API Challenges

Mobile-BFF API Design in µ-services Architecture

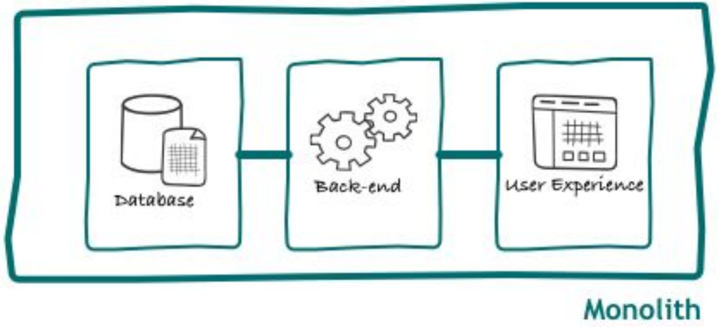
Outline

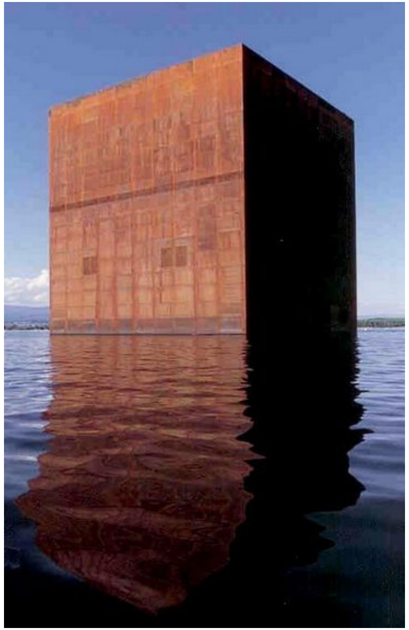
1. Context: µ-services Architecture
2. Backend Evolution
3. Mobile-BFF
4. REST
5. Challenge Definition

µ-Services Architecture: Monolith Hell

Definition:

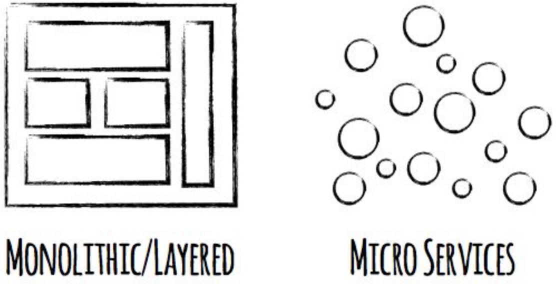
Monolithic application describes a **single-tiered software application** in which the **user interface and data access** code are **combined** into a **single program** from a single platform

