support images only, max 10MB. No video/audio."

Make sure not to overload the system requirements with unnecessary features. Think in terms of MVP (Minimum Viable Product) and pick features that have the biggest value. You can learn more about requirements gathering here.

Functional requirements

- User should be able to see the list of recent chats sorted by date.
- User should be able to open a 1-on-1 chat, send and receive messages.
- User should be able to add photo attachments to chat messages.
- User should be able to see chat message status (sending, sent/failed, received) and read receipts.

Non-functional requirements

- Offline support (reading chat messages).
- Secure message storage.
- Real-time notifications.

Out of scope

- Video and voice attachments.
- Sign-up and log-in.
- Editing/Deleting messages.
- Group chats.

> *NOTE: It is tempting to include additional requirements such as 'last seen`, message replies/forwarding, reactions, etc. Proper time management is crucial during the interview - try keeping everything closer to an MVP and move on to the next section.*

High-Level Diagram

A high-level diagram shows all major system components and their interactions (without implementation details). You can learn more about high-level diagrams here.

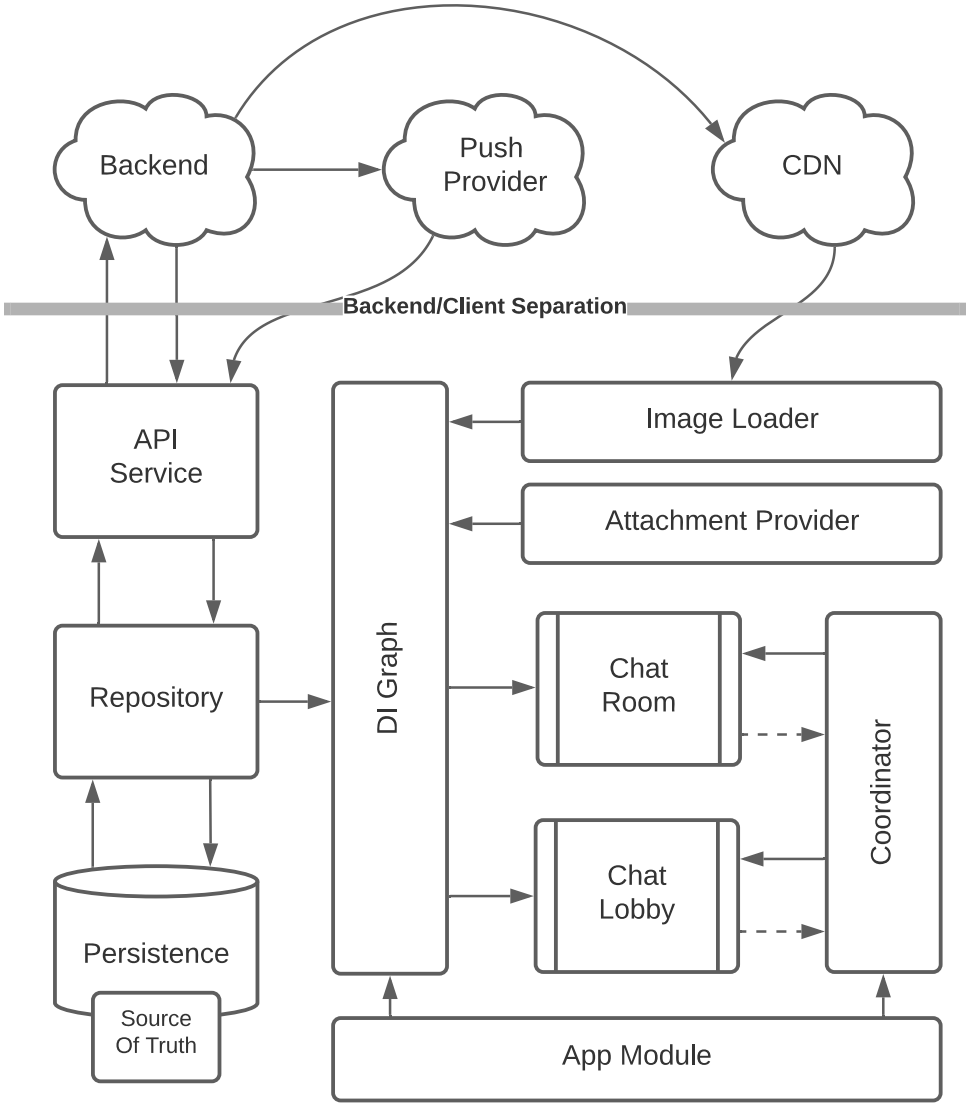


FIGURE 16.1 High-level Diagram

> *NOTE: This diagram looks familiar to other cases from the guide. The main reason is the adopted architectural pattern combination MVx + repository + coordinator.*

Server-side components:

- **Backend**
  Represents the whole server-sider infrastructure. Most likely, your interviewer won't be interested in discussing it.

- **Push Provider**
  Represents the Mobile Push Provider infrastructure. Receives push payloads from the Backend and delivers them to clients.
- **CDN (Content Delivery Network)**
  Responsible for delivering static content to clients.

Client-side components:

- **API Service**
  Abstracts client-server communications from the rest of the system.
- **Persistence**
  A single source of truth. The data your system receives gets persisted on the disk and then propagated to other components.
- **Repository**
  A mediator component between API Service and Persistence.
- **Chat Lobby**
  Represents a set of components responsible for displaying a list of recent chats.
- **Chat Room**
  Represents a set of components responsible for displaying a single chat room.
- **DI Graph**
  Dependency injection graph. See: Dependency injection in Android and Dependency injection in iOS.
- **Image Loader**
  Responsible for loading images from web/disk into UI elements.
- **Attachment Provider**
  Responsible for loading image attachments from the media library and camera into memory.
- **Coordinator**
  Organizes flow logic between Chat Lobby and Chat Room components. Helps decouple components of the system from each other.
- **App Module**
  An executable part of the system which "glues" components together.

Deep Dive

After a high-level discussion, your interviewer might want to discuss some specific components of the system. Make sure to keep your explanation brief and don't overload it with details-let your interviewer guide the conversation and ask questions instead. You can learn more about deep-dive discussions here.

Deep Dive: API Service

> **Interviewer**: "Can you tell me more about the API Service component?"
> **Candidate**: "It's an abstraction for the network communication layer."
> **Candidate**: "The idea is to isolate the low-level transport primitives from the rest of the app. This way, we can better support modularity and testability of the project."

> **Interviewer**: "What protocols would you use for the network communication?"
> **Candidate**: "I would split everything into three major components:"

1. Bi-directional Communication Layer

> **Candidate**: "Real-time bi-directional communication layer for sending/receiving chat messages and status. We need to decide between connection-based (TCP) and connectionless (UDP) protocols."
> **Candidate**: "TCP-based clients establish a virtual connection and provide an ordered delivery guarantee by re-transmitting lost packets. It might be more expensive in terms of battery life (especially on flaky networks where the participants need to frequently repeat handshakes to restore lost connection). Another disadvantage is a 64k limit for the connection count for any given host port and a bigger packet header size compared to UDP. Some examples of TCP-based protocols: WebSocket (Slack), XMPP (WhatsApp, Zoom, Google Talk), MQTT (IoT, Smart Home, etc)."
> **Candidate**: "UDP-based clients are more lightweight and don't require any handshakes. As a result, UDP can't provide an ordered delivery guarantee and has no error checking beyond simple checksums. Some examples of UDP-based protocols: WebRTC (Discord, Google Hangouts, Facebook Messenger) - also works over TCP."

