Mastering the Android System Design Interview

© Stefan Jaindl

Overview

1. Networking
2. Storage
3. Building for Billions
4. Interview Tips & Tricks
5. Mock Interviews
6. Final Notes

© Stefan Jaindl

Networking

HTTP, COMMUNICATION, PAGINATION

© Stefan Jaindl

Networking

OVERVIEW

- HTTP evolution
- Client-Server communication methods / protocols
- Pagination
- Further topics

© Stefan Jaindl

HTTP Evolution

HTTP/0.9

- The one-line protocol
- Proposed 1989 by Tim Berners-Lee
- GET http://www.example.com:8080
- Supported format: HTML

© Stefan Jaindl

HTTP Evolution

HTTP/1.0

- HTTP request + response headers
- Response status
- Different supported file formats
- Different modifications of servers and browsers not agreed-upon
- Network Working Group published HTTP/1.0 (RFC 1945) in 1996

GET /contact HTTP/1.0
Host: www.myservice.com
User-Agent: Mozilla/5.0 (Linux; Android 12)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/100.0.4896.127 Mobile Safari/537.36
Accept: application/json
Accept-Language: en-us
Connection: keep-alive

© Stefan Jaindl

HTTP Evolution

HTTP/1.1

- Persistent & pipelined connections
- Virtual hosting
- Content negotiation
- Character set & language tags
- Content coding: Compression & decompression
- Transfer coding: Chunked transfer
- Cache-Control header
- Basic authentication, digest authentication, secure HTTP

- HTTP Working Group published HTTP/1.1 (RFC 2068, RFC 2616)

GET /multimedia HTTP/1.1

Host: www.myservice.com

User-Agent: Mozilla/5.0 (Linux; Android 12)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/100.0.4896.127 Mobile Safari/537.36

Accept: image/png, image/jpg; q=0.9, image/*; q=0.8

Accept-Language: en-us

Accept-Encoding: compress, gzip

Accept-Charset: iso-8859, utf-8; q=0.8

Transfer-Encoding: chunked

Cache-Control: max-age=120

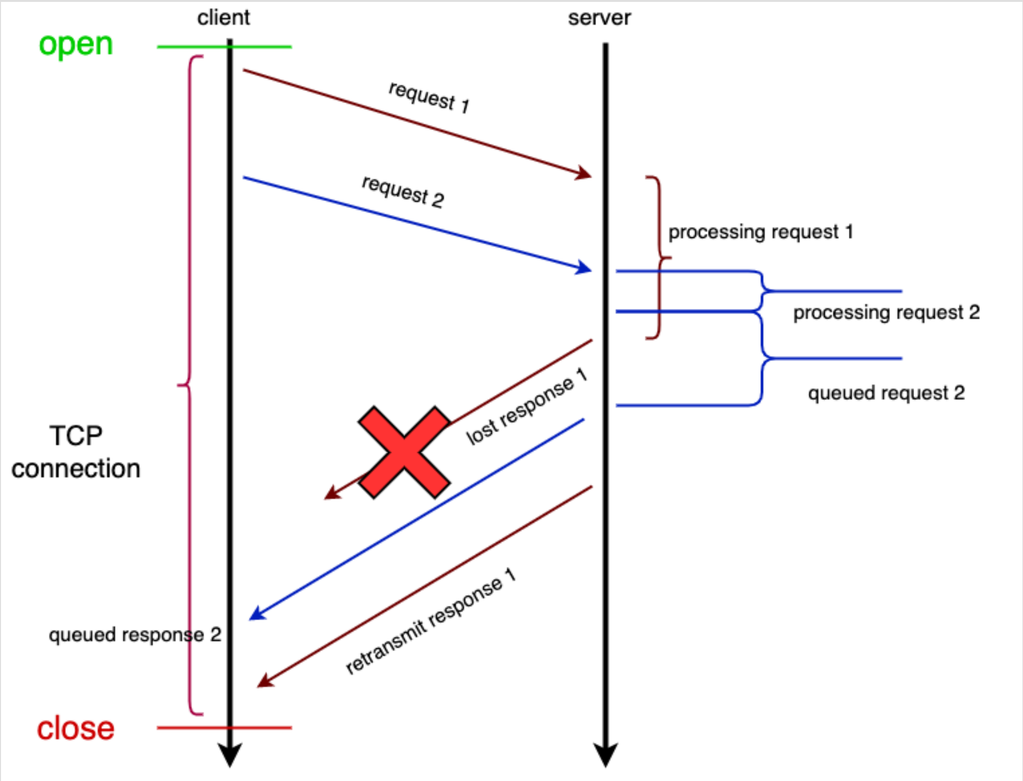
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==

© Stefan Jaindl

HTTP Evolution

HTTP/1.1 SHORTCOMINGS

- Pipelining
- Head-of-line blocking
- Fixed number of concurrent connections
- Repetitive headers & cookies
- TCP slow start
- Workarounds
  - Domain sharding
  - Cookieless domains
  - Image spriting



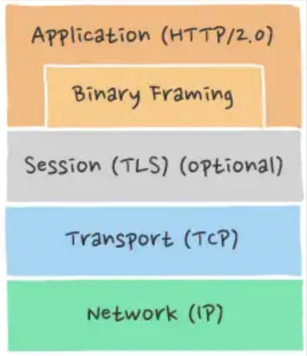
© Stefan Jaindl

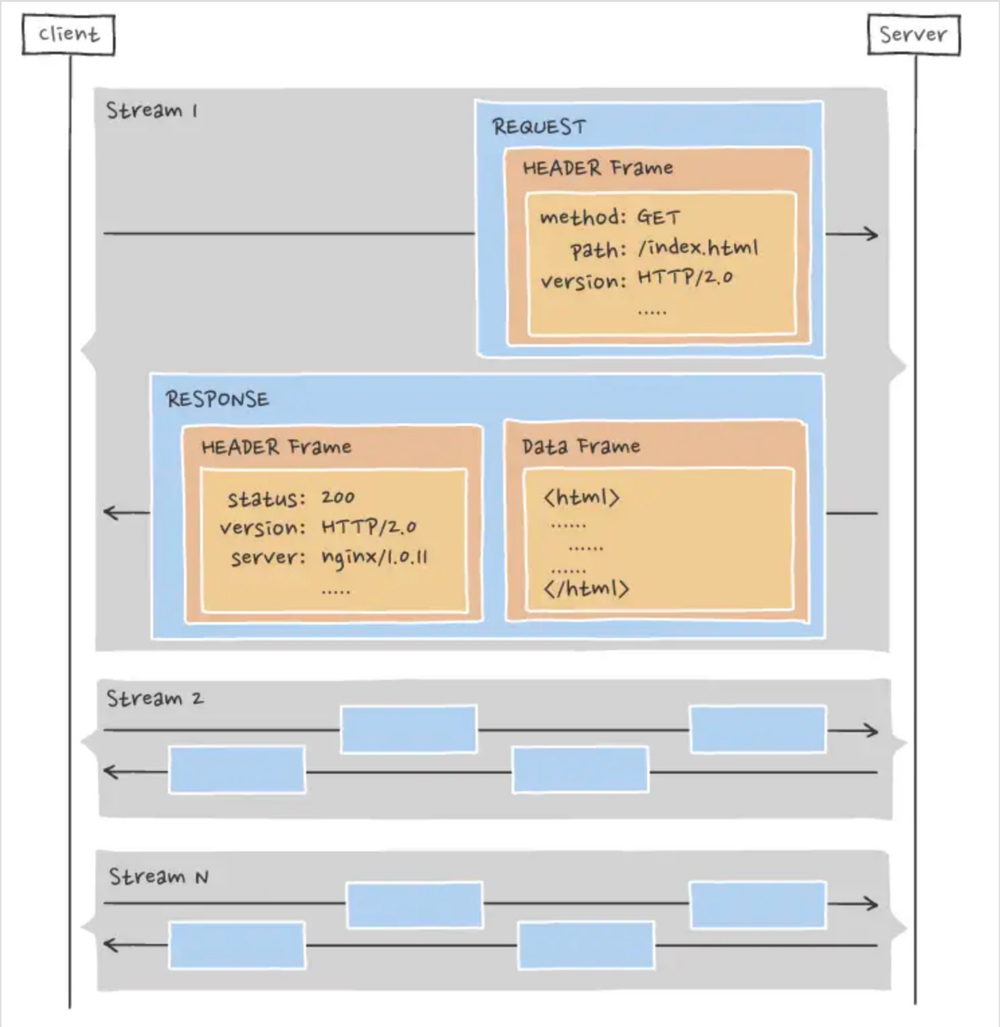
HTTP Evolution

HTTP/2.0

- Major overhaul
  - SPDY (Google)
  - Standard since 2015

- Binary framing layer
  - Frames, messages, streams





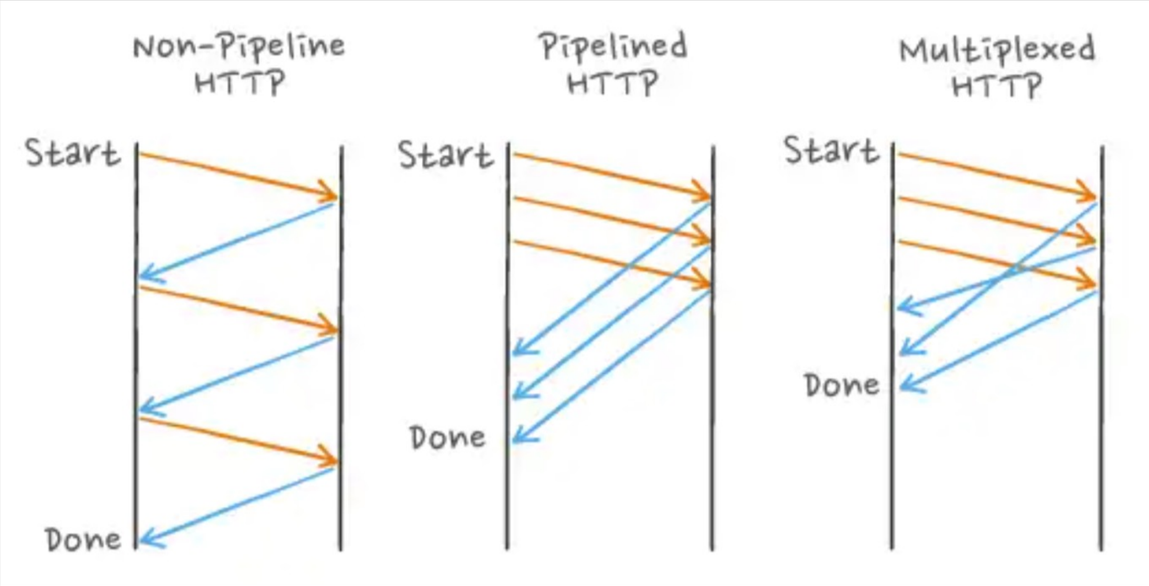
© Stefan Jaindl

HTTP Evolution

HTTP/2.0

- Multiplexing with streams
  - Not ordered & blocking anymore
  - Solving application layer (7) HoL blocking
  - Transport layer (4) HOL still an issue
  - Load balancer

- Single TCP connection

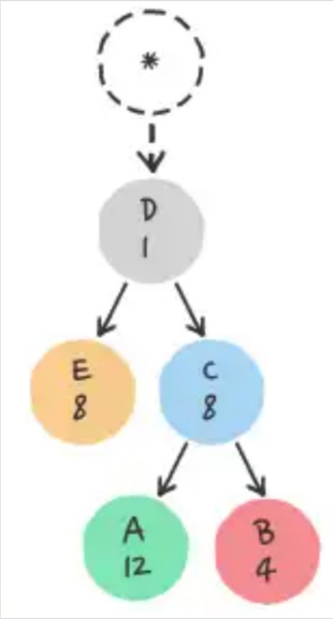


© Stefan Jaindl

HTTP Evolution

HTTP/2.0

- Flow Control
  - Default: 65,535 octes (bytes) buffer
  - WINDOW_UPDATE frames
- Stream prioritization
  - Weighted prioritization tree
- Header compression
  - HPACK
- Server push
  - Push data without requests
  - Cache Digests extension

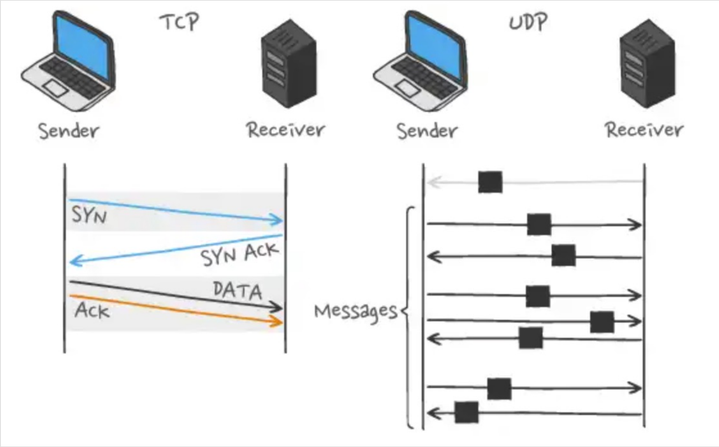


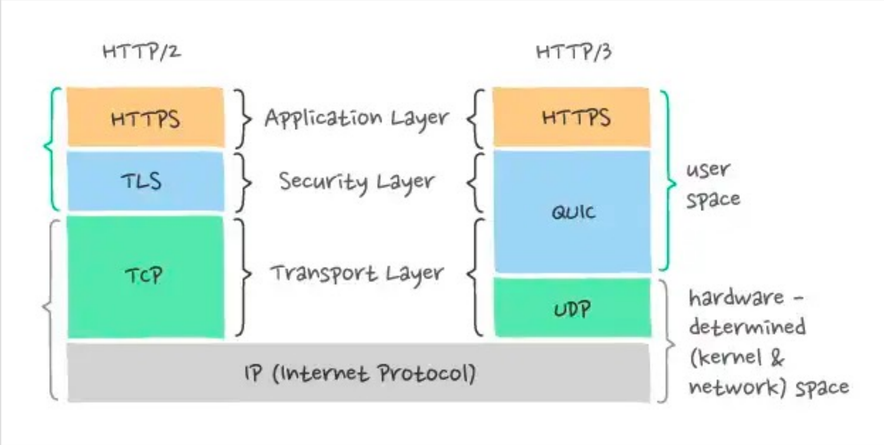
© Stefan Jaindl

HTTP Evolution

HTTP/3.0

- Increasing demand of real-time applications
- Increasing traffic from mobile devices
- New transport protocol: QUIC
  - gQUIC (Google)
  - Runs over UDP instead of TCP
  - Connectionless
- TCP problems
  - No support for stream-level multiplexing
  - Message overhead with known hosts
- Solves transport layer (4) HoL blocking