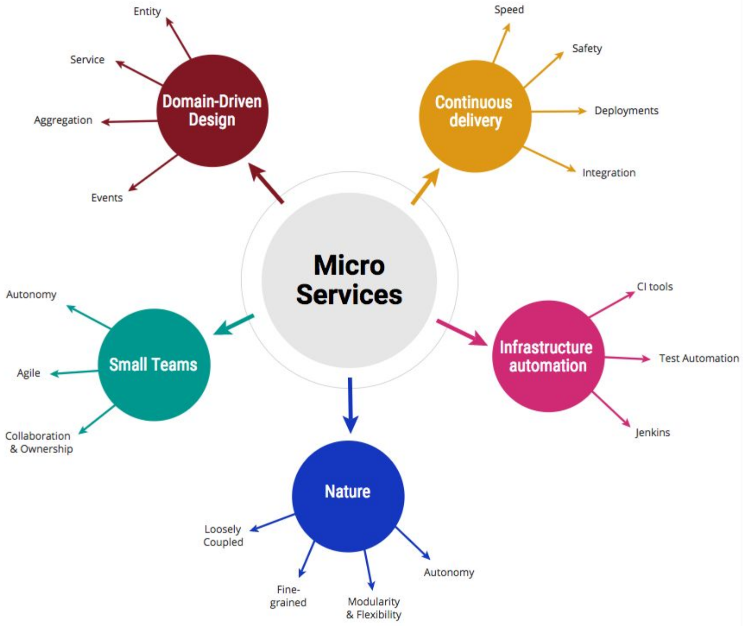




μ-Services Architecture: Def

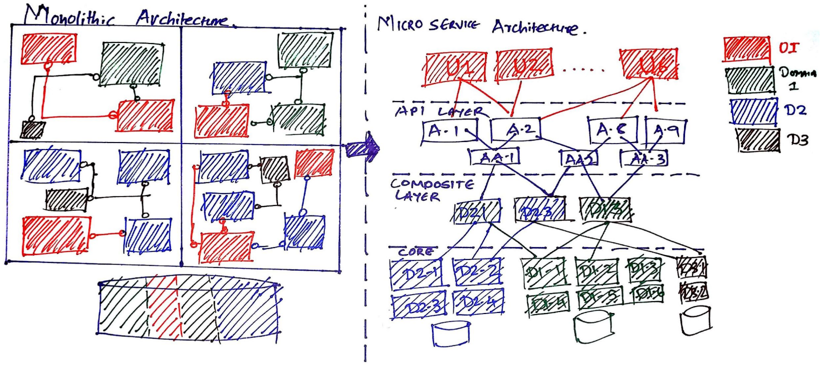
Microservices are **small**, **autonomous** services that **work together**





μ-Services Architecture: Def

Small, and **Focused** on **Doing One Thing Well**



<u>Monolith Problems</u>

- Large Code-Base → Boiler-Plates
- Deployment → expensive in time
- Difficult to have a good knowledge of the entire system

<u>Cohesion</u> → Single Responsibility Principle (Robert C. Martin):

"Gather together those things that **<u>change for the same reason</u>**, and separate those things that **<u>change for different reasons.</u>**"

μ-Services Architecture: Def

Autonomous

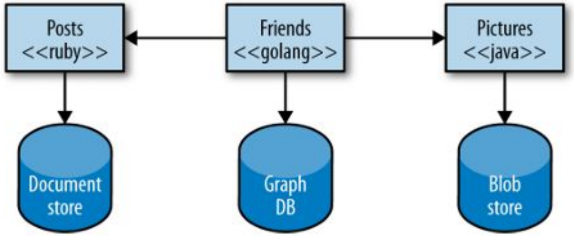
- **Separate entity** → deployed as an isolated service
- **Communication** → network calls → enforces between services
- Need to be able to **change independently** of each other
- Services expose an application programming interface (**API**)

Golden rule: can you make a change to a service and deploy it by itself without changing anything else?

μ-Services Architecture: Benefits

Technology Heterogeneity

- We can decide to use different technologies inside each one
- Pick the right tool for each job
- Faster tech adoption → decrease negative impact
- Example: Database Social Network
  - Graph-oriented database for users
  - Document-oriented data store for posts



µ-Services Architecture: Benefits

Resilience

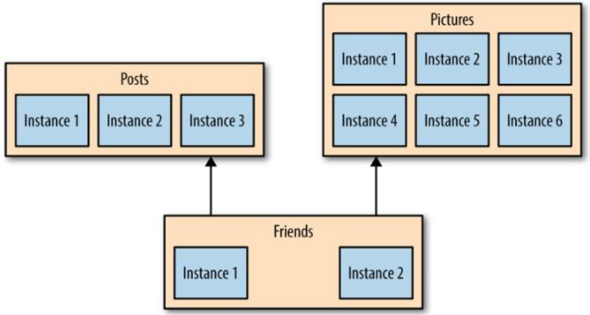
- In case of Failure (not cascade) → problem can be isolated