> *NOTE: You can learn more about the differences between TCP and UDP here.*

> **Candidate**: "I don't have much experience working with real-time protocols, so I select WebSockets to simplify the design. Some disadvantages of this choice - the protocol is schemeless and does not provide automatic reconnection. There are also several security flaws. Alternatively, we can use HTTP-polling - might be a poor choice since it would generate an excessive amount of backend traffic."
> **Candidate**: "We can also use gRPC (bi-directional streaming) or GraphQL (subscriptions), but I don't have enough experience with them."

> *NOTE: It is better not to bring unfamiliar technologies to the discussion. You can mention some of their advantages - but don't go too deep unless you're prepared to discuss implementation-specific details.*

> **Interviewer**: "How would you use WebSockets?"
> **Candidate**: "I only need them to send and receive real-time chat events. For everything else, we can use HTTP-based protocols."

> **Interviewer**: "Can you describe the event data format?"
> **Candidate**: "Sure, a typical data frame might look like this:"

```
{
  connection_id: String?
  event_type: "HELLO|MSG_IN|MSG_OUT|MSG_READ|BYE"
  payload: { ... }
}
```

Events:

- `HELLO` - initiates a new connection session (bi-directional).
- `MSG_IN` - incoming message.
- `MSG_OUT` - outgoing message.
- `MSG_READ` - message "read" acknowledgement (bi-directional).
- `BYE` - closes a connection session

> **Interviewer**: "What is a 'bi-directional' event?"
> **Candidate**: "That means the same event name could be sent both from the client and the server. For example, the client sends `HELLO` and the server responds with another `HELLO`. Alternatively, we can use prefixes like `CLT_HELLO` and `SRV_HELLO`".

> **Candidate**: "Another approach could be push-only events: the client uses `POST` or `PUT` HTTP requests to send the data to the server. The server can use WebSocket to push events to clients. The disadvantage of such approach - a client needs to create a new connection every time it needs to send an event."

> **Interviewer**: "Why do you need a `connection_id`? Where is it coming from?"
> **Candidate**: "To differentiate between clients on the backend. A client receives it with the `HELLO` event from the server."

> **Interviewer**: "How would you keep a socket connection when the app goes to the background?"
> **Candidate**: "I don't need to keep it alive while the app is not active - a push notification can be used to notify the user about new messages and bring the app back to the foreground."
> **Interviewer**: "Sounds good."

> *NOTE: It's tempting for iOS engineers to mention `URLSessionWebSocketTask` as a solution for bi-directional communication. But it's also important to remember that the API is not available until iOS 13. Make sure to keep OS version compatibility in mind.*

2. HTTP-based layer

> **Candidate**: "Request-response layer for fetching a paginated list of chats or message history. REST protocol could be a good choice since we don't need much of the request customization to bring up GraphQL (and have an extra complexity on the backend side)."

> **Interviewer**: "Could you briefly describe the API design?"
> **Candidate**: "We would need to have a couple of end-points:"

- `GET /login` - initiates a new client session and returns a JSON Web Token token for authorizing further requests.
- `GET /chats?after_id=<X>&limit=<Y>` - receives a paginated list of chats.
- `GET /chats/<chat_id>/messages?after_id=<X>&limit=<Y>` - receives a paginated list of messages from a specific chat.

> *NOTE: Avoid over-complicating the design; prioritize covering more ground unless prompted to delve deeper into specific protocols.*

> **Interviewer**: "Why do you need a JWT?"
> **Candidate**: "For client authentication: each request (besides `/login`) should include an authorization header: `Authorization: Bearer <token>`."

> **Interviewer**: "Can you explain these query params `messages?after_id=<X>&limit=<Y>`?"
> **Candidate**: "It's a cursor-based pagination: `before_id` and `after_id` contain the first/last message id in the current page; `limit` represents the maximum amount of items in a single page. Alternatively, we can use a keyset-pagination (using the timestamp of the last message) but it might not be reliable since we can't trust the client device clock."

> *NOTE: Learn more about pagination here.*

> **Candidate**: "We should also discuss some specific HTTP response codes."

- `401 Unauthorized` - the client authorization token is missing or expired: the `login` request must be sent before continuing.
- `422 Unprocessable Entity` - the client request data is malformed and should not be retried.

- `429 Too Many Requests` - the client reached the rate-limiting threshold.
- `500 Internal Server Error` - the client should use exponential back-off to retry a failed request.

3. Cloud Messaging Layer

> **Candidate**: "A typical push payload could look like this:"

```
{
  user_id: String
  messages: [
    {
      user_name: String
      text: String
      created_at: String
    },
    ...
  ]
}
```

> **Interviewer**: "Why do we need a `user_id` along with `user_name` ?"
> **Candidate**: "`user_id` contains the id of the recipient to make sure that nobody else would see the message. Imagine the situation when another user logs in on the same device and receives a push for the previously logged-in user."
> **Candidate**: "`user_name` contains a display name of the sender; `text` contains the first 100-120 characters of the message. This way we don't need to make any additional requests and can display a status bar notification right away."