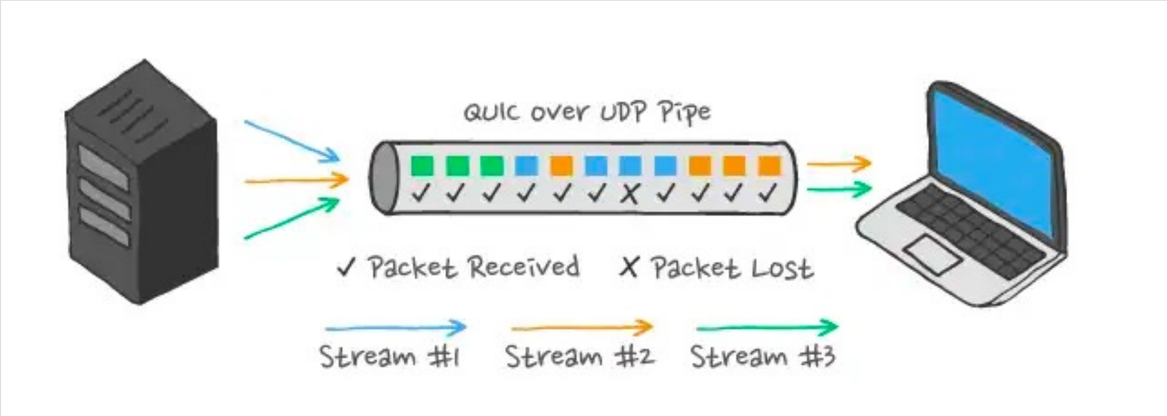


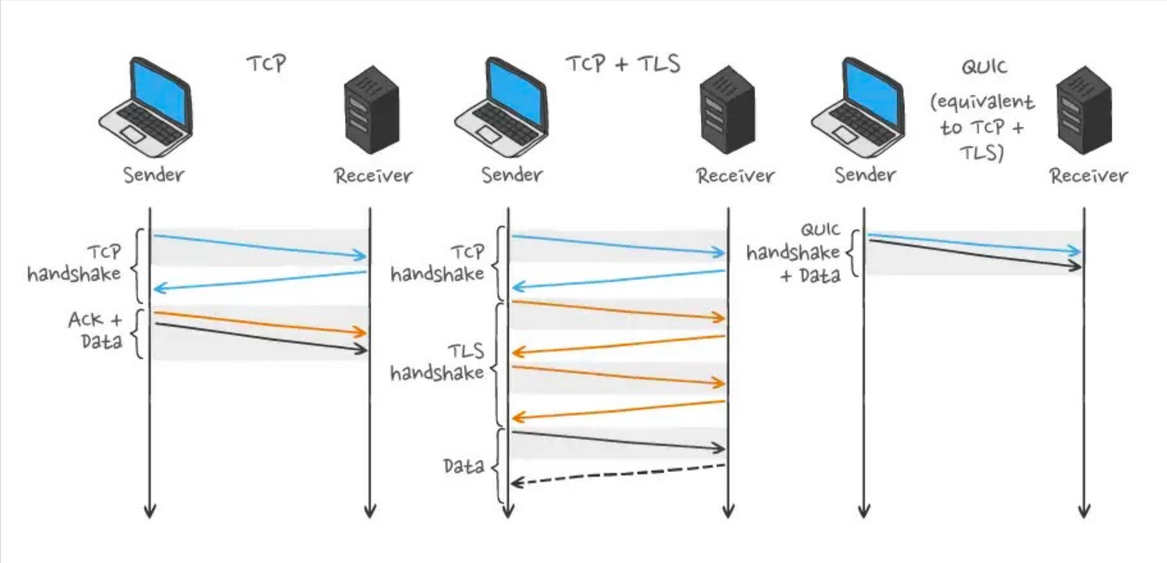
© Stefan Jaindl

HTTP Evolution

HTTP/3.0, QUIC

- Stream-level multiplexing over single connection
- Flow control per stream
- Flexible congestion control
- Improved error handling
  - Loss-recovery mechanism
  - Forward error-correction
- Faster handshaking with known hosts over TLS
- Header compression
  - QPACK
  - Headers can arrive out of order
- Improved server push





© Stefan Jaindl

Client-Server Communication

CHOOSING THE RIGHT PROTOCOL / METHOD

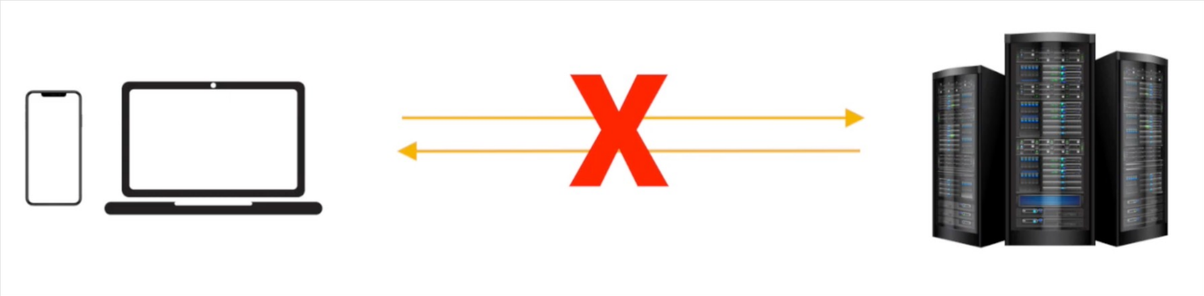
- HTTP Server Push
- HTTP Short Polling
- HTTP Long Polling
- HTTP Streaming
- Web Sockets
- Server-Sent Events
- REST
- gRPC
- GraphQL
- Push Notifications - GCM

© Stefan Jaindl

Client-Server Communication

HTTP SHORT POLLING

- Simple HTTP requests & responses
  - No persistent connection
  - Poll at regular interval (e.g. every x seconds)
- Notification delay
- Overhead
  - TLS handshakes & HTTP headers
  - Responses could be empty
  - Unnecessary network calls & battery drain
- Easy to implement
- Use case
  - No need for real-time data
  - Poll interval can be long



© Stefan Jaindl

Client-Server Communication

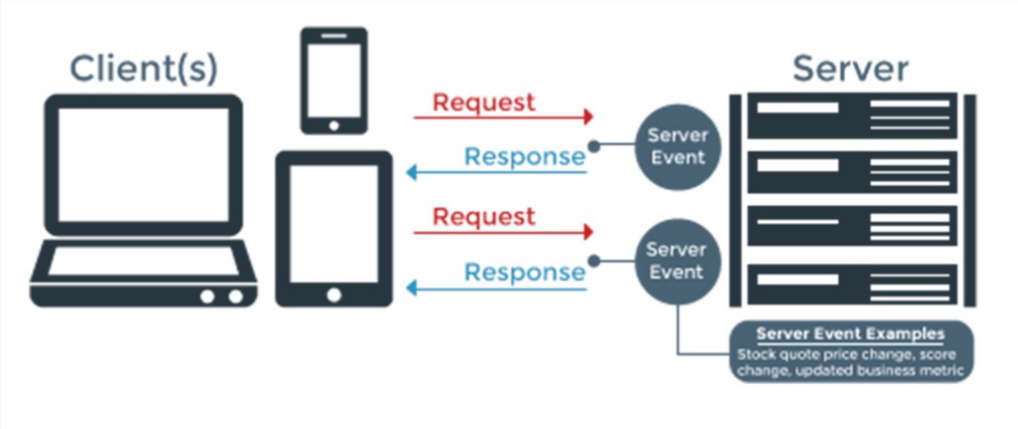
HTTP LONG POLLING

- Single HTTP request
  - Persistent connection
  - Server responds when new data is available
  - Timeouts
  - Connection closed and reopened after timeout or response

- Pros
  - Instant notifications, better latency
  - No empty responses
  - Less TCP/TLS overhead

- Cons
  - Difficult to manage multiple connections
  - More server-side resources needed
  - Performance & scaling issues (batch responses)

- Use case
  - Real-time data needed
  - Server updates not too frequently



© Stefan Jaindl

Client-Server Communication

HTTP STREAMING

- Persistent connection
  - Kept open after server pushes
  - Reduced latency
- Transfer large files or text
- Timeouts
- Content length
  - Content-Length header, HTTP/1.1
  - Transfer-Encoding header (chunked), HTTP/1.1
  - End of File (EOF), HTTP/1.0
- Issues
  - Network intermediaries caching (use Cache-Control)
  - Latency issues (long poll requests, TCP retransmissions)
  - Application layer needs to handle rechunking
  - Security: DoS attacks, cross-site scripting

HTTP/1.1 200 OK
Content-Type: text/plain
Transfer-Encoding: chunked

13
Data of first chunk

1C
and this is the second one

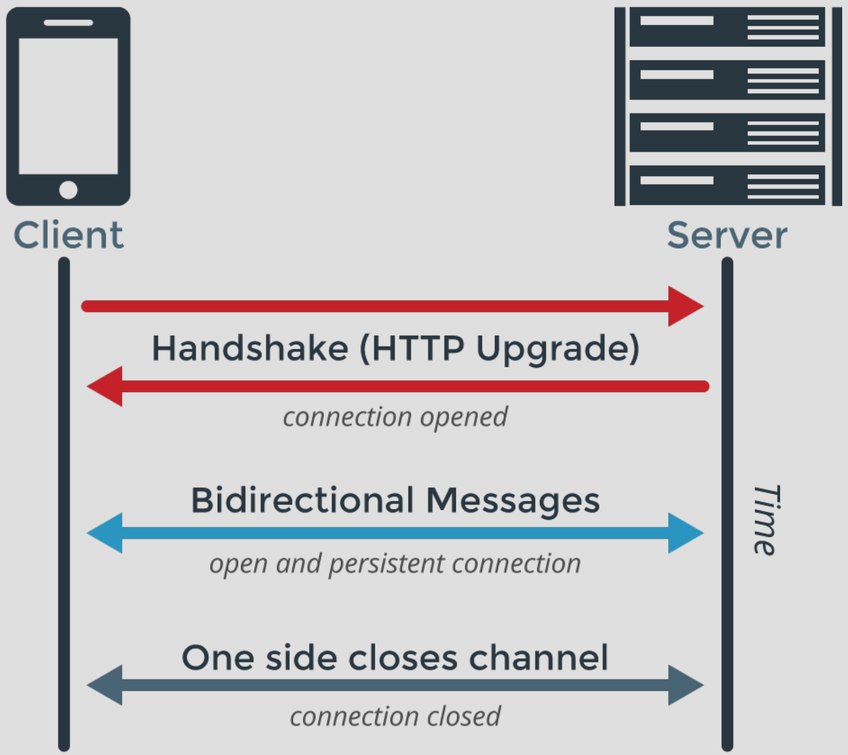
0

© Stefan Jaindl

Client-Server Communication

WEB SOCKETS

- Designed for real-time applications
- Bidirectional client-server communication (layer 7)
- Over persistent single TCP connection (layer 4)
- Leverage existing HTTP technology
  - Supports HTTP intermediaries
  - Uses ports 80 (HTTP) & 443 (HTTPS)
- Secure web sockets (wss)
- Transmit binary, text & control frames
- Client & server can close connection



© Stefan Jaindl

Client-Server Communication

WEB SOCKETS

- Handshake with HTTP Upgrade header
  - Client: random Sec-WebSocket-Key
  - Server: Sec-WebSocket-Accept
    (Base64 encoded SHA-1 hash of key + GUID)
- Message frames
  - Structured message header
  - Payload
    Subprotocol: Message agreement (JSON, XML, ...)
    Sec-WebSocket-Protocol header
- Authentication: Standard HTTP Auth
- Extensions: gzip compression, IoT message transfer
- No automatic connection recovery

Client handshake:

GET /index.html HTTP/1.1
Host: www.example.com
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: BOq0IliaPZlnbMHEBYtdjmKIL38=
Sec-WebSocket-Protocol: xml, json

Server handshake:

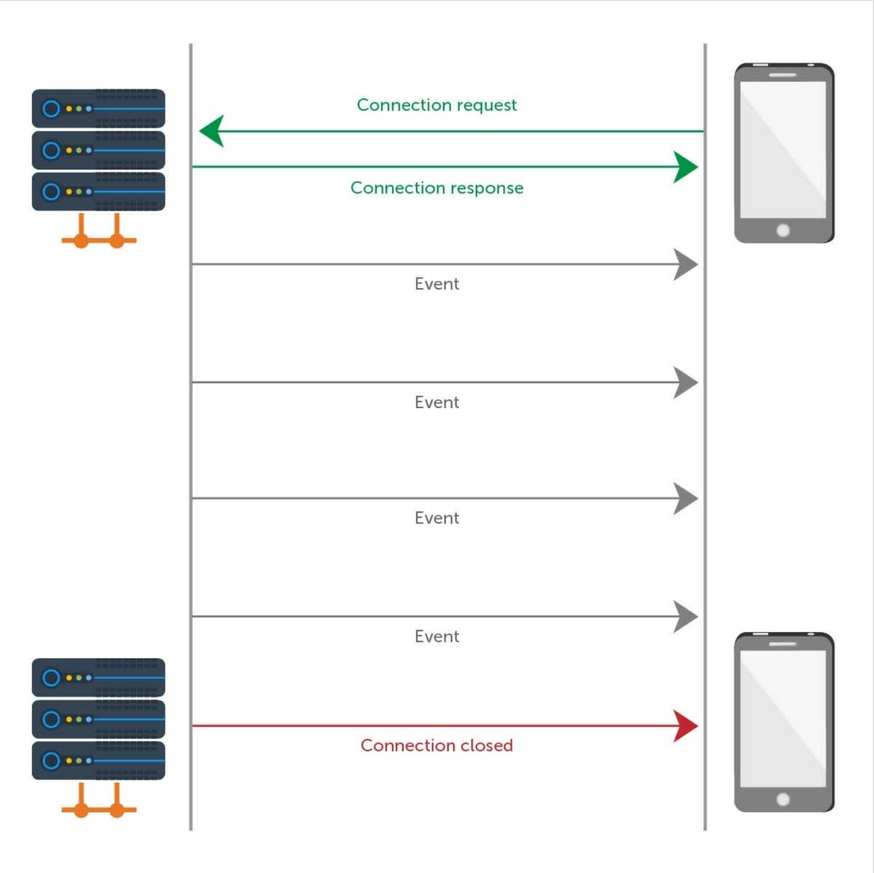
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Accept: m9raz0Lr21hfqAitCxWigVwhppA=
Sec-WebSocket-Protocol: json

© Stefan Jaindl

Client-Server Communication

SERVER-SENT EVENTS

- Server push technology to receive automatic updates from a server
- Persistent unidirectional connection
  - No heartbeats
- Client-initiated communication to an endpoint
  - May include query parameters
- Connection closed if all events sent, connection is stale or explicitly closed by the client
- Reduced latency & less server load as with HTTP streaming
  - No connection management & parsing logic overhead
- Scaling SSE
  - Load balancers with shared/synched resources
  - Message broker: pub/sub mechanism



© Stefan Jaindl

Client-Server Communication

SERVER-SENT EVENTS

- Based on HTTP protocol
  - HTTP header „**Accept:** text/event-stream“

- Data Format
  - UTF-8
  - Binary not supported
  - “event” blocks: “put”, “keep-alive”, ..
  - “data” blocks: e.g. JSON

event: put

data: {data”:{“items”:[“BTC”, “ETH”, DOGE”],”prices”:[36837.95, 2694.25, 0.124639]}}

event: keep-alive

data: null

© Stefan Jaindl

Client-Server Communication

REST - REPRESENTATIONAL STATE TRANSFER

- Uniform Interface
  - Identification of resources (URI's)
    - Document, collection, objects (e.g. persons), ...
  - Resource representation
    - Media Type: HTML, XML, JSON, ..
    - Data
    - Metadata (last-modified, media type, ...)
    - Hypermedia links
  - Self-descriptive resources
    - Client should act only on resource media type
  - Hypermedia
    - Hyperlinks (transitioning between states)
- Resource Methods
  - GET, POST, PUT, DELETE (Uniform interface)

Request:

GET https://www.example.com/v1/platform?type=mobile

Response:

```
{
  "result": [
    {
      "id": "and",
      "name": "Android"
    },
    {
      "id": "ios",
      "name": "iOS"
    }
  ]
}
```

© Stefan Jaindl

Client-Server Communication

REST - PRINCIPLES

- Separation of Concerns principle (client – server)
  - Improve client portability
  - Improve server scalability

- Stateless
  - Session state kept on client (also no session cookies)
  - Improves visibility, reliability & scalability
  - Each request contains all information (network overhead)

- Cache
  - client-cache-stateless-server style
  - Server responses marked as cacheable or non-cacheable
  - Pro: Improved latency and network usage
  - Con: Stale data (time-to-live)