Ease of Deployment

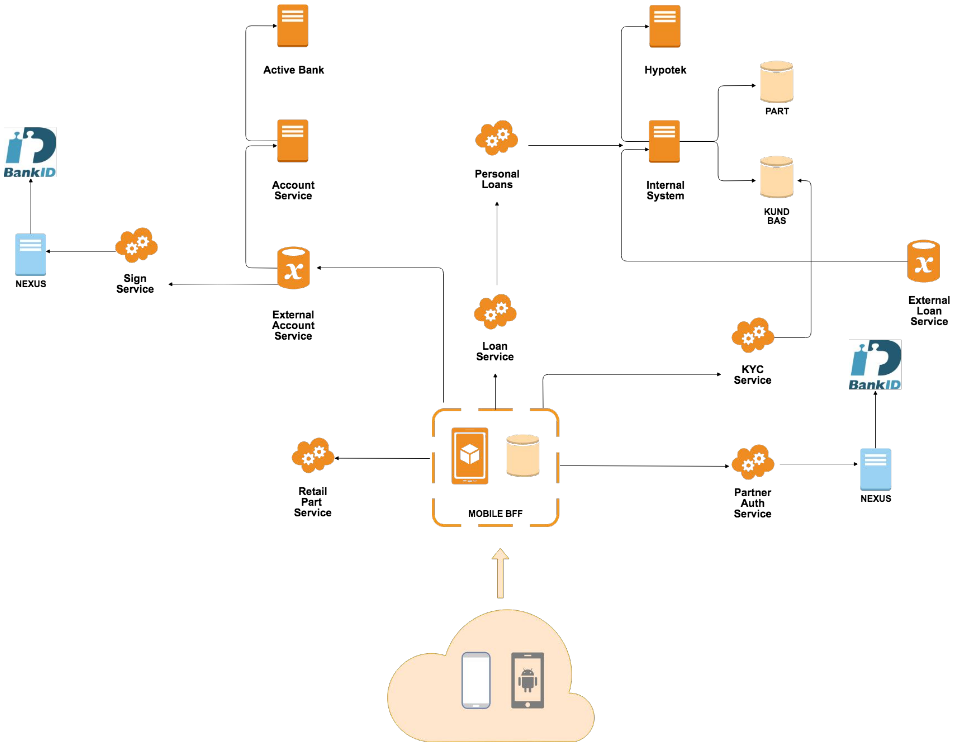
- Small changes → complete Monolith deployment → high risk
- Changes can be deployed into a single service → isolated risk
- Faster deployments → shorter time to market

Scaling

- We can scale small pieces (required ones), instead a big chunk (monolith)

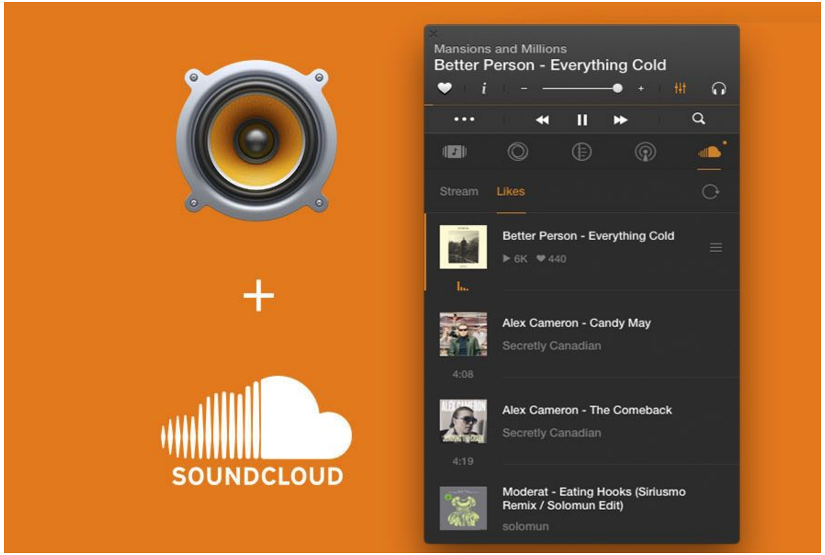


µ-Services Architecture: SBAB



Backend Evolution: SoundCloud

- Online audio distribution platform and music sharing website.
- Enables its users to upload, promote, and share audio.
- Founded in 2007 (Stockholm) by 2 Swedes
- 40M subscribed users