API Service Diagram

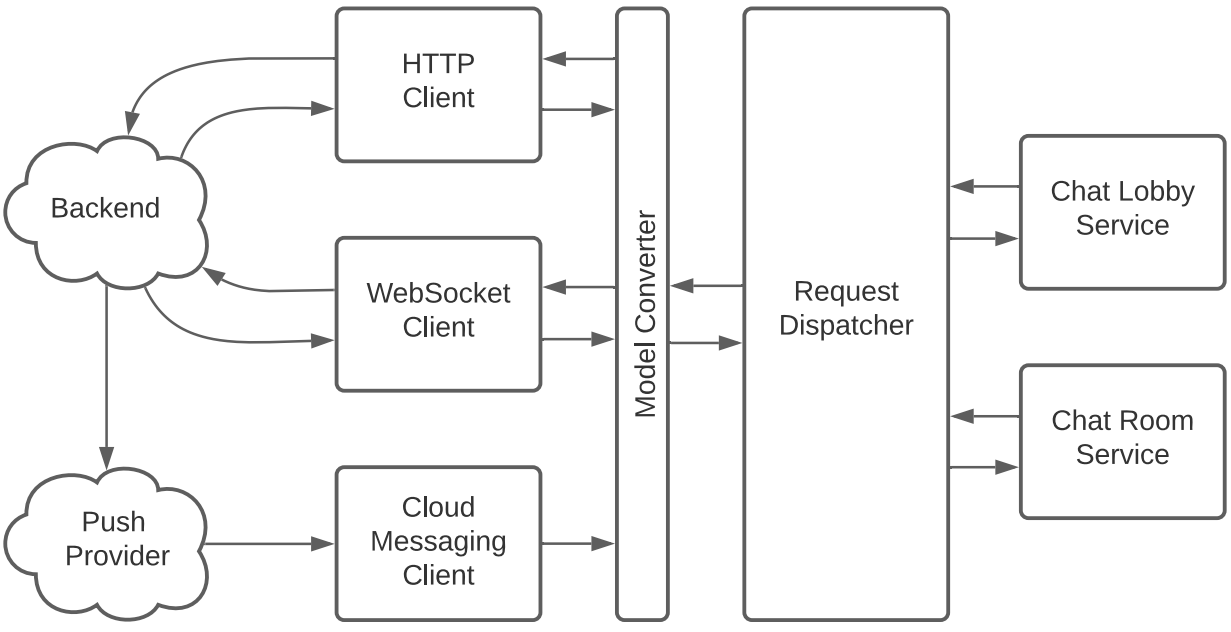


FIGURE 16.2 API Service Diagram

Candidate: "I would introduce separate interfaces (protocols) to abstract communication between the network layer and the business logic: `ChatLobbyService` (receive the list of chats, create/delete chats) and `ChatRoomService` (receive the message history, send/receive messages, download/upload attachments)."

Interviewer: "Why do you need to have separate interfaces?"
Candidate: "To decouple different modules of the app. For example, the `Chat Lobby` flow can receive the `ChatLobbyService` instance from DI and only use the public API contract without knowing anything about the underlying network module implementation. It also makes development/testing easier since developers can swap real implementation with fake instances and serve predefined data from disk or memory."

Candidate: "I would also use a `Model Converter` layer to turn network response classes into model classes to hide protocol/implementation details and perform simple data transformations."
Interviewer: "Not sure if I'm following - can you elaborate?"

Candidate: "Imagine that you received a data frame containing a chat message and deserialized it into an object."

```
ChatMessageData:
+ id: String
+ user_id: String
+ text: String
+ status: String
+ created_at: Long
+ attachments: String // comma-separated list
```

Candidate: "You can convert it into a model object."

```
ChatMessage
+ id: String
+ userId: String
+ text: String
+ status: ChatMessageStatus
+ createdAt: Date
+ attachments: [Attachments]
```

Candidate: "This way the business logic does not need to know about the network layer and the data format of the transport protocols. You can change the networking implementation without affecting the rest of the app."

Deep Dive: Data Model

> **Interviewer**: "How would you organize your data storage?"
> **Candidate**: "I would use a relational database (ORM) to store chats, messages, users, and attachments."
> **Candidate**: "Other alternatives might be less robust since we're looking for querying support and data integrity."

> *NOTE: Avoid spending time on approaches that are clearly unsuitable for the task, such as app preferences or basic text/binary files.*

> **Candidate**: "I would use a relational database (ORM) and create tables for chats, messages, users, and attachments."

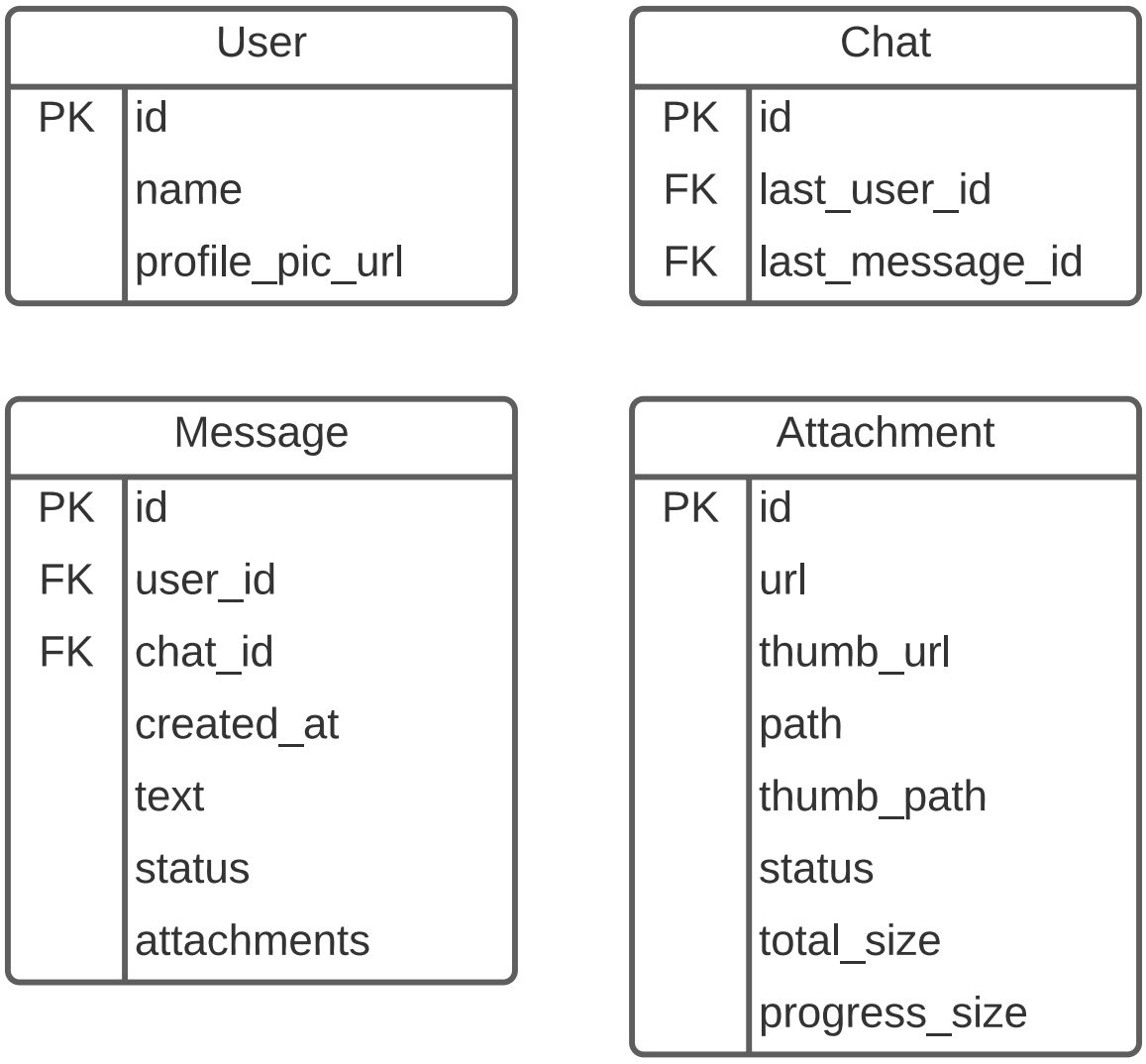


FIGURE 16.3 High-level Diagram

> *NOTE: This diagram loosely follows the entity-relationship diagram notation. The cardinality/ordinality relationships are purposely omitted to save the interview time. A good rule of thumb is to avoid unnecessary drawings altogether.*

Candidate: "The `User` table is pretty straightforward: we would store user id, name, and profile picture URL."