© Stefan Jaindl

Client-Server Communication

REST - PRINCIPLES

- Layered System
  - Hierarchical & independent layers
  - Components only aware of immediate layers
  - Shared intermediaries (caching, load balancing)
  - Pro: Reduced system complexity & improved scalability
  - Con: Higher Latency

- Code-On-Demand (optional)
  - Allows download of executable code/scripts
  - Pro: Improves system extensibility
  - Con: Reduces visibility

© Stefan Jaindl

Client-Server Communication

REST - SUMMARY

- Pro
  - Established standard
  - Easy to learn & understand
  - Caching using built-in HTTP mechanism
  - Loose coupling between client & server

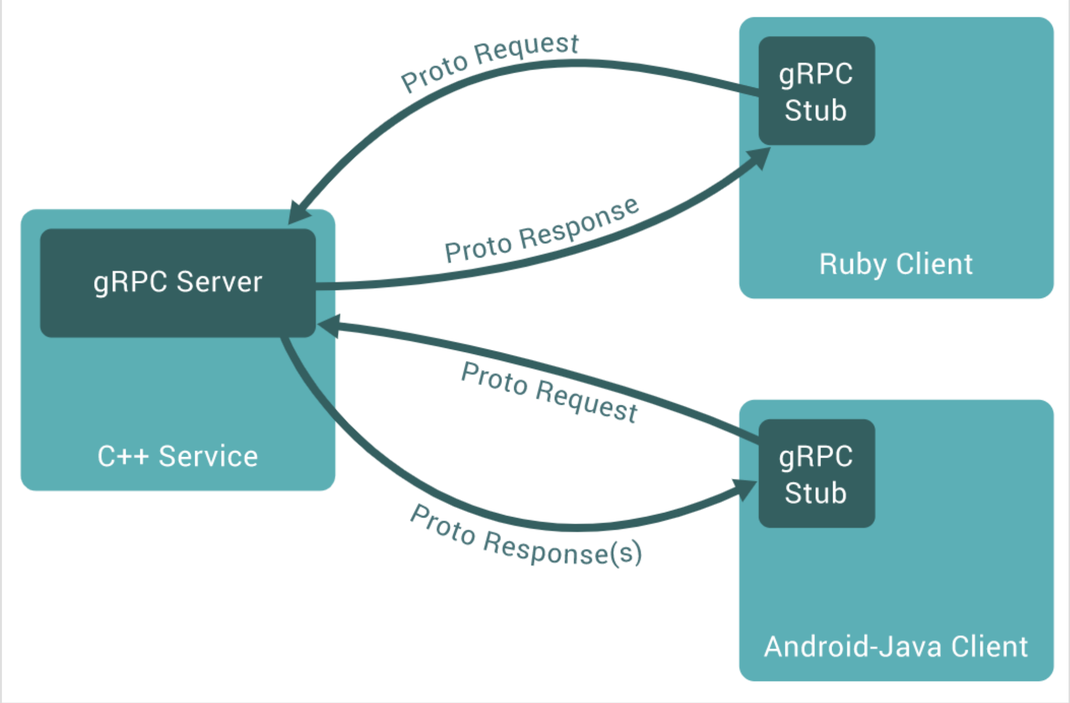
- Con
  - Each request requires a separate connection
  - Each request contains metadata & headers
  - Schemaless – hard to check data validity on client

© Stefan Jaindl

Client-Server Communication

GRPC

- Remote Procedure Call framework
- Protobuf
  - IDL: Service definition
  - Message interchange format (binary)
- Code generation
  - Client & server stubs
  - Many supported programming languages
- Functioning
  - Client performs local function call (method, metadata, opt. deadline)
  - Proto request & response
  - Automatic marshalling & unmarshalling



© Stefan Jaindl

Client-Server Communication

GRPC

- gRPC channel
  - Connection via host & port
- Messages
  - Unary
  - Server stream
  - Client stream
  - Bidirectional stream
- Synchronous & Asynchronuous implementations
- HTTP2-based transport
- Pluggable authentication

```
message HelloRequest { string greeting = 1; }

message HelloResponse { string reply = 1; }

service HelloService {
  rpc SayHello (HelloRequest) returns (HelloResponse);

  rpc ReplyStream(HelloRequest) returns (stream
    HelloResponse);

  rpc GreetStream(stream HelloRequest) returns
    (HelloResponse);

  rpc BidirectStream(stream HelloRequest) returns (stream
    HelloResponse);
}
```

© Stefan Jaindl

Client-Server Communication

REST VS. GRPC

- HTTP/1.1 or HTTP/2.0
  - Typically only unary request/response mechanism
  - HTTP/1.1: Higher battery drain & latency
- Need for HTTP metadata & method
- Rich support across platforms
  - Most often used API

- HTTP/2.0
  - Reduced battery use & latency, HTTP push, bidirectional streaming
- No HTTP metadata & methods
- Rich support still lacking
  - Often used for internal systems communication

© Stefan Jaindl

Client-Server Communication

REST VS. GRPC

- Message Format
  - JSON/XML (no fixed schema)
  - Textual, human-readable
  - Increased message size & network usage
  - Serialization/deserialization to custom types slower
- Code generation
  - 3rd party tools (Swagger, Postman)
- Standardization & universal support

- Message Format
  - Protobuf (schema-based)
  - Binary, non-human-readable
  - Compressed, smaller size, less network usage
  - Serialization/deserialization fast
- Code generation
  - Native for many platforms & languages
- Optimized for performance

© Stefan Jaindl

Client-Server Communication

GRAPHQL

- API query language
  - Server-side runtime for query execution
- Not tied to any specific storage engine
- Schema
  - Strongly typed type system to define data
    - Support for enums, custom types, arguments, interfaces, unions
  - Queries, mutations & fragments
- Code generation
- Possible to query only needed fields
  - Better performance
  - No versioning needed when adding fields
- Possible to query related objects
  - No roundtrips needed as in REST

Type definition:

```
type Hero {
  id: ID!
  name: String!
  episode(num: Int): String
  friends: [Hero]
}
```

Query:

```
query Heroes($episode: Episode)
{
  hero {
    name
    episode($episode)
    friends {
      name
    }
  }
}
```

Query response:

```
{
  "data": {
    "hero": {
      "name": ”Chewbacca",
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        }
      ]
    }
  }
}
```

© Stefan Jaindl

Client-Server Communication

GRAPHQL

- Validation
  - compile-time syntax checks
- Execution
  - Server resolvers until scalar values reached
- Introspection
  - Retrieve defined types, fields, data types
  - Documentation

© Stefan Jaindl

Client-Server Communication

GRAPHQL – BEST PRACTICES

- HTTP
  - Single endpoint
  - GET (queries)
  - POST (mutations)
    - "application/graphql" Content-Type header
- Representation typically JSON
  - HTTP gzip compression header
- Pagination
  - Edge-based (plurals, slices)
  - End-of-list, counts, connections
  - Opaque cursors

Paginated query:

```
{
  hero {
    name
    friendsConnection(first: 1
          after: "Y3Vyc29yMQ==") {
      totalCount
      edges {
        node {
          name
        }
        cursor
      }
      pageInfo {
        endCursor
        hasNextPage
      }
    }
  }
}
```

Paginated response:

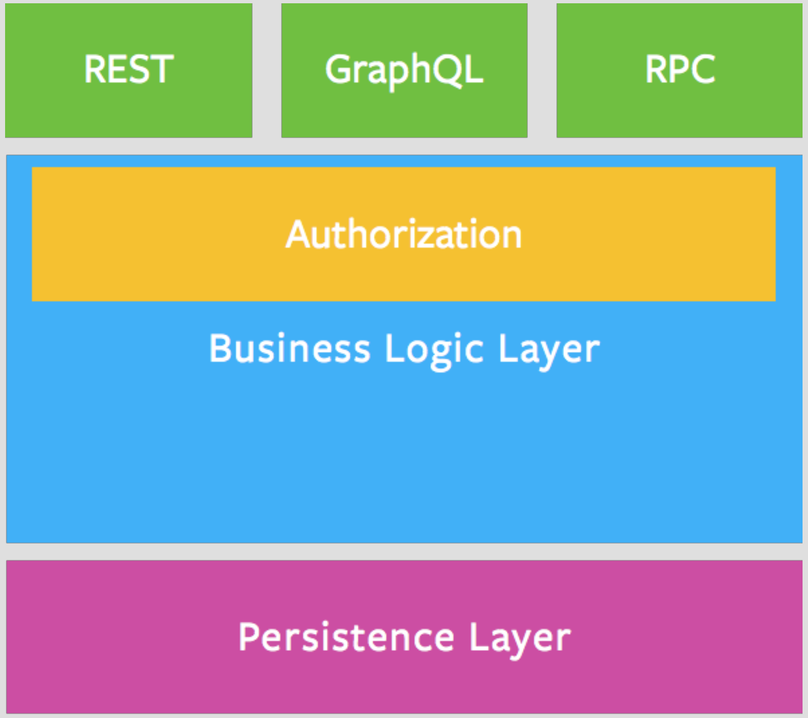
```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friendsConnection": {
        "totalCount": 3,
        "edges": [
          {
            "node": {
              "name": "Han Solo"
            },
            "cursor": "Y3Vyc29yMg=="
          }],
        "pageInfo": {
          "endCursor": "Y3Vyc29yMw==",
          "hasNextPage": false
        }
      }
    }
  }
}
```

© Stefan Jaindl

Client-Server Communication

GRAPHQL – BEST PRACTICES

- Versioning possible, but not best practice
  - Avoid breaking changes & frequent releases
  - Add new types & fields without new version
- Nullable fields by default
  - Network failures, database errors, exceptions, etc.
- Server-side batching
  - Batch multiple database requests of resolvers
- Caching
  - Unique ids for caching (e.g. Node interface with ID)
- Authorization should be placed in business logic layer



© Stefan Jaindl

Client-Server Communication

REST VS. GRAPHQL

- Multiple endpoints
  - One per resource
  - Many requests needed to navigate through resources
  - Standard HTTP caching: path + query params
- Versioning needed when adding fields
- Not strongly typed, no fixed schema
- 3rd party code generation (Postman, Swagger)
- Security
  - DoS attack on 1 endpoint causes less server load

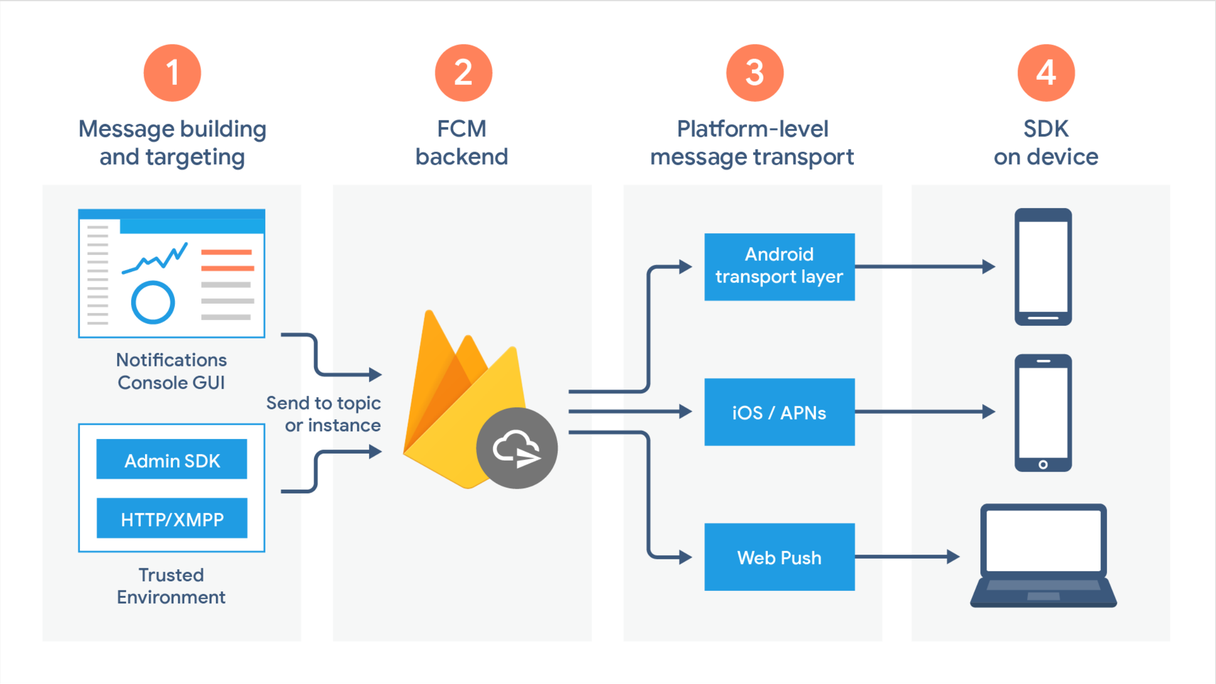
- One endpoint
  - Graph-based queries
  - One query can deliver all needed information. Saves network bandwidth, better latency
  - Caching with global unique ids
- No versioning needed when adding fields
- Strongly typed, schema
- Automatic code generation based on schema
- Security
  - DoS attacks may cause higher server load

© Stefan Jaindl

Client-Server Communication

PUSH NOTIFICATIONS – FIREBASE CLOUD MESSAGING

- Cross-platform messaging solution
  - Notify registered clients when new data is available
- Architecture
  - Trusted environment
    - Cloud Functions, App Engine or own app server
    - Admin SDK & FCM server protocols support
    - Build message requests
  - FCM backend
    - Accept & distribute/fanout messages
    - Generate message metadata (e.g. message ID)
  - Platform transport service
    - Message delivery on Android transport layer
    - Requires Google Play Services
  - iOS, Android or web app
    - FCM SDK for message handling & displaying



© Stefan Jaindl

Client-Server Communication

FCM – MESSAGE FORMATS

- Notification message (displayed by FCM SDK)
  - Predefined keys
  - In foreground: App receives message
  - In background: Notification tray
- Data message (handled by app)
  - Custom keys
  - Payload up to 4000 bytes per message
- Combined message
- Device instance tokens
- Platformspecific fields
  - AndroidConfig: ttl, click_action, etc.

Combined message:

```
{
  "message": {
    "token” : "bk3RNwTe3H0:CI2k_HHwgIpoDKCIZ...",
    "notification” : {
      "title” : ” Italy vs. Austria",
      "body” : ”Great match!"
    },
    "data" : {
      ”referee" : "Mario",
      ”stadium" : ”Giuseppe Meazza Stadium"
    }
  }
}
```

© Stefan Jaindl

Client-Server Communication

FCM – MESSAGES

- Upstream messages to server
  - Acknowledgements, chats, etc.
  - XMPP protocol

- Device targeting
  - Single device
  - Device group
  - Topics (fanout)

- Collapsible & non-collapsible messages

© Stefan Jaindl

Client-Server Communication

FCM – MESSAGES

- Message priority
  - Normal
  - High (delivery even in Doze mode)
- Time to live: Max 28 days
- Automatic message throttling by FCM backend
- Keep track on device token states (store on server)
  - Save network bandwith of stale tokens
  - Update token valditity

© Stefan Jaindl

Client-Server Communication

PAGINATION

- Fetch lists of data in pieces as needed
  - Reduce memory & network usage
- Types
  - Offset/Page-based pagination
  - Keyset pagination
  - Cursor/Seek pagination
- Jetpack Paging library
  - Repository, ViewModel + UI layer

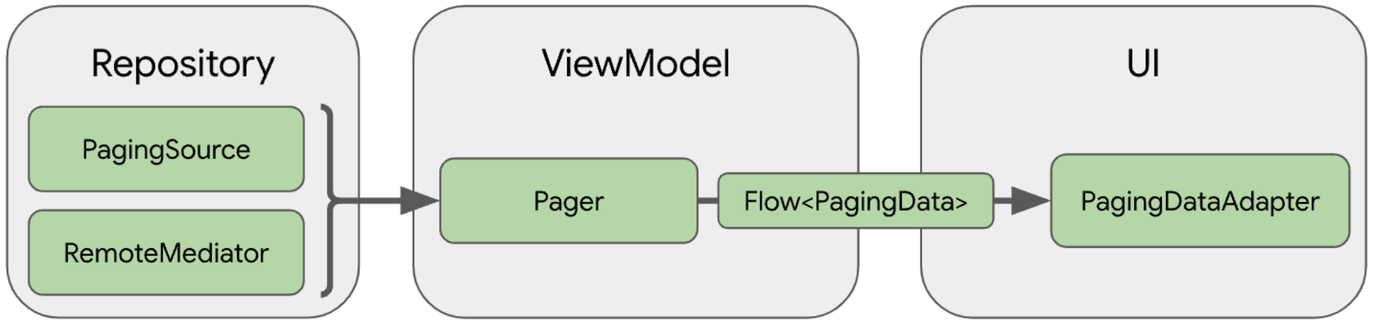


Figure 1. An example of how the Paging library fits into your app architecture.