Mansions and Millions
Better Person - Everything Cold

Better Person - Everything Cold
▶ 6K ♥ 440

Alex Cameron - Candy May
Secretly Canadian
4:08

Alex Cameron - The Comeback
Secretly Canadian
4:19

Moderat - Eating Hooks (Siriusmo Remix / Solomun Edit)
solomun

Backend Evolution: SoundCloud

- Transition from stable and mature team → new incomers.
- Before → code review it was a formality through informal channels.
- After people leaving and new incomers arriving → problems with deployments → **Sol1: Stricter rules**

- **Problem1** → Stricter rules → more time to approve PRs & people avoiding large PRs
- **Sol2: Peer-programing**.

- **Problem2** → large code base → impossible for anyone to understand it all → swap pairs with “the expert” in that feature

Backend Evolution: SoundCloud

Why do we need Pull Requests?

Because often enough people make silly mistakes, push the change live and takes the whole platform down for hours.

Why do people make mistakes so often?

Because the code base is too complex. It’s hard to keep everything in your head.

Why is the code base so complex?

Because SoundCloud started as a very simple website, but grew into a large platform.

Why do we need a single code base to implement the many components?

The mothership already has a good deployment process and tooling, battle-tested architecture against...→ **Economy of Scope**

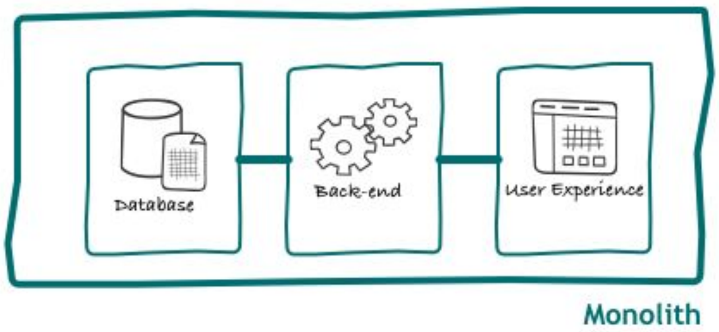
Backend Evolution: SoundCloud - Monolith

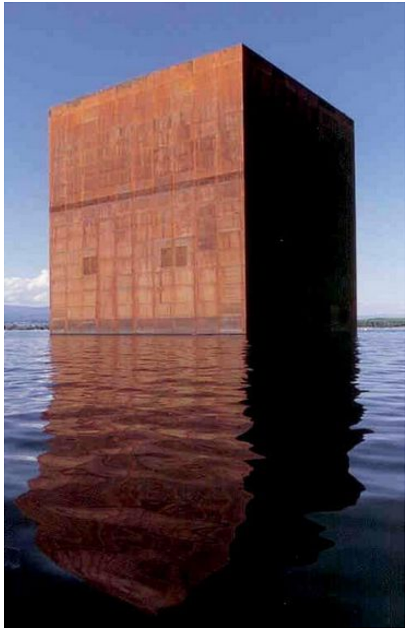
Why can’t we have **economies of scale** for multiple, smaller, systems?

μ-Services Architecture: Monolith Hell

Definition:

Monolithic application describes a **single-tiered software application** in which the **user interface and data access** code are **combined** into a **single program** from a single platform

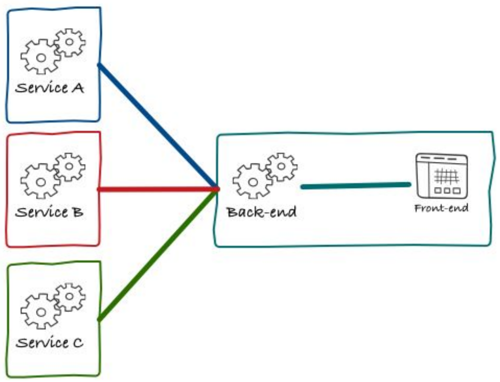




Backend Evolution: SoundCloud

- There was one system, and this system was the application.