Candidate: "The `Chat` table would contain the list of active chats: we would store the chat id, the id of the user who sent the last message, and the last message id."

Candidate: "The `Message` table would contain all messages from all chats: we would store the message id, the user id, the chat id, the message status (`PENDING` , `SENT` , `READ` , `FAILED`), and a comma-separated list of attachment ids."

Candidate: "The `Attachment` table would contain all attachments for all messages from all chats: we would store the attachment id, the image URLs (full-size and the low-res), the cached local versions paths, the attachment status (`UPLOADING` , `DOWNLOADING` , `FAILED` , `READY`), the total file size and the progress size (uploaded/downloaded bytes).

Candidate: "We can join the `Chat` table with the `Message` table (on the `last_message_id` column) and with the `User` table (on the `last_user_id` column) to get a list of recent chats. Each result could be represented by the following model class:"

```
ChatInfo:
+ chatId: String
+ lastUsername: String
+ lastUserProfileUrl: Url
+ lastMessageText: String
+ lastMessageTimestamp: Date
```

Candidate: "We can get the list of messages for a specific chat by selecting on the `chat_id` column."

Interviewer: "Joining three tables every time you open the app might be slow. How would you optimize it?"
Candidate: "Good point. On mobile devices, complex JOINs can lead to dropped frames during scrolling. We can denormalize the `Chat` table by adding `last_message_preview` and `last_message_timestamp` columns. This way, we can fetch the entire lobby list with a single query without any JOINs."

> **Interviewer**: "What would you use for message and attachment ids?"
> **Candidate**: "What do you mean?"

> **Interviewer**: "Are those local ids or server ids?"
> **Candidate**: "We can use client-generated 128-bit UUIDs. The outgoing messages can be identified by `user_id` and the outgoing attachment by empty `url` columns: once an attachment is uploaded - it's undistinguished from a remote attachment."
> **Candidate**: "The advantage of such approach is its simplicity and built-in idempotency; the disadvantage - clients would generate resource ids which is less reliable and less secure compared to the backend generation."

> **Candidate**: "Alternatively, we can maintain separate local and server ids: all local operations would be using local ids while the backend communication would use server ids. We would also need to build a bijection between them."

> *NOTE: Learn more about Local and Servier ids here.*

🔍 Data Model Analysis

Key Insight: The "Two-ID Problem" discussion is the highlight here. Mobile apps must function offline. If an app relies solely on Server IDs, a user can't send a message when offline (because there's no server response yet). Using a local ID (UUID) or a temporary ID allows the UI to update immediately ("Optimistic UI"), while the sync happens in the background.

Deep Dive: Attachments

> **Interviewer**: "How would you handle attachments?"
> **Candidate**: "I'll keep the incoming/outgoing attachments metadata in the same table to simplify the message queries - this way we would only need to join on a single table. Alternatively, we can create a separate table for outgoing attachments but this might complicate the overall design."
> **Candidate**: "The state of the attachments would be controlled by the `status` column. The `READY` status corresponds to an outgoing attachment that has been successfully uploaded or to an incoming attachment that has been successfully downloaded. Other status values - for handling uploads/downloads/failures."
> **Candidate**: "We also need to keep track of uploading/downloading progress to support resumable uploads/downloads."

> **Interviewer**: "How would you distinguish between incoming and outgoing attachments?"
> **Candidate**: "We don't need to keep them distinct - only keep track if the attachment needs to be downloaded or uploaded. This can be determined by the `url` column: it will be `NULL` for outgoing attachments."

> **Interviewer**: "What component would download/upload attachments?"
> **Candidate**: "The API Service via HTTP client. We would also need a task dispatcher to limit the number of the concurrent network operations."

> *NOTE: For more information on task dispatchers see this.*

> **Candidate**: "We can automatically download incoming attachments on WiFi and ask for user input on cellular networks. The application settings might be provided for better user experience."

Follow-up Questions

Some interviewers might ask follow-up questions that might change the original design and introduce new requirements.

Timestamps

> **Interviewer**: "How would you handle timestamps?"
> **Candidate**: "This is a tricky question. I would probably use UTC timestamps (not adjusted to daylight saving and timezones) for messages and just format them to the localized time before displaying."

> **Interviewer**: "Would you use the client or the server time?"
> **Candidate**: "I would use the client time for outgoing messages and the server time for incoming. For example, when a user sends a message - the client local UTC time would be applied. When the server receives a message - it would assign the server local UTC time so the other chat participant would see it as a timestamp."
> **Interviewer**: "Why won't you use the client time for both cases?"
> **Candidate**: "We can't trust the client time: if the device clock reports incorrect time - we can end up with messages from the future."
> **Candidate**: "There's still a chance for the wrong message ordering. For example, when the user sends messages while being offline."

> **Interviewer**: "So how do we solve the ordering problem effectively?"
> **Candidate**: "We can use **Logical Clocks** (like Lamport timestamps) or a simple `sequence_number` per chat. This helps us detect gaps (e.g., 'I have message 5, I need 6, but received 7'). Relying solely on 'Wall Clock' time is risky due to clock skew."

> *NOTE: Ensuring a proper message order is a tricky problem: it's pretty unlikely for a candidate to know a solution unless they've been working on chat apps in the past.*

Security & Privacy

> **Interviewer**: "Where would you store chat messages?"
> **Candidate**: "I would need to have a cached copy on the device for offline viewing. We can also store it on the backend but it imposes privacy issues even if we encrypt them."

> **Interviewer**: "What kind of issues are you talking about?"
> **Candidate**: "Even if the messages are encrypted on the backend - the developer still has an access to encryption keys so an attacker who has the access to the servers can still compromise user privacy."
> **Candidate**: "Additionally, the **user perception** of privacy is as important as the privacy implementation itself. If the developer has the access to the messages - the customers might assume that those messages would be read without any consent."
> **Candidate**: "An end-to-end encryption could be used but I don't have any experience with the algorithms for key exchange."

> *NOTE: For more information about messaging privacy check WhatsApp Encryption Overview.*

Major Concerns and Trade-Offs

Connectivity

- Maintaining a permanent WebSocket connection is expensive in terms of battery life.
- It also requires a mechanism to restore the connection after network failures.
- We might want to use a "heartbeat" mechanism to keep the connection alive (ping/pong frames).

Push Notifications

- Push notifications are not 100% reliable.
- We cannot rely on them to deliver critical information (like message delivery status).
- We should use them only to notify the user about new messages.

Storage

- Storing all messages and attachments on the device might quickly consume all available disk space.
- We need to implement a retention policy to delete old messages/attachments.
- We can also offload old messages to the cloud and only keep the recent ones on the device.

Conclusion

Keep this in mind while preparing for a system design interview:

- Prioritize providing a "signal" over achieving perfection.
- Listen to your interviewer and keep track of the time.
- Try to cover as much ground as possible without digging too much into the implementation details.

Looking for more content?

I'm thinking about creating an in-depth mobile system design course on top of the free articles. Please, fill out this form to express your interest!

PART IV

Preparation Toolkit

Practical preparation material: an interview note-taking template, common mistakes to avoid, and curated further reading.