© Stefan Jaindl

Pagination

OFFSET/PAGE-BASED PAGINATION

- Limit
  - Number of elements to fetch
- Offset
  - Number of elements to skip from start
  - Alternatively: Page number
- Built-in SQL commands
- Pro
  - Easiest pagination type
  - Stateless
  - Works well together with sorting
- Con
  - Bad performance with high offset values
    O(L+O) time complexity (L = Limit, O = Offset)
  - Page drift

GET /items?limit=20

GET / items?offset=20&limit=20

SQL:

SELECT * FROM items WHERE name > '' ORDER BY name LIMIT 20 OFFSET 20

© Stefan Jaindl

Pagination

KEYSET PAGINATION

- Limit
  - Number of elements to fetch
- After
  - Value of first element to return
- Pro
  - Simple SQL query
  - No additional backend logic needed
  - Stateless
  - Good performance with large datasets
    O(L) time complexity (L = Limit)
- Con
  - Requires types with natural ordering

GET /items?limit=20

GET /items?after=01-05-2022&limit=20

SQL:

SELECT * FROM items WHERE date >= ‘01-05-2022'
ORDER BY date LIMIT 20

© Stefan Jaindl

Pagination

CURSOR/SEEK PAGINATION

- Unique cursors
- Pro
  - Good performance with large datasets
    O(L) time complexity (L = Limit)
  - Back and forward navigation
  - Null cursors
  - Cursors can be Base64-encoded & encrypted on server-side (safeguard against manipulation)
  - Stateless
  - No duplicated and skipped data

GET /items?limit=20

```
{
  “cursor”: {
    “previous_page”: null,
    “next_page”: "next__coke"
  }
}
```

GET /items?after_id=coke&limit=20

```
{
  “cursor”: {
    “previous_page”: "prev__cola" ,
    “next_page”: "next__fanta"
  }
}
```

First page SQL:

SELECT * FROM items WHERE name > '' ORDER BY name LIMIT 5

SQL for next_page cursor:

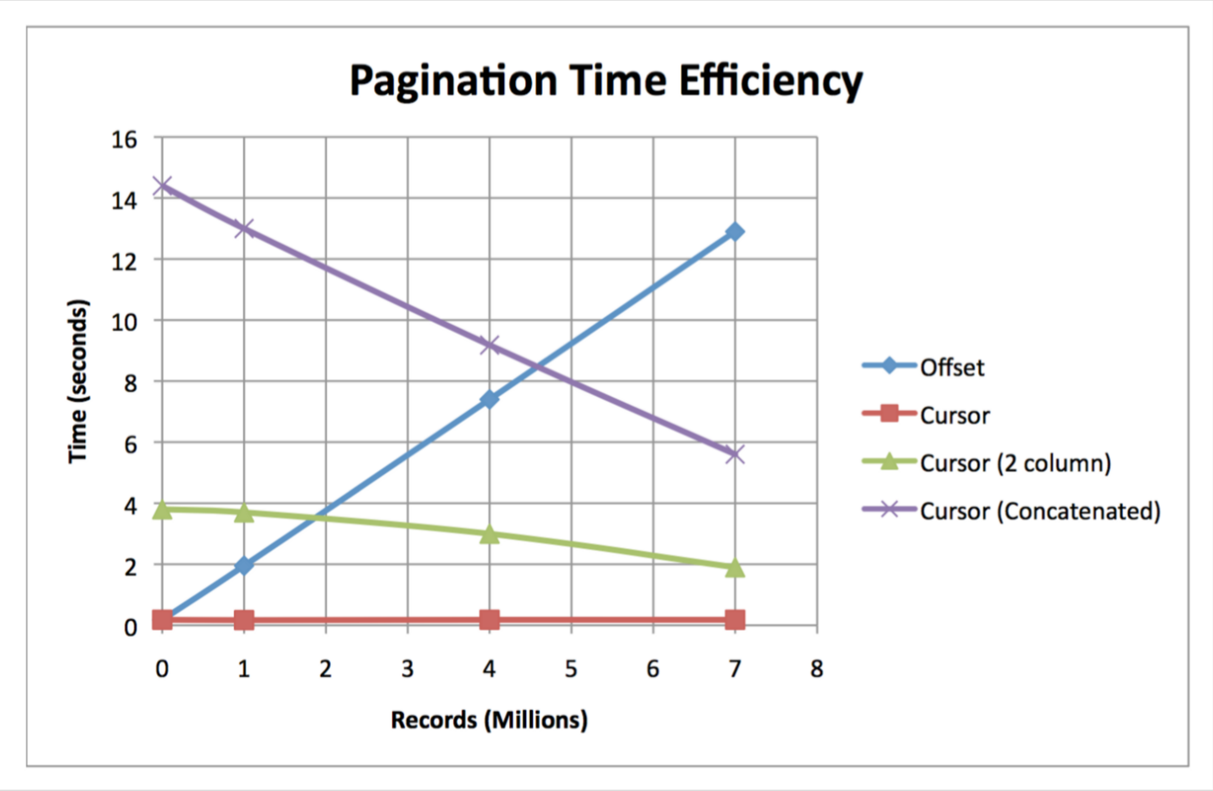
SELECT * FROM items WHERE name > ‘coke' ORDER BY name LIMIT 5

© Stefan Jaindl

Pagination

CURSOR/SEEK PAGINATION - CON

- More complicated to implement in backend
- Cursor IDs get invalid, if deleted
- No way to skip pages
- Requires a unique sequentially ordered column
  - Data sorted by this column
  - SQL concat or select on multiple columns slow
  - Create concatenated column, if needed (e.g. firstName + lastName)



© Stefan Jaindl

Further Topics

BACKGROUND WORK

- Services
  - Background Service: Running as long as app is running
  - Foreground Service: Running after app termination
  - Bound Service: Bound to specific app component

- Background restrictions

- JobScheduler & WorkManager

© Stefan Jaindl

Further Topics

NETWORK FAILURE HANDLING

- Logging
- Retry strategies
  - Linear
  - Exponential backoff
  - Exponential backoff with jitter
- API rate limiting (HTTP code 429 – Too many requests)
- Informing the user

© Stefan Jaindl

Further Topics

NETWORKING

- API Design (“Mock Interviews” chapter)
- Authentication (“Security & Privacy" chapter – Advanced topics course)
- XMPP
  - Bidirectional transport protocol
  - More built-in security but slower than web sockets
- Web Transport
  - Alternative to web sockets (HTTP/2.0 & HTTP/3.0, QUIC), draft
- Quality of Service (“Build for Billions” chapter)

© Stefan Jaindl

Storage

STORAGE SYSTEMS

© Stefan Jaindl

Storage Types

OVERVIEW

- File system
- Shared Preferences
- DataStore
  - Preferences DataStore
  - Proto DataStore
- Database
  - SQL
  - NoSQL
- Secure keystore

© Stefan Jaindl

Storage Types

FILE SYSTEM STORAGE

- Access scope
  - App-specific storage (app use only)
  - Shared storage (media, documents)

- Storage location
  - Internal storage (phone hard drive)
  - External storage (e.g. SD card)

- Scoped storage
  - Default permission model since API 29
  - Default access to app-specific directories in internal + external storage & media store
    - No READ_EXTERNAL_STORAGE & WRITE_EXTERNAL_STORAGE needed
  - MANAGE_EXTERNAL_STORAGE
    - Permission to manage all files except system & app-specific files (e.g. file explorer apps)

© Stefan Jaindl

File System Storage

APP-SPECIFIC STORAGE

- Private data store meaningful to app only
  - Special subfolder for app-only media
- Files & cache directories
- Files removed on app uninstall
- Internal storage
  - Use for sensitive data (encrypted >= API 29)
  - No permissions needed
  - Limited storage space
- External storage
  - No permissions needed (> API 19)
  - Less reliable (storage can be unmounted)

Internal storage:

context.getFilesDir()
context.getCacheDir()

External storage:

context.getExternalFilesDir()
context.getExternalCacheDir()

© Stefan Jaindl

File System Storage

SHARED STORAGE

- Data store for files to be shared with other apps
  - Media, documents, other files

- Media
  - Image, audio, video
  - MediaStore API
  - Read-permission needed for accessing other app‘s media (>= API 30)
  - Files not deleted on app uninstall

- Documents & other files
  - Storage Access Framework
  - No permissions needed
  - Can be accessed via system file picker
  - Files not deleted on app uninstall

- Datasets
  - Machine learning, media playback
  - BlobStoreManager API

© Stefan Jaindl

Storage Types

SHARED PREFERENCES

- Simple file-based key-value storage
- Useful for storing settings and simple unstructured data
- Private and shared mode

Retrieve SharedPreferences:

```
val sharedPref = context.getSharedPreferences(
        getString(R.string.preference_file_key),
        Context.MODE_PRIVATE)
```

© Stefan Jaindl

Storage Types

DATASTORE

- SharedPreferences
  - Sync I/O operations on UI thread
  - Runtime exceptions, (parsing) errors
  - No type-safety
- DataStore
  - SharedPreferences replacement
  - Based on Flow & coroutines
  - Stores data async, consistently, and transactionally
- Preferences DataStore
  - Key-value pairs without schema & type safety
- Proto DataStore
  - Custom data types defined with protobuf schema

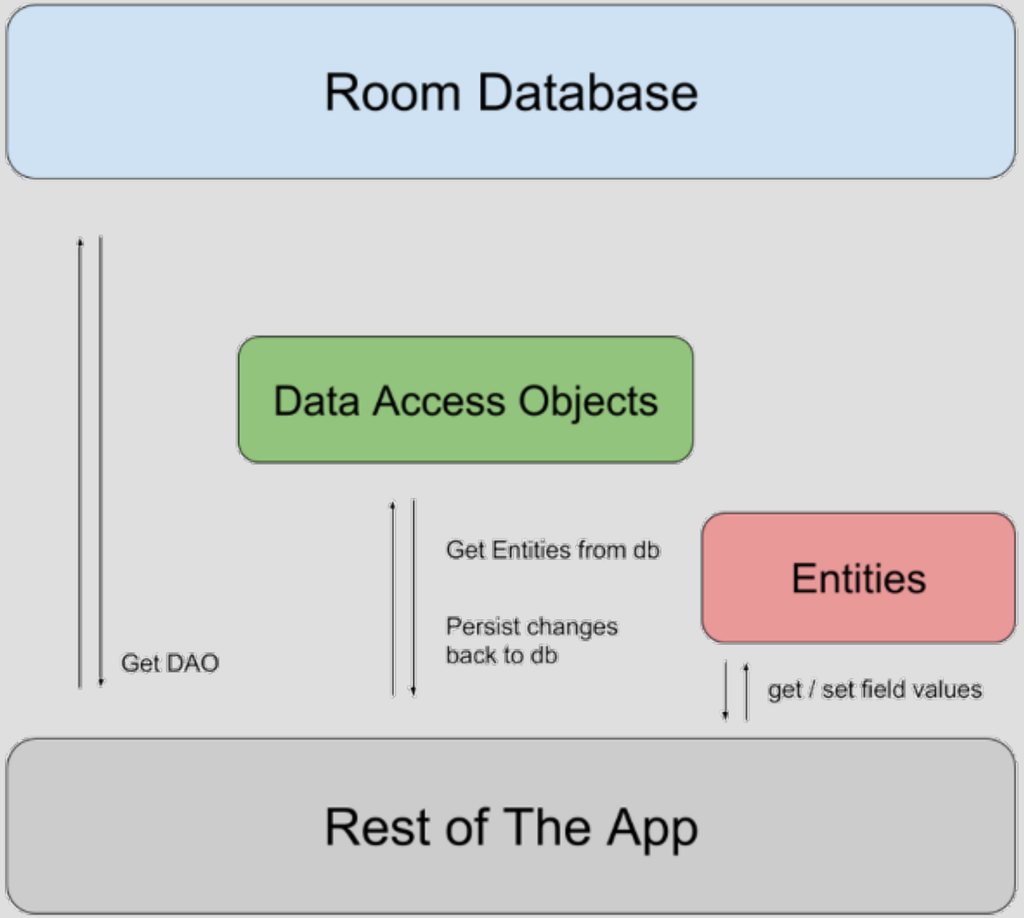
Feature	SharedPreferences	Preferences DataStore	Proto DataStore
Async API	✅ (only for reading changed values, via listener)	✅ (via `Flow`)	✅ (via `Flow`)
Synchronous API	✅ (but not safe to call on UI thread)	❌	❌
Safe to call on UI thread	❌	✅ (work is moved to `Dispatchers.IO` under the hood)	✅ (work is moved to `Dispatchers.IO` under the hood)
Can signal errors	❌	✅	✅
Safe from runtime exceptions	❌	✅	✅
Has a transactional API with strong consistency guarantees	❌	✅	✅
Handles data migration	❌	✅ (from SharedPreferences)	✅ (from SharedPreferences)
Type safety	❌	❌	✅ with Protocol Buffers

© Stefan Jaindl

Storage Types

DATABASE – SQL - ROOM

- SQL Database (part of Jetpack)
  - Structured data
  - Compile-time query verification
  - Annotations to reduce boilerplate
- Cache network data for offline use
- Room database
  - Holds database main access point to persisted data
- Entities
  - Tables in database
  - Full-text search support
- Data Access Objects
  - Provide methods to query, insert, delete & update
  - Code generation
  - One-shot & observable queries