- Many problems → decided to split Monolith and implement multiple Services

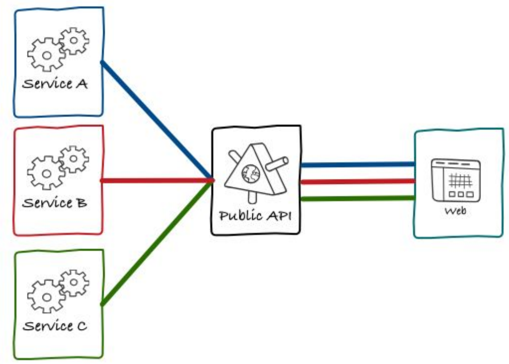


Backend Evolution: SoundCloud

- Main Motivation → reduce TimeToMarket

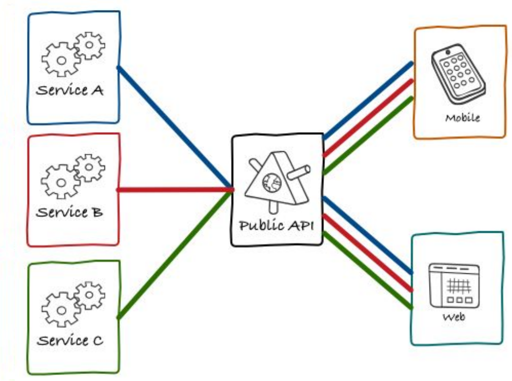
- Problem → bottleneck when touching the Monolith → UI changes really often :-(

- Solution → Extract UI to it's own layer a offer a public API



Backend Evolution: SoundCloud

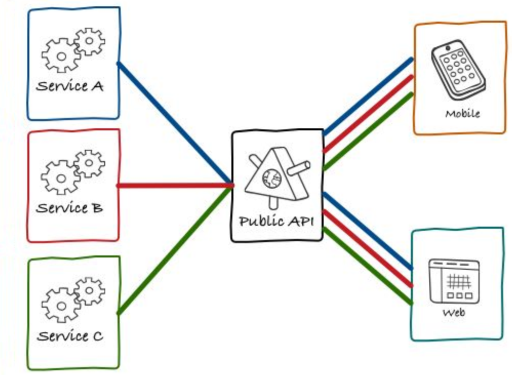
- Before 2011 → most clients were web
- After this point, mobile clients increased fast
- Solution → **Dogfooding**



Backend Evolution: SoundCloud

Problems

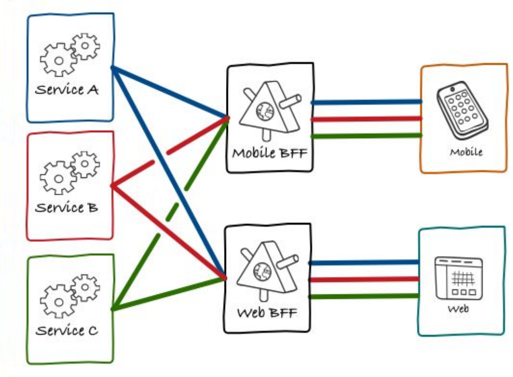
- **Nothing that we could offer** in our platform that wouldn't be available for third-party API
- **Fine-grained APIs** → empower third-party developers to build interesting integrations → more complex clients
- **API Bottleneck** → be sure changes not breaking any client (or 3rd parties) or over-specialized a client
- **iOS client** → massive project



Backend Evolution: SoundCloud

Solution

- **Different backends** → no coordination → more speed → (primitive) **Backend For Frontend borns**