IN THIS PART

CHAPTER 17

Mobile System Design Interview Template

Use this template to structure your thoughts and notes during the interview.

1. Requirement Gathering (5 min)

Goal: Clarify scope and identify the "MVP".

Questions to Ask:

- **Users:** DAU/MAU? Spike traffic patterns?
- **Geography:** Target market? (Network speed, data costs, device capabilities).
- **Platform:** iOS, Android, or both? Minimum OS version?
- **Team:** Team size? (Influences modularity).
- **Features:** What are the top 3-5 critical features?

Functional Requirements:

1.
2.
3.

Non-Functional Requirements:

- **Offline Support:** (Yes/No/Partial)
- **Performance:** (Latency, startup time)
- **Security:** (E2E encryption, local storage encryption)
- **Reliability:** (Data consistency, conflict resolution)

Out of Scope:

- (e.g., Auth, Settings, Analytics, Web version)

---

2. High-Level Diagram (10-15 min)

Goal: visual "Big Picture". Identify key components.

Core Components:

- **UI Layer:** (View, ViewModel/Presenter)
- **Navigation:** (Coordinator/Router)
- **Domain Layer:** (Use Cases/Interactors)
- **Data Layer:** (Repository)
  - **Remote:** (API Service -> Retrofit/Alamofire)
  - **Local:** (Persistence -> Room/CoreData)
- **DI Graph:** (Dependency Injection)
- **Image Loader:** (Coil/SDWebImage)

Sketch: (Draw boxes and arrows here. Focus on data flow: UI -> ViewModel -> Repository -> API/DB)

---

3. Deep Dives (20-30 min)

Select 1-2 topics to explore.

A. API Design

- **Protocol:** REST vs GraphQL vs WebSocket vs gRPC.
- **Pagination:** Offset vs Cursor vs PageNumber.
- **Endpoints:**
  - `GET /resource`
  - `POST /resource`
- **Payloads:** (JSON structure, field types)

B. Data Storage

- **Schema:** (Tables, Columns, Relationships)
- **Sensitive Data:** (Keychain/Keystore)
- **File Storage:** (Images/Videos -> Internal vs External cache)

C. Synchronization (Offline Support)

- **Strategy:** Write-Through vs Write-Back?
- **Conflict Resolution:** Last-Write-Wins vs Server-Side vs User-Manual?
- **Queue:** Request Queue / Worker Manager for background sync.

D. Media Handling

- **Images:** Caching (Memory/Disk), Resizing, Formats (WebP).

- **Uploads:** Resumable/Chunked uploads?

---

4. Signal Checklist

- ☐ Did I mention **Trade-offs**? (e.g., "WebSockets are faster but harder to scale than polling").
- ☐ Did I mention **Edge Cases**? (e.g., "What if the network drops in the middle of an upload?").
- ☐ Did I mention **Platform Specifics**? (e.g., "On Android, we need to handle process death").
- ☐ Did I check assumptions with the interviewer?

CHAPTER 18

Common Interview Mistakes

This guide expands on common mistakes made by candidates and interviewers during mobile system design interviews, specifically focusing on iOS and Android platforms. The aim is to provide candidates with a practical understanding of expectations and help them prepare effectively.

As a Candidate

Not Being Prepared

Preparation is key. Don't just memorize patterns; understand the *why* behind them.

- **Study Open-Source Projects:**
  - **Beyond the Code:** Look beyond the UI and try to understand the architectural choices, networking layers, data persistence strategies, and error handling.
  - **Platform-Specific Patterns:** Pay attention to iOS-specific patterns like `Coordinator` pattern, `Combine` framework, or `SwiftUI` integration with `UIKit`. For Android, focus on `MVVM` with `LiveData` / `Flow`, `Coroutines`, `Jetpack Compose`, and dependency injection frameworks like `Hilt` or `Dagger`.
  - **Example Resources:**
    - **iOS:** Look into popular open-source iOS apps or libraries on GitHub (e.g., a well-architected media player or a networking library). Analyze how they handle background tasks, push notifications, or data synchronization.
    - **Android:** Explore apps like AntennaPod or projects using the Architecture Components. Focus on understanding background processing (WorkManager), UI rendering (RecyclerView optimizations), and data management (Room, Retrofit).
  - **Focus on Architecture:** Identify common architectures like `MVVM`, `MVP`, `Clean Architecture`, and understand their trade-offs in the context of mobile development.
- **Study Company Dev Blog:**
  - **Tech Stack Insights:** Company blogs often reveal details about their tech stack, chosen architectural patterns, and solutions to specific technical challenges.
  - **Identify Priorities:** Understand the company's priorities, such as performance optimization, security, or scalability, which might be reflected in their interview questions.
  - **Specific Technologies:** Look for articles about the specific languages (Swift/Kotlin), frameworks (`UIKit` / `SwiftUI` vs. `Compose`), and libraries they use. If they've written about challenges with certain technologies, that's a *major* hint.

  - **Example:** If a company's blog discusses their migration from Objective-C to Swift, be prepared to discuss Swift interoperability, memory management, and the benefits of Swift.

- **Conduct Mock Interviews with your peers:**
  - **Realistic Scenarios:** Simulate real-world scenarios with your friends. This could include designing a chat application, a news feed, or an e-commerce platform. Ensure you cover areas like offline support, data synchronization, and push notifications.
  - **Platform-Specific Concerns:** In iOS mock interviews, consider questions around memory management (reference cycles), UI responsiveness, background execution limits, and Core Data/Realm integration. For Android, address issues like activity lifecycle management, background services, battery optimization, and handling different screen sizes and densities.
  - **Time Management:** System design questions can be broad. Practice timeboxing each part of the discussion (requirements gathering, high-level design, component breakdown, etc.).
  - **Communication is Key:** The mock interviewer should provide feedback not only on your technical solution but also on your communication skills, clarity, and ability to explain complex concepts.
- **Practice Handling Novelty:**
  - **Don't Freeze on Niche Topics:** If asked to design something unexpected (e.g., an SDK, a Library, or Server-Driven UI), don't panic. Fall back to first principles: Define the API, the Data Format, and the Storage.
  - **Practice by Doing:** Do not just read guides. Open the App Store, pick a random feature from a top 10 app (e.g., Spotify Player, Uber Map), and spend 30 minutes trying to design it yourself. This builds the intuition needed to handle unique problems rather than reciting memorized solutions.

Rushing Towards a Solution

Take a step back and think!

- **Not Gathering System Requirements:**
  - **User Base:** "How many users will the app support initially? What is the expected growth?"
  - **Platform:** "Are we targeting only iOS or Android, or both? What are the minimum supported OS versions?"
  - **Features:** "Which features are core to the app's functionality? Which are optional or can be deferred?"
  - **Non-Functional Requirements (NFRs):** "What are the key non-functional requirements like performance, security, scalability, and reliability? Are there any specific privacy concerns (GDPR, HIPAA)?"
  - **Data Volume & Constraints:** Don't just ask about users; ask about data. "What is the maximum file size for uploads?" (Critical for apps like Dropbox). "Are we storing stock history for 1 day or 10 years?"
  - **Example:** Don't assume "build a chat app." Ask about: Message volume? Media support? Group chats? End-to-end encryption? Offline access?