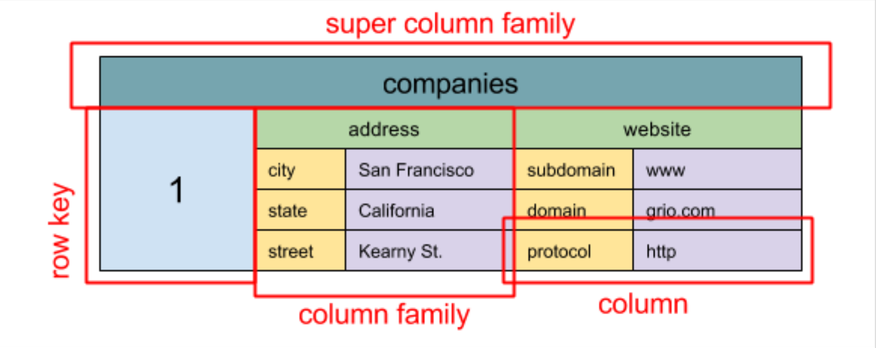


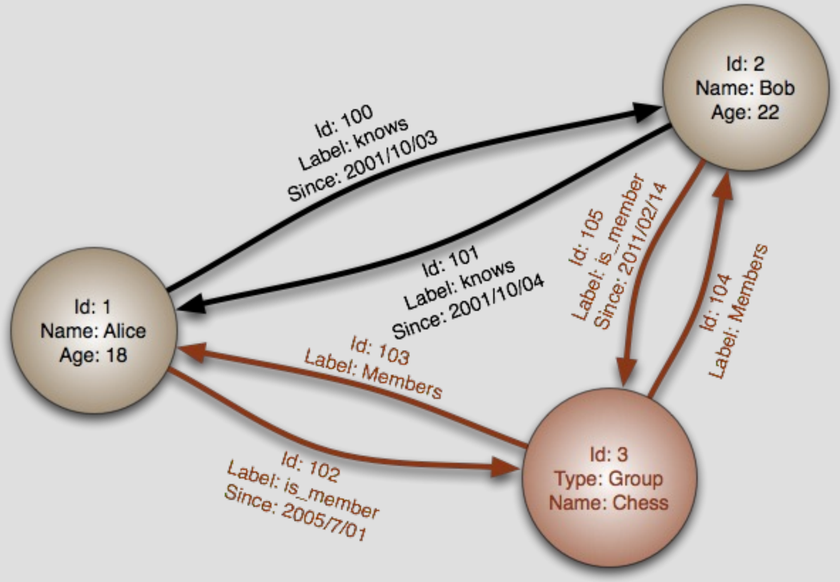
© Stefan Jaindl

Storage Types

DATABASE – NOSQL

- Key-Value store
  - Caching, base for more complex stores
- Document store
  - Flexible, infrequently changing data
  - Firebase Cloud Firestore, Realm
- Wide-column store
  - High availability & scalability
  - Google BigTable
- Graph store
  - Complex relationships, many foreign keys
- BASE
  - Basically available
  - Soft state
  - Eventual consistency





© Stefan Jaindl

Databases

SQL VS. NOSQL

- Structured data & strict schema
- Relational data
- Need for joins
- ACID
  - Favor consistency
  - Transactions
- Scaling patterns
  - SQL tuning, federation, sharding, master-slave replica, denormalization
- Lookups by index fast
- More established community, tools & documentation

- Semi-structured data & flexible schema
- Non-relational data
- No need for complex joins
- BASE
  - Favor availability

- Data-intensive workload
  - Store many TB of data
  - Very high I/O throughput

© Stefan Jaindl

Storage Types

LOBS

- Large Objects
- CLOB
  - Character Large Object
  - Store large textual data (character streams)
  - Text, PDF, Word documents , etc.
- BLOB
  - Binary Large Object
  - Store large binary data (binary streams)
  - Images, videos, audio files, etc.

© Stefan Jaindl

Storage Types

LOB VS. FILE SYSTEM

- LOBs
  - Database can get very large and inperformant
  - Query throttling issues
  - Used for transactional edits
- Filesystem
  - Store files in file system and path in database
  - DB Paths must be updated when files are moved
  - Better memory usage and performance than LOBs

- In most cases the filesystem is preferable

© Stefan Jaindl

Storage Types

ANDROID KEYSTORE SYSTEM

- Store cryptographic keys in container
  - Keys are non-exportable
  - Secure hardware store for supported key algorithm/block mode etc.
- Key use authorization
  - Cryptographical, validity, user authentication
- Keychain API (API 14)
  - System-wide credentials
- Android Keystore provider feature (API 18)
  - Single app credentials
- Jetpack Security lib
  - Symmetric & assymetric key generation, signatures

© Stefan Jaindl

Storage Types

WHEN TO USE WHICH STORAGE TYPE?

- Unstructured, non-sensitive data, settings, flags
  - Shared Preferences
  - Preferences DataStore

- Simple, non-sensitive, well-defined custom data types
  - Proto DataStore

- Large amounts of structured data, joins needed
  - SQL database

© Stefan Jaindl

Storage Types

WHEN TO USE WHICH STORAGE TYPE?

- Large amounts of semistructured data, performance/availability important
  - NoSQL database
- Secure storage needed (e.g. sensitive data, encryption keys)
  - Android Keystore
- File-based storage needs
  - Internal, external
  - Media & documents store
  - Documents, cache & temp directories

© Stefan Jaindl

Build for Billions

CONNECTIVITY, BATTERY, MEMORY & CO

© Stefan Jaindl

Build for Billions

CONCERNS

- Connectivity
  - Low-speed connections
  - No connection
- Data cost
  - Network & storage cost
  - App size, network usage
- Device ranges
  - Multiple Android versions
  - RAM configurations
  - Screen sizes & resolution

© Stefan Jaindl

Build for Billions

CONCERNS

- Battery consumption
- UI
  - Responsiveness
  - Localization
- App startup
- Memory

© Stefan Jaindl

Build for Billions

CONNECTIVITY - NETWORKING

- Caching data
  - Room, SharedPreferences
  - Use WorkManager for updating

- Deduplicate network requests
  - UI ←→ Database ←→ Network

© Stefan Jaindl

Build for Billions

CONNECTIVITY - NETWORKING

- Data prefetching

- Network fine-tuning
  - Prioritize network requests
  - Detect network changes (ConnectivityManager.NetworkCallback)
  - Adjust to network quality (ConnectivityManager/TelephonyManager)

© Stefan Jaindl

Build for Billions

CONNECTIVITY - IMAGE OPTIMIZATION

- WebP images
  - Lossless & lossy
- Dynamically sized images
  - Server provides appropriately sized images
  - Target rendering size
  - Adapt to network type & quality
  - Image placeholders
- Image loading libraries usage
  - Glide, Picasso, ...
  - One-time fetch
  - Optimized caching

glide

Picasso

© Stefan Jaindl

Build for Billions

CONNECTIVITY - OFFLINE CAPABILITY

- Local cache & data storage
- Queue network requests
- Two-way sync between server and database
  - Deltas
  - Failures
    - Permanent failures
    - Temporary failures (exponential backoff)
  - Idempotence (avoid duplicate data)
  - Conflict resolution
  - Local & server ID‘s
  - Files: upload immediately or store for later upload
- Minimize user notifications

Offline is a status, not an error

© Stefan Jaindl

Build for Billions

DATA COST

- Reduce app size
  - Use vector graphics (SVG), where possible
  - Use WebP for non-vector images
  - Multiple APK support for different screen densities
  - Reduce code size
    - Selective library and Google Play feature usage
    - Use ProGuard to shrink and remove unused resources & code
  - Put cached data in dedicated cache directory

- Configurable network use
  - Network settings/preferences screen
  - ACTION_MANAGE_NETWORK_USAGE
  - E.g. video bitrates, data synching, prefetching

© Stefan Jaindl

Build for Billions

DEVICE RANGES

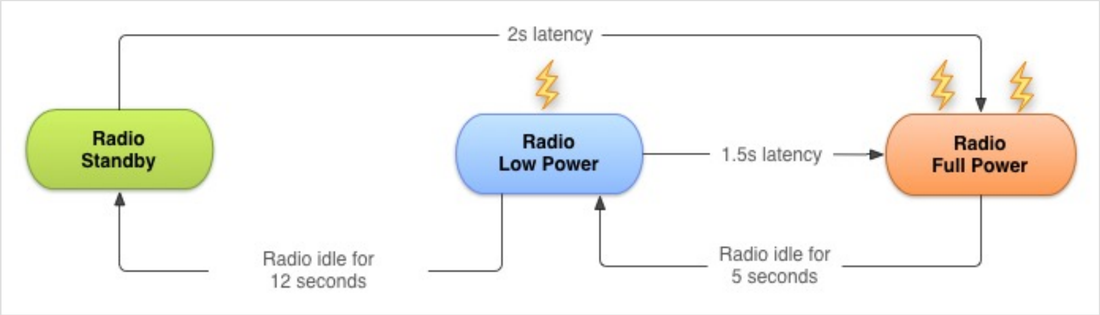
- Support multiple screen sizes
  - Use density-independent pixels (dp)
  - Test on ldpi/mdpi screen densities & small/medium screen sizes
  - Test heavy animations & graphics rendering
- Backwards compatibility
  - targetSdkversion
  - minSdkVersion
  - Use Android Support libraries
  - Use Google Play services
- Reduce memory usage on low-cost devices
- Avoid long-running processes (e.g. use FCM)
- Benchmarking

© Stefan Jaindl

Build for Billions

BATTERY CONSUMPTION

- The radio state machine
  - Latency vs. battery consumption
- Avoid wake lock usage
- Location: Use FusedLocationProvider API (accuracy requirements, caching, cross-app requests)
- Use WorkManager API
  - Optimize network & charging state, retries & backoff



© Stefan Jaindl

Build for Billions

BATTERY CONSUMPTION

- Batch network requests
- Prefetch data when possible
- Greater bandwidth: Download more data less often
- Check network connection before making requests
- Connection pooling (supported by e.g. OkHttp)
- Benchmarking: Batterystats + Battery Historian

Continuous requests:

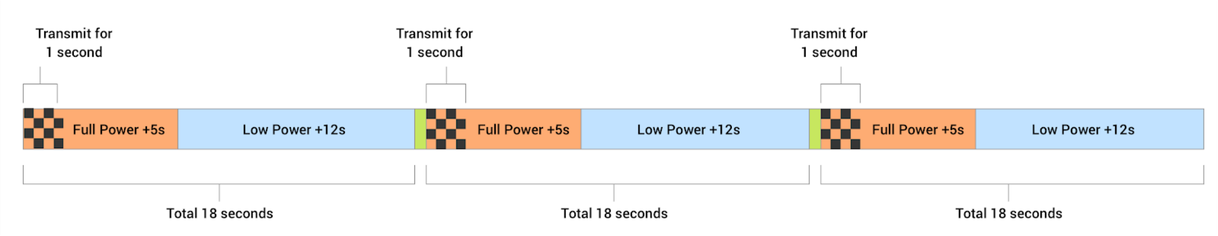
Batched requests:

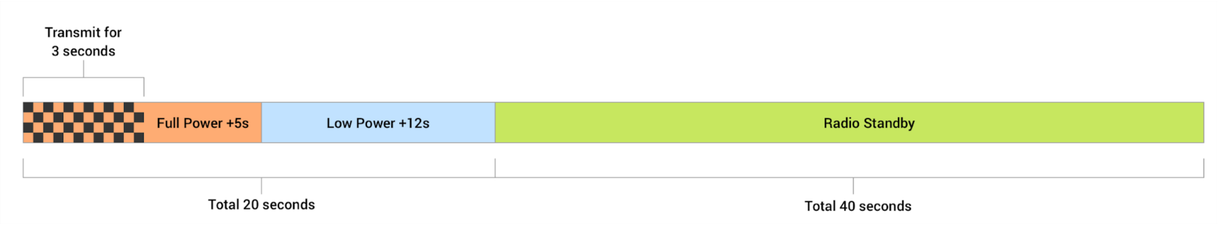
© Stefan Jaindl

Build for Billions

BATTERY CONSUMPTION

- User-Initiated network traffic
  - User experience vs. efficiency tradeoff
  - Caching
  - Throttling
- App-Initiated network traffic
  - Batch & prioritize data
  - Increase inactivity periods
  - Network activity on WiFi and charging state (WorkManager)
- Server-Initiated network traffic
  - FCM vs. Long polling





© Stefan Jaindl

Build for Billions

BATTERY CONSUMPTION

- Resumable uploads
  - Init – Append – Finalize
  - Efficient on unreliable networks for large files

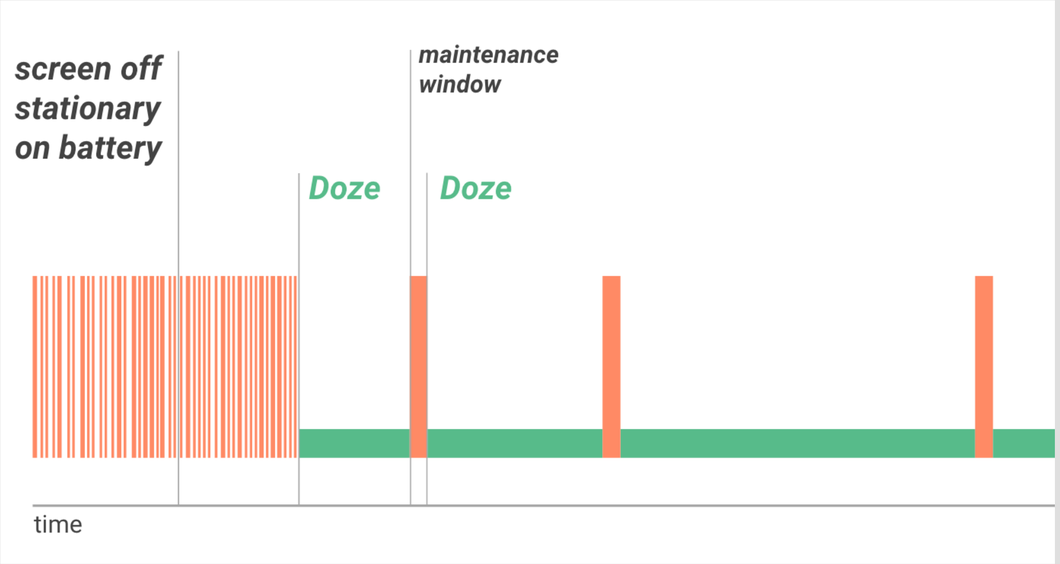
- Quality of Service
  - Limit number of concurrent network operations
  - Priority queue processing requests in order of importance
  - Service types, e.g.:
    - User critical
    - UI critical
    - UI non-critical
    - Background

© Stefan Jaindl

Build for Billions

DOZE MODE

- API level >= 23
- Device unplugged and screen-off for a certain time
- Restrictions
  - Defer network & CPU activity, jobs, syncs & standard alarms
  - Wake locks ignored
  - No WiFi scans
- Maintenance windows
  - Less frequently over time
- Cause actions in Doze Mode
  - FCM high priority messages (downstream)
  - Critical alarms



© Stefan Jaindl

Build for Billions

APP STANDBY

- API level >= 23
- Defer background work for not-recently-used apps
- Conditions
  - Explicit launch, foreground process, notification, device admin apps
- Buckets
  - Working set
  - Frequent
  - Rare
  - Restricted
- Cause actions in App Standby
  - FCM high priority messages (downstream)

Setting	Jobs *	Alarms †	Network ‡	Firebase Cloud Messaging §
User Restricts Background Activity				
Restrictions enabled:	Never	Never	No restriction	No restriction
Doze				
Doze active:	Deferred to window	Regular alarms: Deferred to window While-idle alarms: Deferred up to 9 minutes	Deferred to window	High priority: No restriction Normal priority: Deferred to window
App Standby Buckets (by bucket)				Prior to Android 13 (API Level 33)
Active:	No restriction	No restriction	No restriction	No restriction
Working set:	Deferred up to 2 hours	Deferred up to 6 minutes	No restriction	No restriction
Frequent:	Deferred up to 8 hours	Deferred up to 30 minutes	No restriction	High priority: 10/day
Rare:	Deferred up to 24 hours	Deferred up to 2 hours	Disabled	High priority: 5/day
Restricted:	Once per day	One alarm per day, either an exact alarm or an inexact alarm	Disabled	High priority: 5/day

© Stefan Jaindl

Build for Billions

USER INTERFACE

- Fast & responsive UI
  - Touch feedback
  - Interactive UI
    - Don‘t misuse Main Thread
    - Material design, single activity indicators
  - Target 60 frames per second on low-cost devices (= 16 ms per refresh)
    - Efficient layout hierachies
    - Reduce animations & overdraw
  - ViewStubs
  - Include & merge tags
  - Launch screen