Mobile-BFF: General Purpose Backends

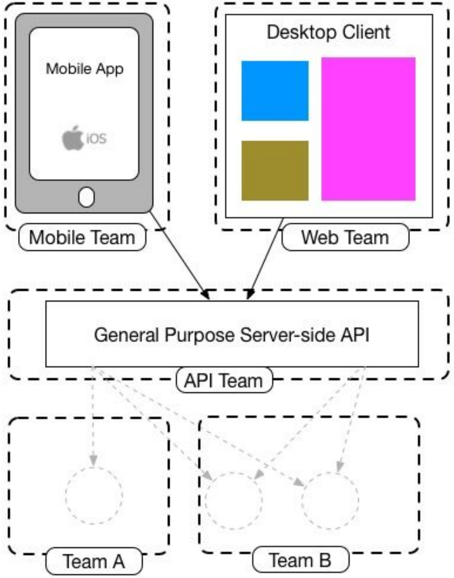
Problems:

- Mobile → different nature. Fewer calls, less data than desktop → we need more functionalities

- API → bottleneck

- Specific code to handle different platforms → middleware (against SOA)

Conclusion → Different clients have different needs and they expect something different from you



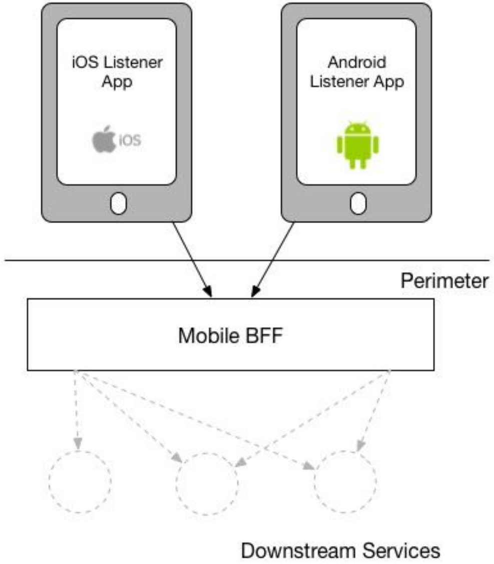
Mobile-BFF: What?

BFF is **tightly coupled to a specific UX** → maintained by the same team as the user interface

BFF is **tightly focused on a single UI**, and just that UI. That allows it to be focused, and will therefore be smaller.

Architectural Pattern → **API Composition** **(Gateway)** → **Data Aggregation**

Idea → **One experience, one BFF**



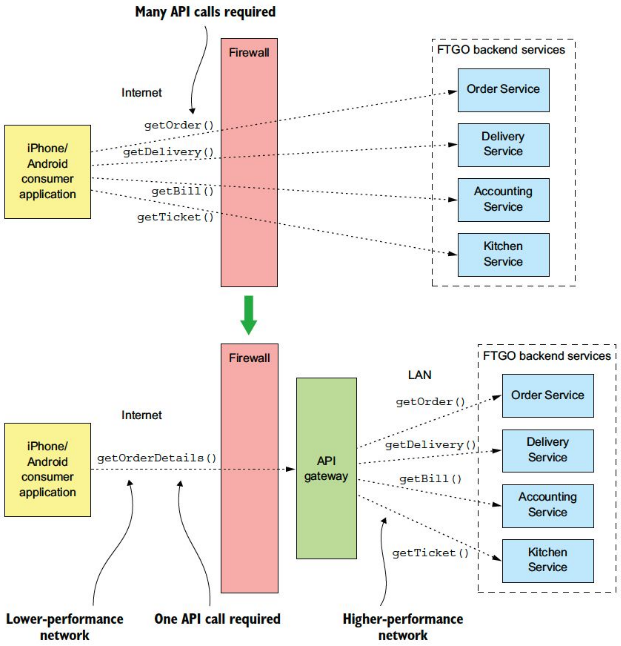
Mobile-BFF: Why?

Insulates the clients from how the application is partitioned into microservices → SOA not exposed

Hides changes of SOA → minimal impact on client (BFF will handle them).