- **Not Asking Clarifying Questions:**
  - **Target Market Constraints:** Understand the geographical location. If the target is a developing market, you must design for spotty internet, high data costs, and lower-end device storage. Consider peer-to-peer sharing (Bluetooth) or data-saver modes.
  - **Audience Size:** "Is this for a small team, a large enterprise, or the general public?" The scale of the audience will impact the architectural decisions around backend infrastructure and scalability.
  - **Dev Team Details:** "What is the size and expertise of the development team?" (A large team requires more modularization to avoid merge conflicts; a small team benefits from simplicity).
  - **Device Capabilities:** "What is the minimum device capability we want to support? This will determine the memory available, processing power and supported OS features. It is important for memory management and performance optimizations."
  - **Product Specifics:** If designing a known product (e.g., Trello, Spotify), do not assume you know how it works. Ask: "Do columns move?" "Does the music stop if I go offline?"
  - **Example:** "What kind of data are we storing? Is it sensitive personal data that requires encryption at rest?"
- **Jumping Straight to Implementation:**
  - **High-Level Design First:** Start with a high-level overview of the system architecture, outlining the key components, their responsibilities, and how they interact with each other.
  - **Avoid the "UI Trap":** A common mistake is spending 20 minutes drawing ViewControllers, Screens, or Lists. This provides very little system design signal. Focus on the "Hard Parts" (Sync Engine, Pagination Logic, Conflict Resolution) instead of the UI layer.
  - **UI Architecture Patterns:** Delay discussing specific view classes or UI architecture patterns (MVC, MVVM, VIPER, MVI) until you've established the overall system design.
  - **Example:** Instead of immediately discussing `UITableView` or `RecyclerView`, talk about the overall data flow, data models, networking layer, and caching strategy.
  - **Vendor-Specific Solutions:** Avoid mentioning vendor-specific solutions at this stage. For example, don't say "We'll use Firebase for push notifications" before discussing the overall push notification architecture. Discuss the requirements and different potential approaches for achieving the goal.

The "GitHub Template" Syndrome

One of the most frequent negative signals is trying to fit a pre-memorized solution into a specific problem.

- **Avoid Generic Diagrams:** Drawing standard boxes labeled "RemoteDataSource," "Repository," "UseCase," and "ViewModel" provides zero signal because almost every app has these. It looks like you memorized a template from a guide.
- **System Components > Code Patterns:** Instead of class diagrams, draw *System Components* that match the specific problem. For example: "Sync Engine," "Upload Manager," "Conflict Resolver," or "Image Processing Queue."
- **Justify Abstractions:** Do not add a "Repository" or "Dependency Injection" box just because a blog post said so. If the app is a simple map viewer, a complex repository layer might be over-engineering.

Scope Creep & Over-Engineering

Balancing what to build vs. what *not* to build is a senior skill.

- **Scope Creep:** Do not voluntarily add complex features that weren't asked for (e.g., "Let's also add video editing," "Let's add AI recommendations," or "Let's add real-time user status"). This invites hard questions you might not be ready for and wastes valuable time.
- **Failure to Negotiate:** If the stakeholder (interviewer) asks for a complex feature with low business value (e.g., "Real-time updates for every 'like' on a post"), a Senior/Staff engineer should push back. Explain the high technical cost vs. low user value and propose a simpler alternative (eventual consistency).
- **YAGNI (You Ain't Gonna Need It):** Don't design for hypothetical futures (e.g., "What if we need to support VR headsets later?"). Focus on the MVP first.
- **Avoid "Rube Goldberg" Solutions:** Don't create complex chains of events (e.g., "Task/Job/Query/Command" architectures) for simple problems. Start simple and add complexity only when the requirements demand it.

Being Unresponsive

Communicate clearly and constantly.

- **Keeping Silence:**
  - **Think Aloud:** Verbally articulate your thought process, even if you're unsure of the solution. Silence provides no signal to the interviewer.
  - **Explain Your Reasoning:** Explain the rationale behind your design decisions, highlighting the trade-offs and alternatives considered.
  - **Example:** Instead of silently pondering, say, "Okay, I'm thinking about using a message queue for handling background tasks. This would allow us to decouple the UI from the task execution and improve responsiveness. However, it also adds complexity. Let me explain further."
- **Waiting Until the Interviewer Starts Asking Questions:**
  - **Drive the Discussion:** Take initiative by presenting your solution in a structured manner, covering all key aspects of the design.
  - **Anticipate Questions:** Anticipate potential questions from the interviewer and proactively address them in your explanation.
  - **Example:** After presenting the high-level architecture, say, "Now, I'd like to dive into the data synchronization strategy and discuss how we can handle conflicts. This is an area where we need to consider different approaches based on the requirements."

Giving Up Early

Don't let a single roadblock derail you.

- **Persistence is Key:** System design interviews are challenging. If you encounter a problem, don't give up. Try to identify alternative approaches or break down the problem into smaller, more manageable parts.
- **Never say "We can't do anything":** If asked a tough edge case (e.g., "What if the user never connects to the internet?"), do not say "Then we can't do anything." Speculate on alternatives: Desktop sync? Peer-to-peer sharing? SD card transfer?
- **Focus on the Process:** Even if you don't arrive at the perfect solution, demonstrate your problem-solving skills, critical thinking, and ability to learn from mistakes.

Talking Too Much & "Title Dropping"

Be concise and relevant.

- **Long Introduction:**
  - **Focus on Relevance:** Keep your introduction brief (1-2 minutes). Focus on the experiences that are directly relevant to the role (e.g., mobile dev) rather than discussing irrelevant history (e.g., C++ gaming from 10 years ago).
  - **Highlight Key Achievements:** Highlight key achievements and projects that demonstrate your expertise in mobile system design, architecture, and development.
- **Avoid Self-Aggrandizement:** Avoid introducing yourself as a "Software Architect," "Expert," or "Guru". It invites the interviewer to grill you harder to prove the title. Be humble and describe your actual responsibilities.
- **Trying to Force an Interviewing Framework:**
  - **Adapt to the Interviewer:** Be flexible and adapt to the interviewer's style and preferences. Don't rigidly adhere to a specific framework if it doesn't align with the flow of the conversation.
  - **Use as a Guide:** Treat frameworks as a guide, not a strict protocol.
- **Ignoring the Interviewer:**
  - **Listen Carefully:** Pay close attention to the interviewer's questions, comments, and suggestions. They may be providing valuable hints or guidance.
  - **Pause for Interruption:** If the interviewer interrupts you, stop talking and listen to what they have to say. Don't try to finish your thought at the expense of their input.
- **Repeating Yourself:**
  - **Move Forward:** Once you've clearly explained a concept or decision, move on to the next topic. Don't reiterate the same points without adding new information.
- **Jumping from Topic to Topic:**
  - **Structured Approach:** Maintain a logical flow in your presentation, covering each topic in a clear and organized manner.

    - **Transition Smoothly:** Use transitional phrases to guide the interviewer through your thought process and indicate when you're moving on to a new topic.