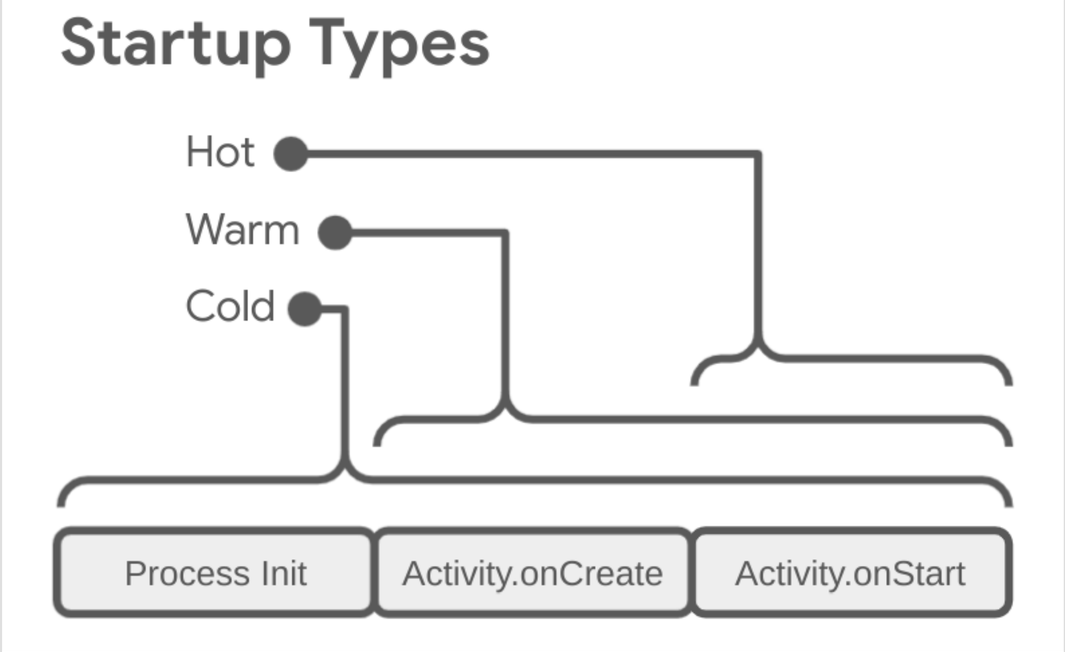
- Localization

© Stefan Jaindl

Build for Billions

APP STARTUP

- Cold start
  - Launching app, create application process
  - Create app object & main thread
  - Inflate, layout & draw MainActivity
- Warm start
  - Relaunch app
  - Starts from Activity.onCreate()
- Hot start
  - Bring activity to foreground
- Time to initial display (TTID) metric
- CPU profiler
- Issues
  - Heavy app or activity initialization
- Jetpack App Startup Lib

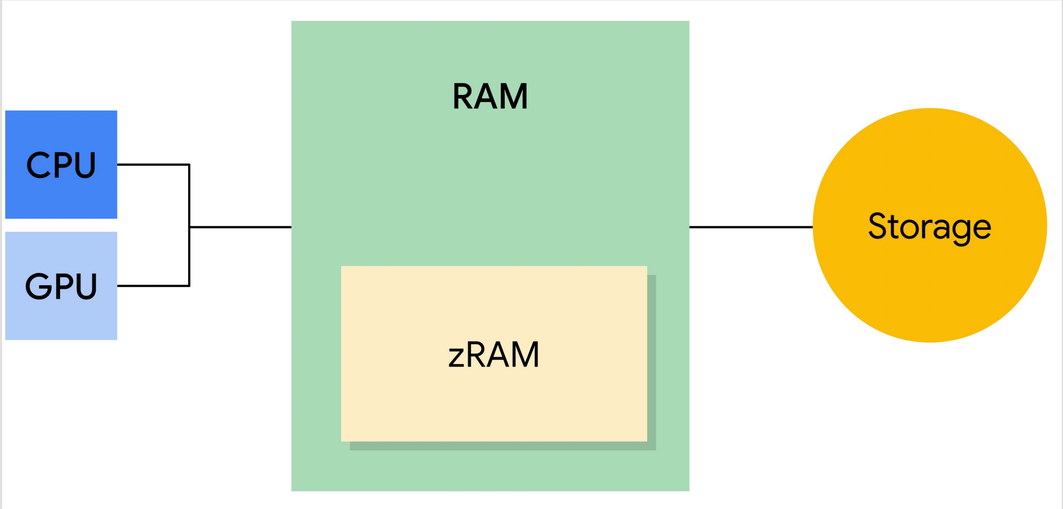


© Stefan Jaindl

Build for Billions

MEMORY

- Android system uses as much memory as possible
- Paging & memory mapping
- Garbage Collection
  - Detect and reclaim unused objects to the heap
- Shared memory
  - RAM pages shared across processes
- Hard limit on heap size per app (OutOfMemoryError)
- App switching – process cache
- Memory types
  - RAM (fastest, fewest space)
  - zRAM (compressed swap space)
  - Storage (slowest, most space, persistent data)

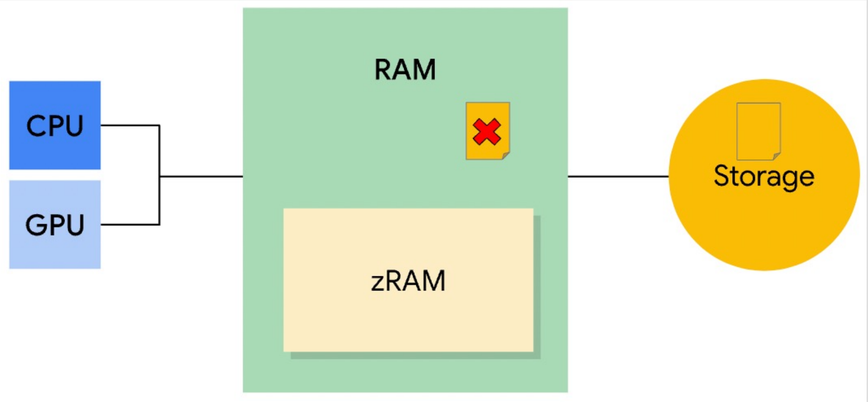


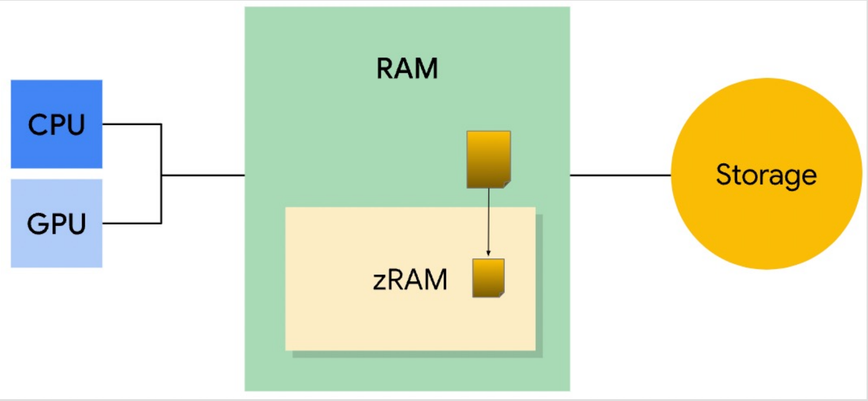
© Stefan Jaindl

Build for Billions

MEMORY

- Memory pages
  - Typically 4KB size
  - Free or used
  - „Cached“ (private or shared) or „Anonymous“
- Low Memory Management
  - Kernel swap daemon
  - Low-memory killer
    - onTrimMemory()
    - Kill processes





© Stefan Jaindl

Build for Billions

MEMORY

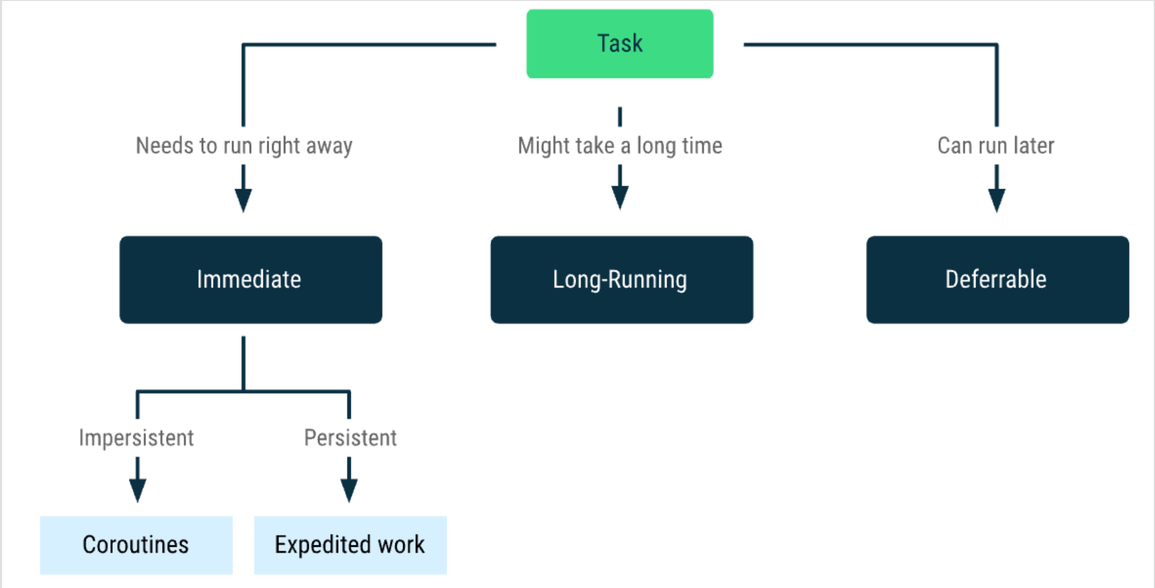
- Avoid memory leaks
  - Don‘t hold object refs in static vars
  - Release objects in appropriate lifecycle callbacks
- Memory Profiler, leakcanary
- Implement onTrimMemory()
- Memory info: ActivityManager.MemoryInfo()

© Stefan Jaindl

Build for Billions

MEMORY - REDUCE MEMORY USAGE

- Appropriate background work usage
  - Persistent work: WorkManager
  - Immediate impersistent work: Coroutines
  - Stop unneeded services (use IntentService)
- Use optimized data structures (e.g. SparseArray)
- Be careful with code abstractions
- Consider lite protobufs for serialized data
- Identify and fix memory churn
- Compile-time dependency injection (e.g. Hilt)
- Check 3rd party libs
- Reduce APK size, use proGuard



© Stefan Jaindl

Interview Strategies

PROCESS, TIMING & ADVICE

© Stefan Jaindl

Interview Strategies

INTERVIEW PROCESS & TIMING

- Total: 45 - 60 minutes:
  - Min 0-5: Getting to know
  - Min 5-10: Question, define task
  - Min 10-15: Define requirements
  - Min 15-25: High level design
  - Min 25-40: Detailed design
  - Min 40-45: Questions to the interviewer

© Stefan Jaindl

Interview Strategies

INTERVIEW PROCESS

- Min 0-5: Getting to know

- Min 5-10: Question, define task
  - Repeat question
  - Write down
  - Ask clarifying questions & understand
  - Listen also for details
  - Walk through examples, if applicable
  - Never jump directly to next steps!

© Stefan Jaindl

Interview Strategies

CLARIFYING QUESTIONS

- Context
  - Start from scratch or existing system?
  - General purpose or specific?
  - Who are the users?
  - What are the use cases? How is the system used?

- Implementation scope
  - Client-side, client-side + API, client-side + API + backend?
  - Multiplatform?

© Stefan Jaindl

Interview Strategies

CLARIFYING QUESTIONS

- Networking
  - Concrete client-server communication method required?
  - How often does the data change? Is it real-time data?
  - Data flows one-way or bidirectional?
  - API versioning needed?
  - API internal or customer-facing?
  - How important is API performance?
  - Are different server-side teams/programming languages needed?
  - Amount of data returned flexible?
    - Pagination
  - Up-/downloads: Progress reporting needed?

© Stefan Jaindl

Interview Strategies

CLARIFYING QUESTIONS

- Storage
  - Should data be persisted in local storage?
  - How does the data look like?
  - Size constraints?
  - File/data format?
  - Sensitive data to store?
    - Android keystore

© Stefan Jaindl

Interview Strategies

CLARIFYING QUESTIONS

- Building for Billions
  - Support for emerging markets?
    - Low-cost devices, battery, data cost, app size, ...
  - How many users will be using the system? Growth expectations?
    - Think beyond app scope - More users normally mean higher backend load
    - API design, caching, network optimizations
  - Resumable/pausable/cancelable up- or downloads?
    - HTTP ranges or transfer encoding?
    - Simultaneous connections?
  - Should the app be optimized for storage, memory, network or CPU efficiency?

© Stefan Jaindl

Interview Strategies

CLARIFYING QUESTIONS

- Further topics
  - Authentication & authorization
  - Privacy
  - Size of engineering team
    - Project structure and modularization (small vs. big team)

© Stefan Jaindl

Interview Strategies

INTERVIEW PROCESS

- Min 10-15: Gather requirements
  - Focus on most important requirements
  - Functional requirements
    - 3-5 user-visible functionalities
  - Non-functional requirements
    - Non-user-visible, but generally important
    - Availability (offline support, phones/tablets, devices, screen sizes, languages, features/permissions)
    - Scalability (modularisation, 3rd party libs usage)
    - Performance (battery, network, memory, CPU-intensive work)
    - Deployment (CI/CD, feature flags/remote config)
    - Testability (DI, unit tests, integration tests, etc.)
    - Security (sensitive data, encryption, authentication, authorization, HTTPS usage)
  - Out of scope
    - Authentication/authorization, analytics, etc.

© Stefan Jaindl

Interview Strategies

TASK + REQUIREMENTS - WHAT‘S IMPORTANT?

- System design question are often deliberately ambiguous

- Interviewer looks for:
  - What clarifying questions did you ask?
  - What (functional & non-functional) features did you define?
  - Have you completely understood the task before diving into the actual design?
  - What concerns and trade-offs did you discuss?

© Stefan Jaindl

Interview Strategies

INTERVIEW PROCESS

- Min 15-25: High level design
  - Identify system components
    - Server-side
      - Backend web server, CDN, push provider, …
    - Client-side
      - Modules / components of chosen functional requirement(s)
      - API-Services
      - Persistence layer
      - Repositories
      - Routing
      - 3rd party libs
  - Client-Server communication method/protocol
  - Draw system diagram, if applicable

© Stefan Jaindl

Interview Strategies

HIGH LEVEL DESIGN - WHAT‘S IMPORTANT?

- Interviewer looks for:
  - Did you understand the big picture?
  - Are you thinking about how to structure/modularize the system?
  - Can you define how components/modules interact with each other?

© Stefan Jaindl

Interview Strategies

INTERVIEW PROCESS

- Min 25-40: Detailed design
  - Choose specific feature (either interviewer or you)
  - Design lower-level components (exact repositories, services, UI, ViewModels, etc.)
  - Networking
    - Protocol, API design, pagination, failure handling, authentication, caching, etc.
  - Storage
    - Define data, discuss kind of data (sensitive data, amount of data, structured vs. unstructured)
    - Choose storage type

© Stefan Jaindl

Interview Strategies

INTERVIEW PROCESS

- Min 25-40: Detailed design
  - Build for billions
    - Connectivity, data cost, battery consumption & memory usage
    - Device ranges
    - Doze mode & app standby
    - UI responsiveness & localization
    - App startup time
  - Architectural patterns
    - MVVM, MPV, MVI, etc.
  - Additional topics
    - Image loading, location, dependency injection,
      privacy & security, accessibility, cloud vs. on-device, etc.

© Stefan Jaindl

Interview Strategies

DETAILED DESIGN - WHAT‘S IMPORTANT?