- # service instances, service locations
- Service partitioning. Ex: Account Service → Account Service and Transfer Service

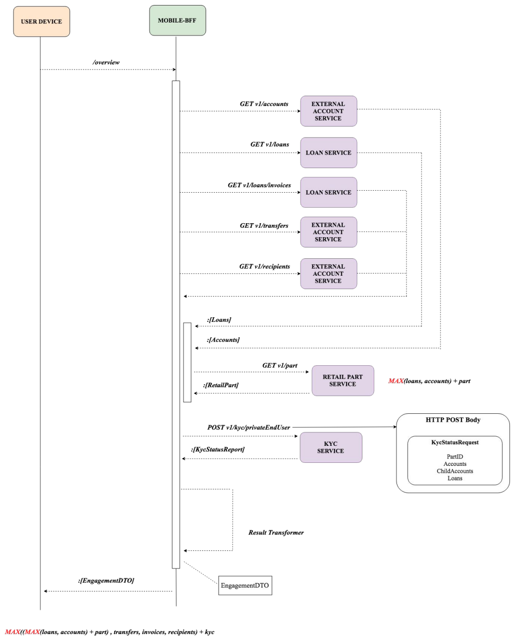
More → Authentication, Authorization, Rate limiting, Caching, Metrics collection, Request logging...



Mobile-BFF: Data Aggregation

1. **Reduces the number of requests/roundtrips** (all network latency gets reduced)
2. Provides the **optimal API for each client** → highly customized APIs → from fine-grained to custom
3. **Simplifies the client** by moving logic the client to the → API gateway → avoids boiler-plates