- **Going Too Broad with the Answers:**
    - **Stay Focused:** Answer the questions directly and avoid tangents that are not relevant to the topic at hand.
    - **Provide Meaningful Details:** Provide sufficient detail to demonstrate your understanding of the concepts, but avoid getting bogged down in unnecessary minutiae.

Treating the Interviewer as an Adversary

The interviewer is on your side (or should be!).

- **Collaboration, Not Confrontation:** Remember that the interviewer is trying to assess your skills and potential fit within the team. Treat the interview as a collaborative problem-solving exercise, not an adversarial interrogation.

Trying to Fit a Solution into an Existing Scheme

Don't force a square peg into a round hole.

- **Tailor Your Approach:** Design your solution based on the specific requirements of the problem, not on pre-conceived notions or existing designs.
- **Flexibility is Key:** Be flexible and willing to adapt your approach if new information emerges or the interviewer provides feedback.

Being Toxic

No one wants to work with someone unpleasant.

- **Being Opinionated:**
    - **Open-Mindedness:** Acknowledge that there are multiple valid approaches to solving a problem and be open to considering alternatives.
    - **Justify Your Choices:** Provide a rationale for your preferred approach, but avoid dismissing other options out of hand.
- **Interrupting the Interviewer:**
    - **Respectful Communication:** Allow the interviewer to finish their thought before interjecting with your own opinions or questions.
- **"Educating" the Interviewer:**
    - **Humility:** Avoid lecturing the interviewer or assuming that you know more than they do.
    - **Polite Disagreement:** If you disagree with something the interviewer says, express your disagreement politely and respectfully, providing evidence or reasoning to support your point of view.

- **Being a "Pleaser":**
  - **Authenticity:** Be genuine and authentic in your responses. Don't try to tell the interviewer what you think they want to hear.

Technical Competence & Justification

Show your thought process and avoid "faking it."

- **Provide Justification:** Always explain the reasoning behind your design decisions.
  - **Avoid "Documentation" Answers:** Don't say "I used MVC because Apple recommends it." Say "I used MVC because it separates concerns, allowing us to test the logic independently of the UI."
  - **Avoid "Simplicity" as a Crutch:** Don't say "I used Shared Preferences because it's simple/native" if the requirement (e.g., large dataset) clearly demands a Database.
- **Faking Knowledge:**
  - **Don't Bluff:** If you don't know a specific technology (e.g., "How does Protobuf schema evolution work?"), **admit it**. Trying to bluff or dodge the question is a major red flag.
  - **Don't Guess:** If asked about backend databases, don't guess "Cassandra on mobile." Admit you aren't sure about the specific vendor but explain the *properties* you need (e.g., NoSQL, key-value store).
- **Buzzword Dropping:**
  - **Depth over Keywords:** Don't throw out terms like "GraphQL," "WebSockets," or "Clean Architecture" unless you can explain exactly *why* they solve a specific problem in *this* design better than the alternatives.
  - **Know Your Tools:** Using "Dependency Injection" for "performance" is a wrong answer. Know the real purpose of the patterns you cite.
- **Technical Missteps:**
  - **Client vs. Server:** Do not suggest performing heavy, trusted logic (e.g., conflict resolution, timestamp generation for ordering) on the client. The client is unreliable and clock-skewed.
  - **Security:** Do not suggest "Asymmetric Encryption" for local storage (it's slow/overkill). Do not suggest storing sensitive data if you can just keep it in RAM.
  - **Consider Alternatives:** For each key decision, briefly discuss alternative approaches and explain why you chose the specific solution you did.

Lying

Honesty is always the best policy.

- **Be Truthful:** Never exaggerate your skills or experience. If you don't know the answer to a question, it's better to admit it than to make something up.
- **Focus on What You Know:** Focus on highlighting your strengths and areas of expertise, rather than trying to cover up your weaknesses.

Lack of Decisiveness (Asking for Permission)

- **Be the Driver:** The most common behavioral mistake is constantly asking: "Is this okay?", "Should we support offline mode?", or "Do you think I need a database?"
- **State Your Intent:** A Senior/Staff candidate leads. Say: "I am going to support offline mode because this is a music player and users expect it," or "I will use UUIDs for IDs to prevent collisions."
- **Avoid "Hint Seeking":** Constantly looking to the interviewer for validation or hints signals a lack of confidence.

Not Having Any Structure for the Solution

Organize your thoughts.

- **Prepare an Outline:** Before the interview, prepare a general outline of the topics you want to cover.
- **Time Management (Mental Timer):**
  - **Requirements:** ~5 minutes. (Don't spend 15 minutes here).
  - **High-Level Design:** ~10-15 minutes.
  - **Deep Dive:** ~20-25 minutes.
- **Use a Framework:** Consider using a system design framework as a guide, but be flexible and adapt it to the specific requirements of the problem.

Speaking in Terms of Specific Vendors

Focus on core principles.

- **Abstract Away Implementation Details:** Focus on the underlying concepts and principles of system design, rather than getting bogged down in the specifics of particular vendors or technologies.
- **Vendor-Agnostic Language:** Use vendor-agnostic language to describe your solutions, focusing on the functionality and behavior of the system.

Being Overly Optimistic

Consider edge cases and failure scenarios.

- **Think About Failure:** Consider potential failure scenarios, such as network outages, server errors, or device crashes, and design your system to handle these situations gracefully.
- **Error Handling:** Discuss error handling strategies, such as retries, fallbacks, and circuit breakers, to ensure the reliability and resilience of the system.
- **Example:** "What happens if the user loses network connectivity while uploading a large file? How do we handle data inconsistencies if there's a conflict between the local cache and the server?"
- **Backward Compatibility:** Specifically for SDKs or new features, ask: "What happens if the user has an old version of the app?" "How do we handle API changes without breaking existing clients?"

Making Assumptions

Don't guess; ask.

- **Clarify Ambiguities:** If you're unsure about something, ask clarifying questions to ensure you understand the requirements and constraints of the problem.
- **State Your Assumptions:** If you have to make assumptions, explicitly state them and explain why you're making them.

Ignoring the Interviewer's Hints

The interviewer wants you to succeed.

- **Pay Attention to Clues:** The interviewer may provide hints or suggestions to guide you in the right direction. Pay attention to these clues and use them to refine your solution.
- **Ask for Clarification:** If you're unsure about a hint, ask the interviewer to clarify it or provide more context.

As an Interviewer

Your role is to evaluate fairly and help the candidate showcase their abilities.

Not Being Prepared

- **Well-Defined Questions:** Have a clear understanding of the problem you want the candidate to solve and the key aspects of system design you want to evaluate.
- **Potential Discussion Topics:** Anticipate potential discussion topics and prepare follow-up questions to probe the candidate's understanding and explore different aspects of the design.
- **Ideal Solutions:** Have a general idea of the ideal solution and the trade-offs involved, but be open to alternative approaches and creative solutions.
- **iOS & Android Specifics:** For mobile system design, you *must* understand the OS specific constraints (Battery, Network, Lifecycle, Storage limits) and APIs to realistically assess a candidate. If you are a backend engineer interviewing a mobile candidate, ensure you don't penalize them for client-side specific patterns you aren't familiar with.