- Interviewer looks for:
  - Which topics are important to you?
  - How much experience do you have?
  - Are you familiar with app modularization/project structure and patterns?
  - Have you designed a clear separation between components and UI & business logic?
  - Are you working closely together with the interviewer?
  - Are you a nice person to work with? Is collaboration with you enjoyable?

© Stefan Jaindl

Interview Strategies

INTERVIEW PROCESS

- Min 40-45: Questions to the interviewer
  - End of actual interview may be suddenly
  - Prepare questions!
    - Company-related, job-related or relevant from a personal perspective
    - Could give important insights into company

© Stefan Jaindl

Interview Strategies

FINAL ADVICE

- Attitude: always be friendly and cooperative
- Preparation: Learn & repeat relevant topics, do mock interviews
- Knowledge: Personal projects, job, open-source projects, reading of tech blogs & articles
- Structured interview approach, do not rush!
- Practice with timing
- Always talk out loud what you are thinking
- Work together with the interviewer
- Don‘t give up!

© Stefan Jaindl

Mock Interviews

COMPLETE WALKTHROUGHS

© Stefan Jaindl

Mock Interview

CLARIFYING QUESTIONS / TASK

Task:

Design a chat application

© Stefan Jaindl

Mock Interview 1

REQUIREMENTS

Functional

- Send and receive chat messages in real-time
- Send and receive media attachments up to 1 GB
- Display a list of recent chats sorted by date
- Display message status (pending, delivered, read)
- Progress reporting for up- and downloads

Non-Functional

- Offline functionality
- Optimize network usage for emerging markets
- Minimize backend load
- Data should be stored in a secure storage

© Stefan Jaindl

Mock Interview 1

REQUIREMENTS

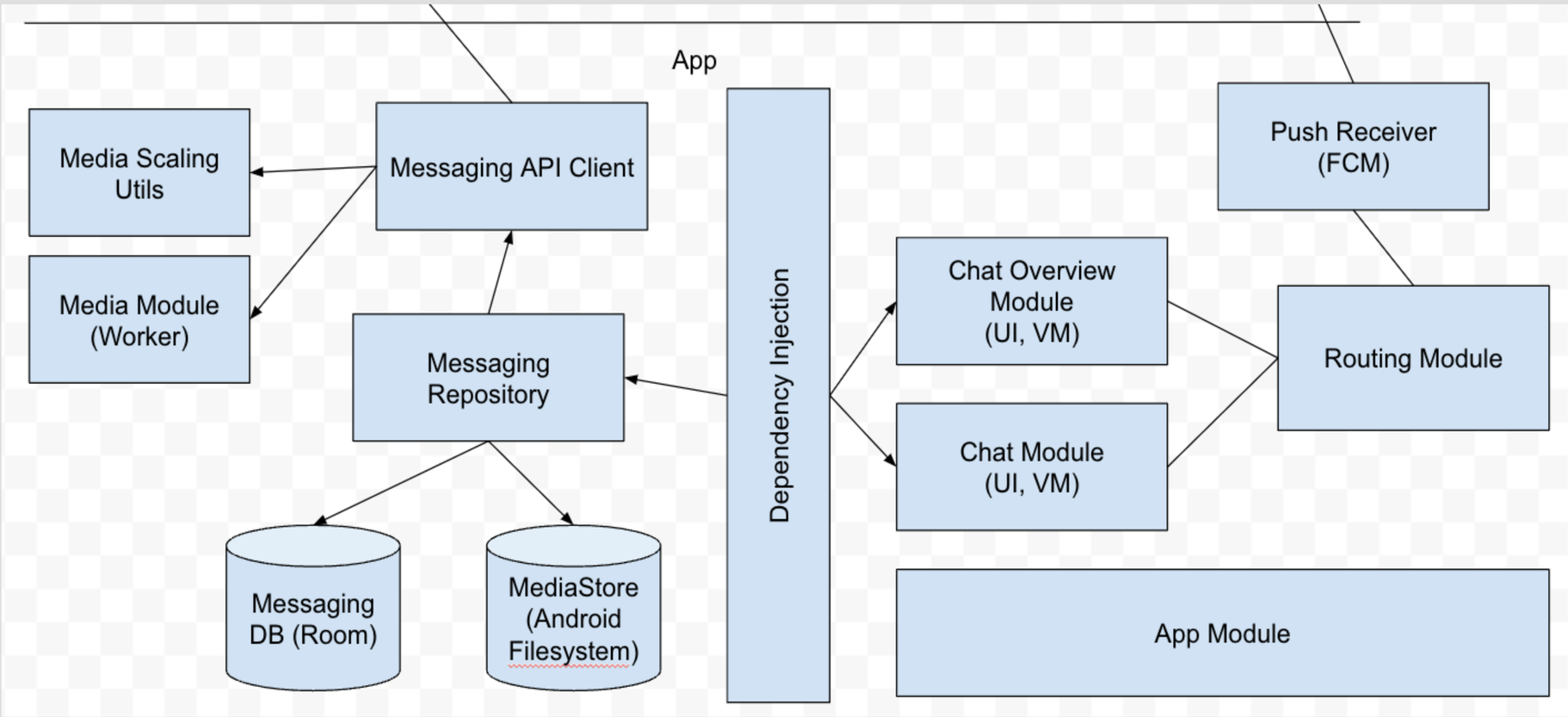
Out of scope

- Authentication & authorization
- Group chats
- Message editing & deletion

© Stefan Jaindl

Mock Interview 1

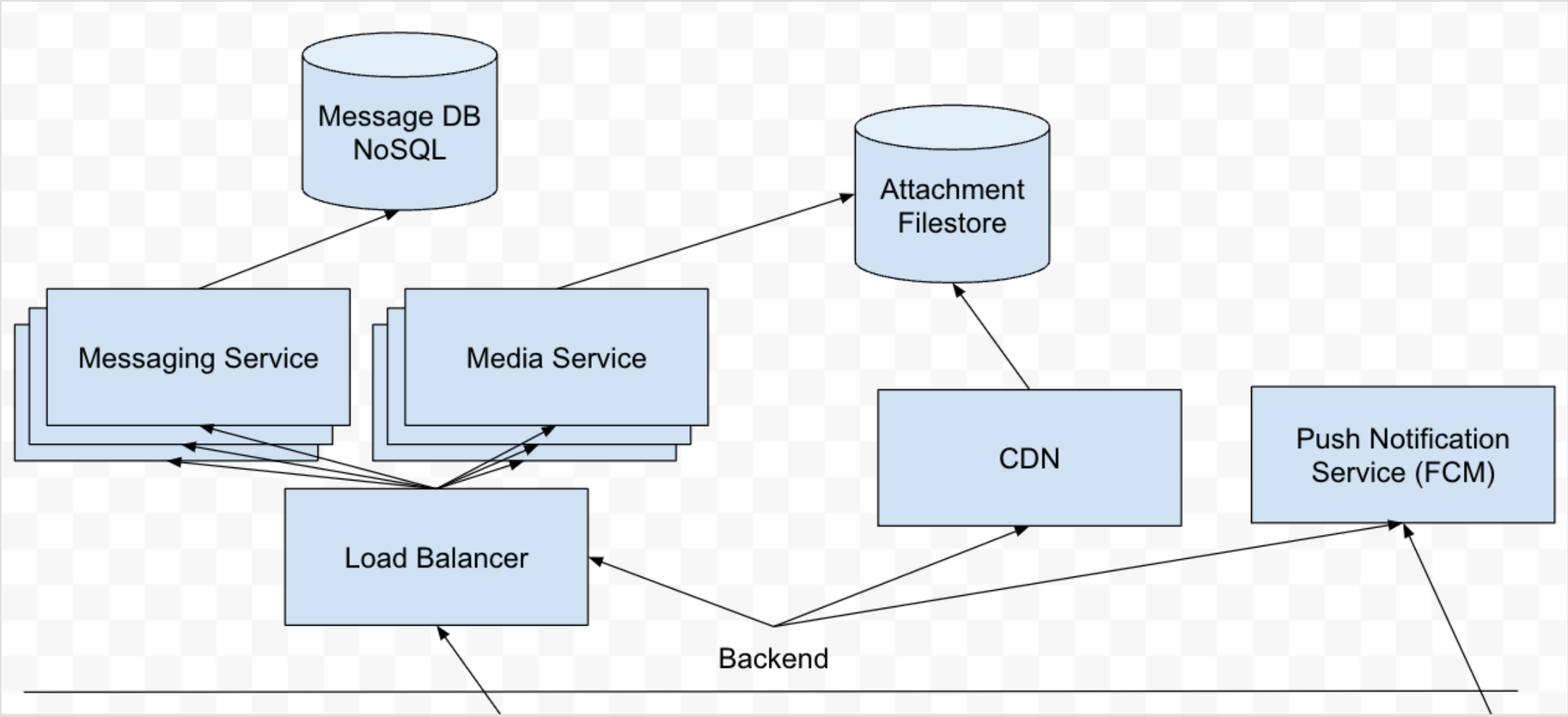
HIGH LEVEL DESIGN - CLIENT



© Stefan Jaindl

Mock Interview 1

HIGH LEVEL DESIGN - BACKEND



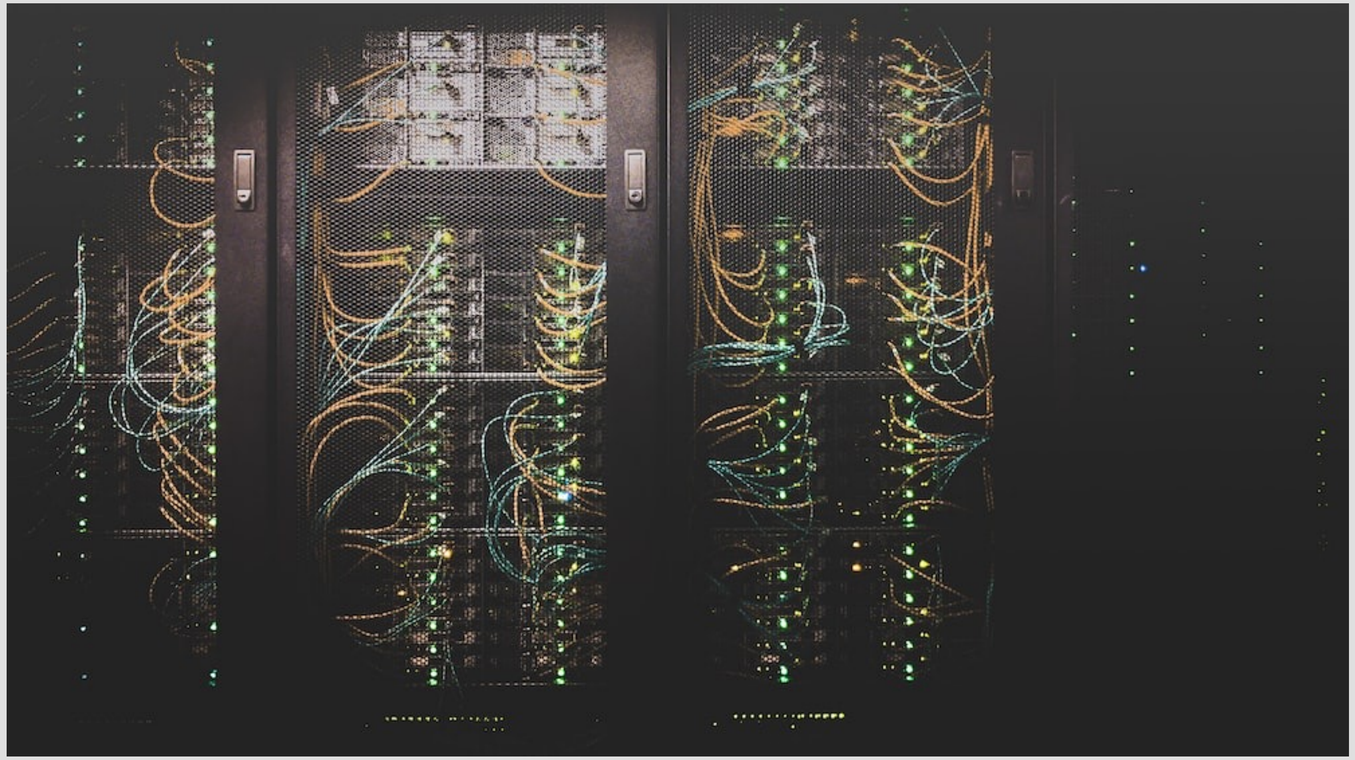
© Stefan Jaindl

Mock Interview 1

DETAILED DESIGN - NETWORKING

Task:

Design a chat application



© Stefan Jaindl

Mock Interview 1

DETAILED DESIGN – NETWORKING – GRPC SCHEMA

```
enum Status { PENDING = 1; DELIVERED = 2; READ = 3; }

message SendMessageRequest { string senderId = 1; string receiverId = 2; string messageId = 3;  bytes payload =
4; }
message SendMessageResponse { Status status = 1; }
rpc BidirectMessageStream(stream SendMessageRequest) returns (stream SendMessageResponse);


message MessageStatus { Status messageId = 1; string status = 2; }
rpc Status returns (MessageStatus)
```

© Stefan Jaindl

Mock Interview 1

DETAILED DESIGN – NETWORKING – FCM

```
{
   "message": {
      "token":"bk3RNwTe3H0:CI2k_HHwgIpoDKCIZ...",
      "notification": {
         "title":”New message from John",
         "body":"Hello, I hope you are fine and ..."
      },
      "data" : {
         "senderName" : "john_doe",
         "lastMessageId" : "b32b2af123389fab",
         "attachmentId" : ”a66b2af182969c17"
   }
  }
}
```

© Stefan Jaindl

Mock Interview 1

DETAILED DESIGN – NETWORKING – REST

Authorization: Bearer <token>

Chat group:

GET /chats/{id}?limit=20&afterId=xyz

Chat Messages:

GET /chats/{id}/messages?limit=20&afterId=xyz

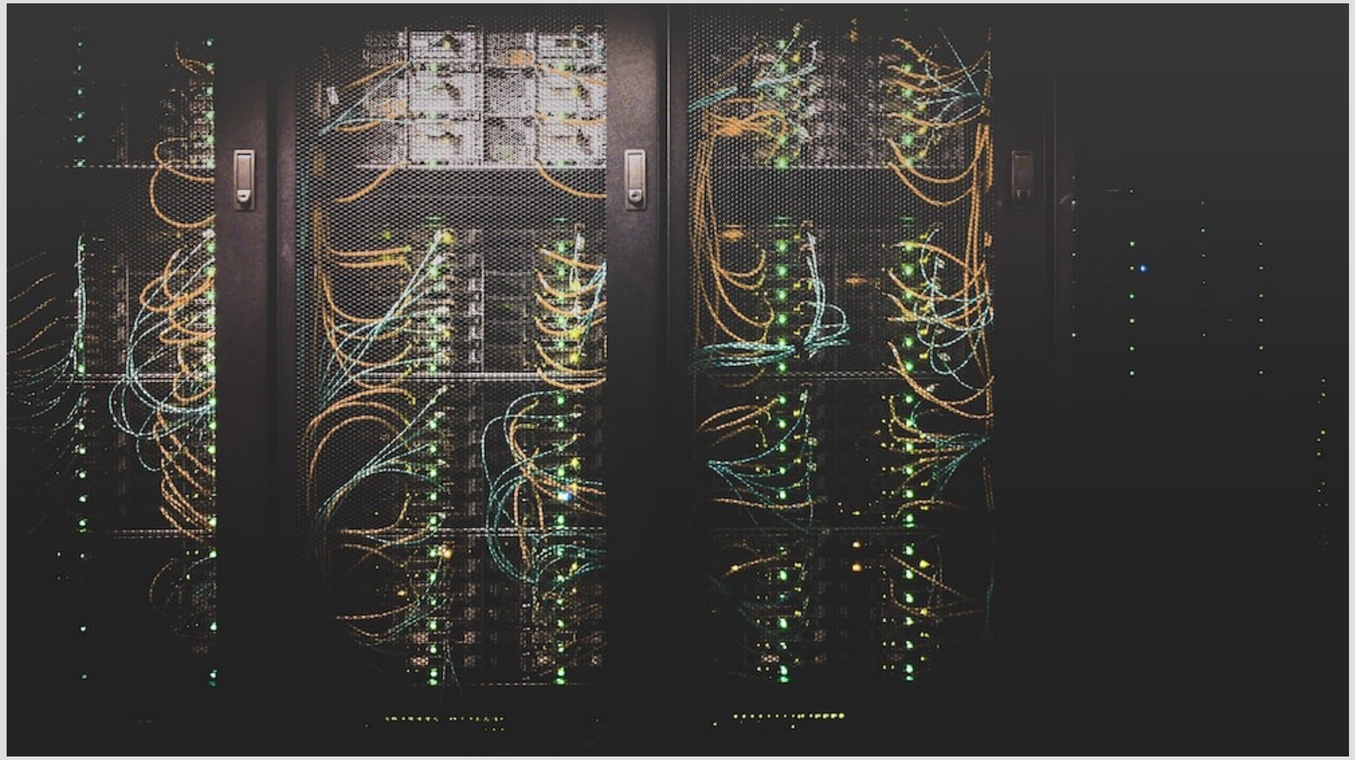
© Stefan Jaindl

Mock Interview 1

DETAILED DESIGN - NETWORKING

Task:

Design a chat application



© Stefan Jaindl

Mock Interview 1

DETAILED DESIGN – DATA STORAGE

Task:

Design a chat application

© Stefan Jaindl

Mock Interview 1

DETAILED DESIGN – DATA STORAGE

```
enum MessageStatus {
    case pending
    case send
    case delivered
    case read
}
```

Chat

- chatId: String
- chatPartner: User
- createdAt: Date
- messages: List<Message>

Message

- messageId: String
- message: String
- status: MessageStatus
- createdAt: Date
- sender: User
- attachments: List<Attachment>

User

- userId: String
- userName: String
- nickName: String
- profileUrl: URL

© Stefan Jaindl

Mock Interview 1

DETAILED DESIGN – DATA STORAGE

```
enum AttachmentType {
    case upload
    case download
}

enum AttachmentStatus {
    case ready
    case inProgress
    case failed
}
```

Attachment

- attachmentId: String
- type: AttachmentType
- localPath: String
- remoteUrl: URL
- sessionUrl: URL
- processedContentLengthBytes: Long
- totalContentLengthBytes: Long
- status: AttachmentStatus

© Stefan Jaindl

Mock Interview 1

INTERVIEW SUM UP

Task:

Design a chat application

© Stefan Jaindl

Mock Interview 2

CLARIFYING QUESTIONS / TASK

Task:

Design a Push Notification System

© Stefan Jaindl

Mock Interview 2

REQUIREMENTS

Functional

- Push notification manager app to receive push notifications
- Distribute push notifications and included data to relevant apps on the device
- Settings screen to subscribe and unsubscribe to multiple topics/components
- Configurable update frequency

Non-Functional

- Minimize backend load
- Minimize battery consumption
- Client heartbeats, reconnects
- Network failure handling

© Stefan Jaindl

Mock Interview 2

REQUIREMENTS

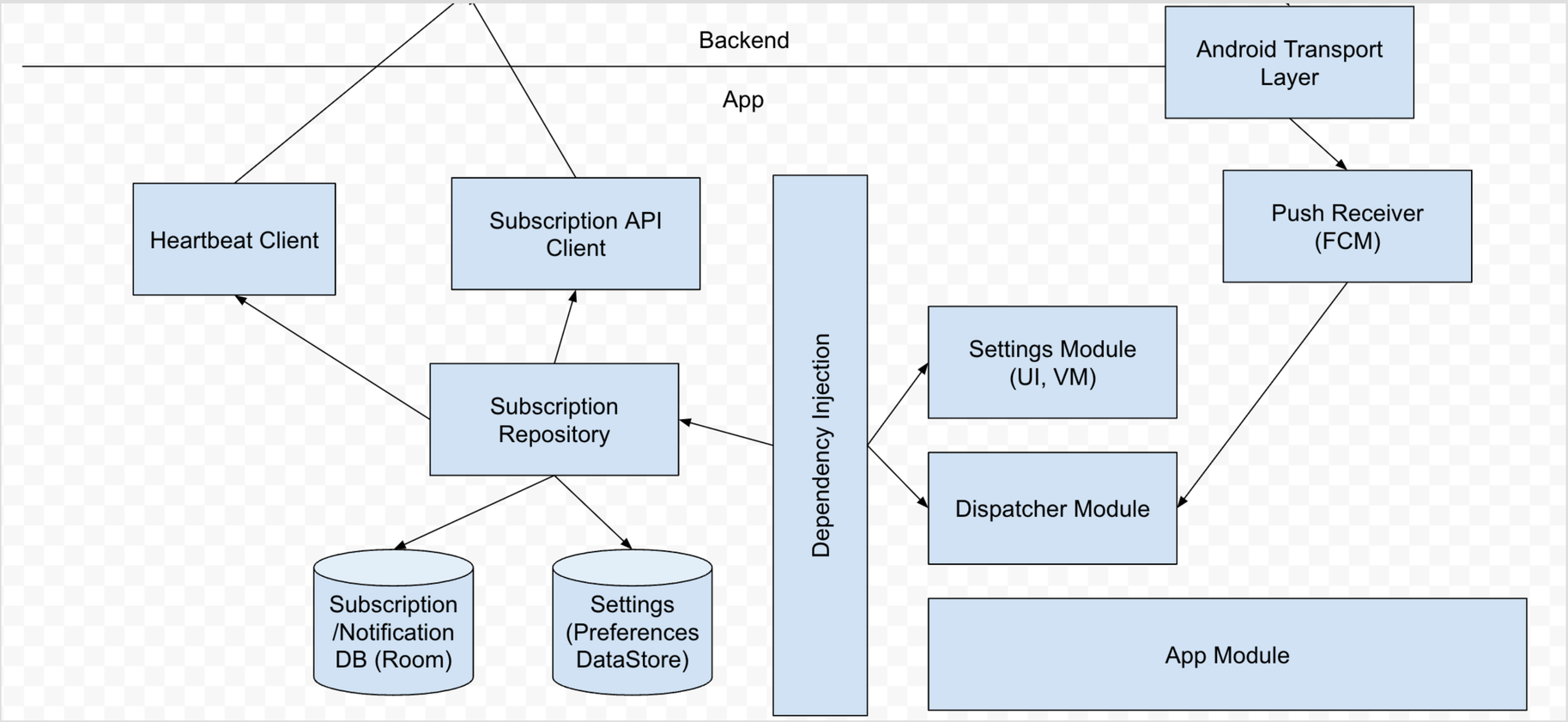
Out of scope

- Authentication & authorization
- Security-related topics (certificates, encryption, ...)
- Privacy

© Stefan Jaindl

Mock Interview 2

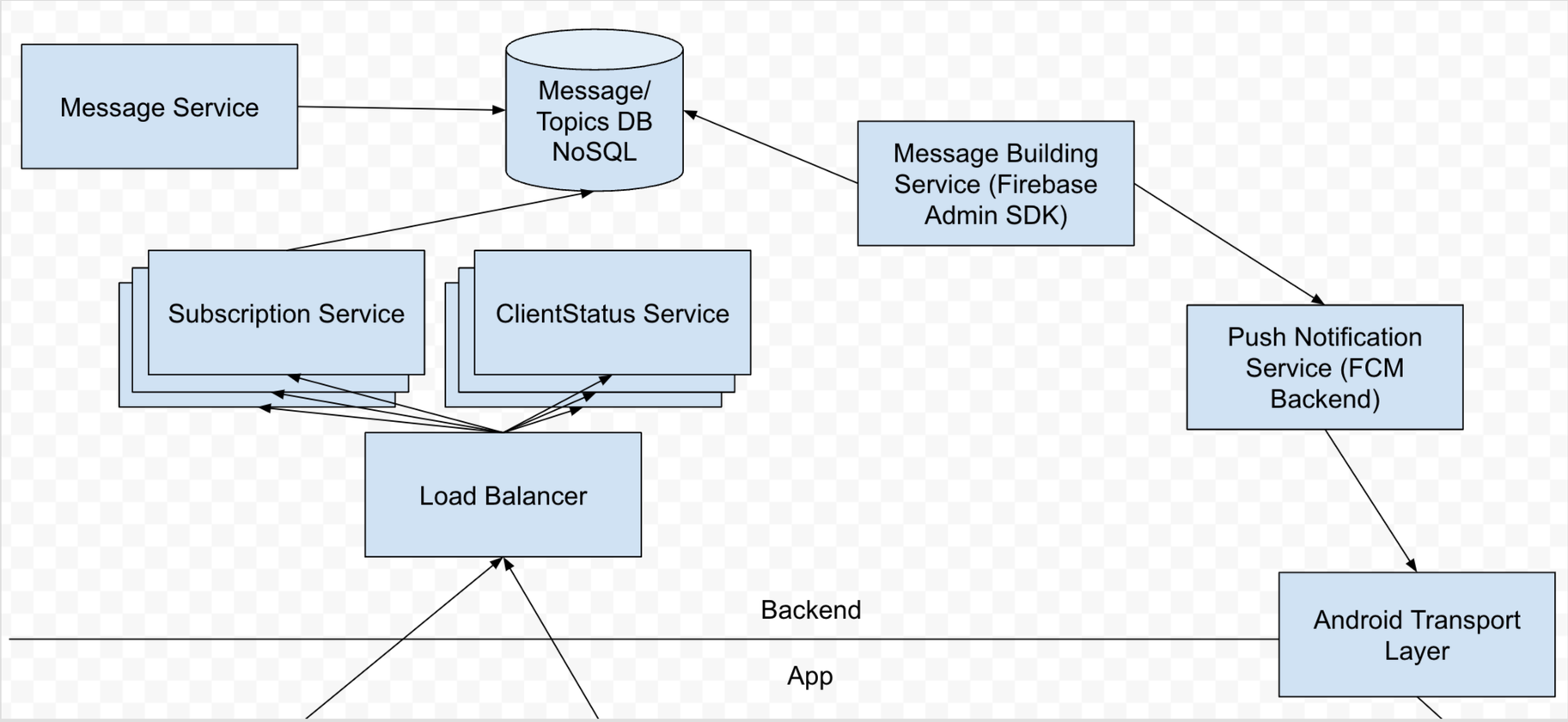
HIGH LEVEL DESIGN - CLIENT



© Stefan Jaindl

Mock Interview 2

HIGH LEVEL DESIGN - BACKEND



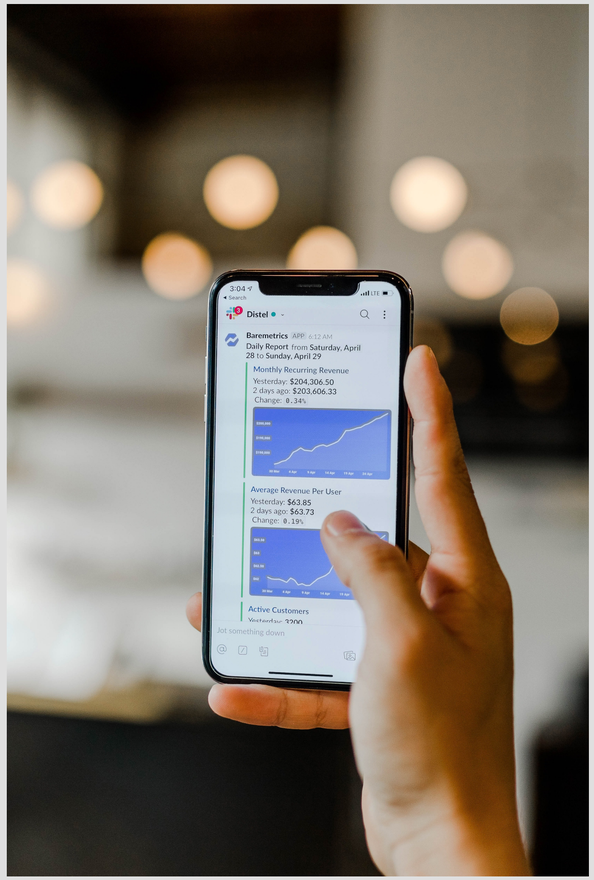
© Stefan Jaindl

Mock Interview 2

DETAILED DESIGN - SUBSCRIPTIONS

Task:

Design a Push Notification System



© Stefan Jaindl

Mock Interview 2

DETAILED DESIGN – PUSH NOTIFICATIONS & DISPATCHER

```
{
  "message": {
    "topic": "news",
    "notification": {
      "body": "Android 13 – latest features",
      "title": "Google News"
    },
    "data": {
      "action": "ACTION_VIEW",
      "data": {
        "scheme": "https",
        "host": "blog.google",
        "path": "/products/android/android-13"
      },
      ...
      ...
      "categories": [
          "android.intent.category.DEFAULT",
          "android.intent.category.BROWSABLE"
        ]
      },
      "android": {
        "priority": "normal"
      }
    }
  }
```

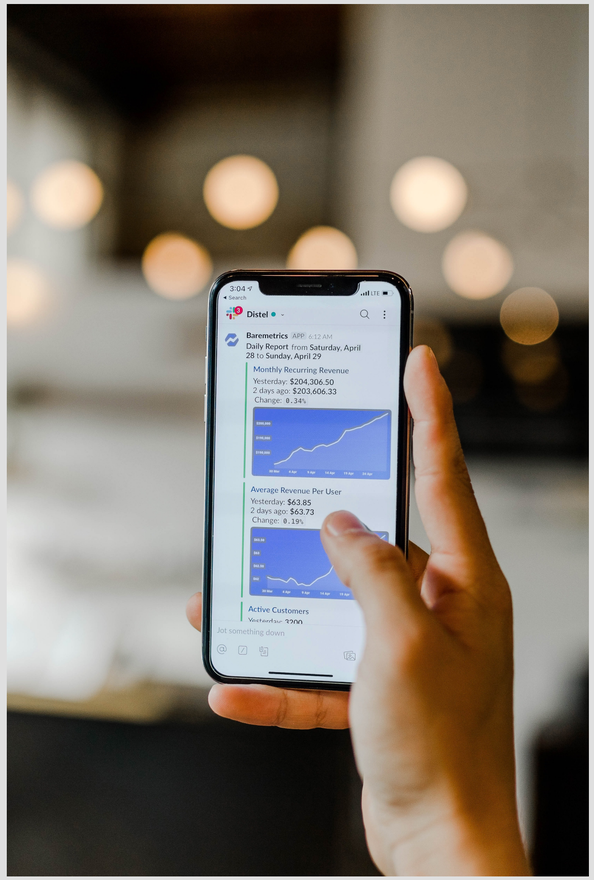
© Stefan Jaindl

Mock Interview 2

DETAILED DESIGN – HEARTBEATS & NETWORK FAILURES

Task:

Design a Push Notification System



© Stefan Jaindl

Mock Interview 2

INTERVIEW SUM UP

Task:

Design a Push Notification System

© Stefan Jaindl

Final Notes

LESSONS LEARNT

1. Networking
2. Storage
3. Building for Billions
4. Interview Tips & Tricks
5. Mock Interviews

LOOKOUT

Mastering the Android System Design Interview – Advanced Topics

1. Privacy & Security
2. Design Patterns / Architectures
3. Additional Topics
4. More Mock Interviews

© Stefan Jaindl

Mastering the Android System Design Interview

© Stefan Jaindl