Understanding Levels

- **Differentiate Candidates:**
  - **L4 (Mid-level):** Focus on the "Happy Path." Can they build a working solution?
  - **L5 (Senior):** Focus on the "Unhappy Path." Bad network, low disk space, edge cases, conflict resolution.
  - **L6 (Staff):** Focus on "Business & Crisis." Privacy lawsuits, team bottlenecks, OS breaking changes, long-term maintenance, and business value.

- **Allow "Stubbing":** If a candidate doesn't know a specific backend detail (e.g., specific DB sharding), allow them to abstract it ("Let's assume the backend handles this") so you can focus on their mobile expertise.

Being Toxic

- **Respectful Demeanor:** Treat the candidate with respect and create a positive and supportive environment for them to showcase their skills.
- **Avoid Belittling:** Avoid making demeaning or condescending remarks, even if the candidate is struggling to answer a question.

Being Unresponsive

- **Active Listening:** Pay close attention to the candidate's explanations, ask clarifying questions, and provide constructive feedback.
- **Engagement:** Show genuine interest in the candidate's ideas and demonstrate that you're actively engaged in the conversation.

Being Too Engaged

- **Let the Candidate Lead:** Allow the candidate to lead the discussion and explore different aspects of the design on their own.
- **Provide Guidance, Not Solutions:** Provide gentle guidance and hints when the candidate is stuck, but avoid giving away the answer or forcing them into a specific solution.

"Grilling" the Candidate

- **Relevant Questions:** Focus on asking questions that are directly relevant to the job requirements and assess the candidate's ability to solve real-world problems.
- **Avoid Gotcha Questions:** Avoid asking obscure or trivial questions (e.g., specific syntax or method names) that are designed to trip up the candidate.

Asking Meaningless Questions

- **Evaluate Core Skills:** Focus on questions that assess the candidate's understanding of core system design principles, such as scalability, reliability, and security.
- **Avoid Redundant Questions:** Avoid asking questions that have obvious answers or don't provide any meaningful insights into the candidate's abilities.
- **Focus on Decisions:** Ask "Why?" constantly. "Why GraphQL over REST?" "Why Image sizes in the URL?" "Why a Repository pattern here?"

Not Moving On When the Candidate is Stuck

- **Provide Hints:** If the candidate is stuck on a particular problem, provide a hint or suggestion to help them get unstuck.

- **Shift Focus:** If the candidate is unable to make progress after several attempts, move on to a different topic to avoid wasting time.

Forcing the Candidate into a Solution They Have in Mind

- **Open to Alternatives:** Be open to alternative solutions and creative approaches, even if they differ from your own preferred solution.
- **Evaluate Based on Rationale:** Evaluate the candidate based on the rationale behind their decisions, not just on whether they arrived at the "correct" answer.

Digging Too Much into Details (BFS Approach)

- **Prioritize High-Level Understanding:** Focus on assessing the candidate's understanding of the overall system architecture and key design principles, rather than getting bogged down in implementation details.
- **Balance Depth and Breadth:** Strike a balance between exploring specific topics in detail and covering a broad range of topics relevant to the system design.

CHAPTER 19

Curated List of Real-world Design Blog Posts

Real world examples where companies discuss trade-offs of certain technologies and how they design things to fit their use case. Contains a selection of posts that may be targeted to a platform, but concepts should be relevant to all

Here is a big list of company engineering blogs

Company Engineering Blogs

- **Uber:** Mobile Engineering
- **Meta (Facebook/Instagram/WhatsApp):**
  - iOS Engineering
  - Android Engineering
- **Square (Cash App):** The Code
- **Dropbox:** Mobile
- **Reddit:** r/RedditEng
- **Airbnb:** Mobile
- **Microsoft:** Mobile Engineering

Backend

- Airbnb
  - How Airbnb is Moving 10x Faster at Scale with GraphQL and Apollo
- Instacart
  - Building Instacart's view model API - Part 1: Why view model?
- Netflix
  - It's All A/Bout Testing: The Netflix Experimentation Platform

API Design

- Slack
  - How We Design Our APIs at Slack
  - Evolving API Pagination at Slack

- Trello
  - A look at Trello: adopting GraphQL and Apollo in a legacy application

Build / Platform

- Airbnb
  - Designing for Productivity in a Large-Scale iOS Application
- Facebook
  - Rethinking Android app compilation with Buck
- Reddit
  - iOS and Bazel at Reddit: A Journey
- Uber
  - The Journey To Android Monorepo: The History Of Uber Engineering's Android Codebase Organization
- Various
  - Monorepo Discussion

Caching / Offline

- Instagram
  - Building an Open Source, Carefree Android Disk Cache
- Trello
  - Airplane Mode: Enabling Trello Mobile Offline
  - Syncing Changes
  - Sync Failure Handling
  - The Two ID Problem
  - Offline Attachments
  - Sync is a Two-Way Street
  - Displaying Sync State
- Wikimedia
  - Designing for offline on Android

Emerging Markets

- Facebook
  - How we built Facebook Lite for every Android phone and network

- Microsoft
  - Microsoft Teams - Designing for Emerging Markets - Part 1 (Network Profile)
- Spotify
  - How We Built It: Spotify Lite, One Year Later
- Uber
  - Expanding Access: Engineering Uber Lite

Networking / Performance

- Dropbox
  - Making camera uploads for Android faster and more reliable
- Instagram
  - Improving performance with background data prefetching
  - Making Direct Messages Reliable and Fast
- Uber
  - How Uber's New Driver App Overcomes Network Lag

Networking / Reliability

- SnapChat
  - QUIC at Snapchat
- Uber
  - Engineering Failover Handling in Uber's Mobile Networking Infrastructure
  - Employing QUIC Protocol to Optimize Uber's App Performance

Server-Driven UI

- AirBnB
  - A Deep Dive into Airbnb's Server-Driven UI System
- DoorDash
  - Improving Development Velocity with Generic, Server-Driven UI Components
- Uber
  - Building the New Uber Freight App as Lists of Modular, Reusable Components
  - Architecting a Safe, Scalable, and Server-Driven Platform for Driver Preferences with RIBs
- Various - Strategies Discussion
  - Server-driven UI (or Backend driven UI) strategies

Cross-Platform

- Cash App
  - Cash App Case Study: Kotlin Multiplatform for Shared Business Logic
- Dropbox
  - The (not so) hidden cost of sharing code between iOS and Android

AI / Machine Learning

- Google
  - On-Device Machine Learning
- Meta
  - PyTorch Mobile

Architecture

- Martin Fowler

PART V

About This Guide

Contribution guidelines and the license governing the original material.

IN THIS PART

CHAPTER 20

Contributing

Thanks for helping out the project! You're awesome!

All the contributors to the `mobile-system-design` repo should sign the Individual Contributor License Agreement (CLA). This form ensures we can provide a solid open source project and makes life easier.

Please, make sure to agree to the terms and conditions before sending a pull request.

CHAPTER 21