ENGAGEMENT DIAGRAM

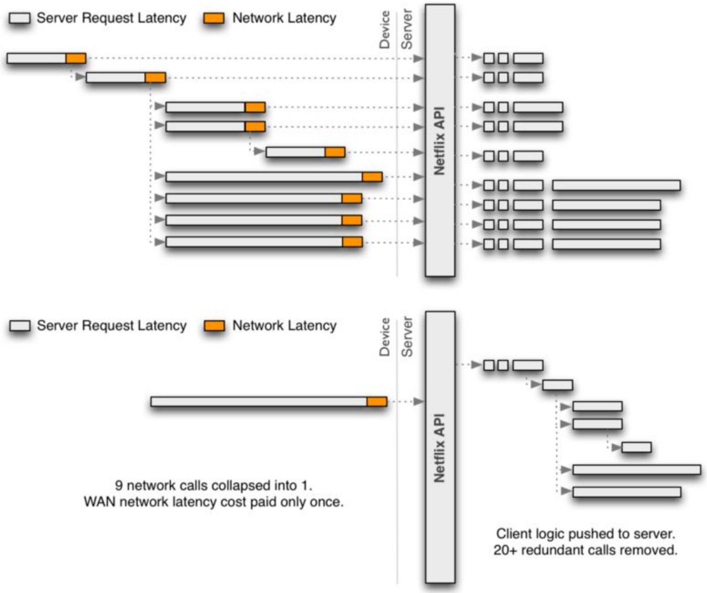


Mobile-BFF: Data Aggregation

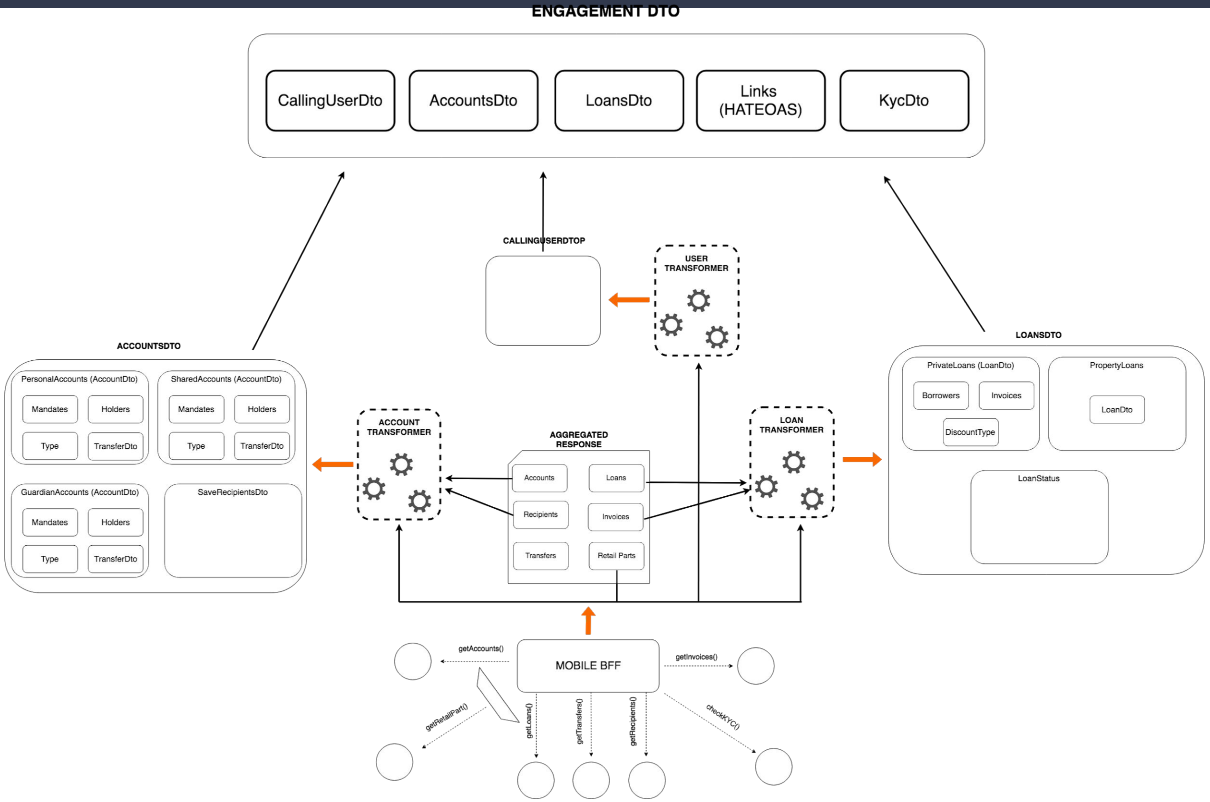
Typical, proximate, values for latency that you might experience include:

- 10ms for a modern Carrier Ethernet
- 20ms BT IP Connect, when using Class of Service to prioritise traffic
- **60ms** for 4G cellular data
- **120ms** for 3G cellular data
- 800ms for satellite

Sweden → **45ms - 85ms** (3G→ 4G)


Server Request Latency
Network Latency
Device
Server
Netflix API
Server Request Latency
Network Latency
Device
Server
Netflix API
9 network calls collapsed into 1.
WAN network latency cost paid only once.
Client logic pushed to server.
20+ redundant calls removed.

Mobile-BFF: Data Aggregation



ENGAGEMENT DTO

CallingUserDto

AccountsDto

LoansDto

Links (HATEOAS)

KycDto

CALLINGUSERDTOP

USER TRANSFORMER

ACCOUNTSDTO

PersonalAccounts (AccountDto)

Mandates

Holders

Type

TransferDto

SharedAccounts (AccountDto)

Mandates

Holders

Type

TransferDto

GuardianAccounts (AccountDto)

Mandates

Holders

Type

TransferDto

SaveRecipientsDto

ACCOUNT TRANSFORMER

AGGREGATED RESPONSE

Accounts

Loans

Recipients

Invoices

Transfers

Retail Parts

LOAN TRANSFORMER

LOANSDTO

PrivateLoans (LoanDto)

Borrowers

Invoices

DiscountType

PropertyLoans

LoanDto

LoanStatus

MOBILE BFF

getAccounts()

getInvoices()

getRetailPart()

getLoans()

getTransfers()

getRecipients()

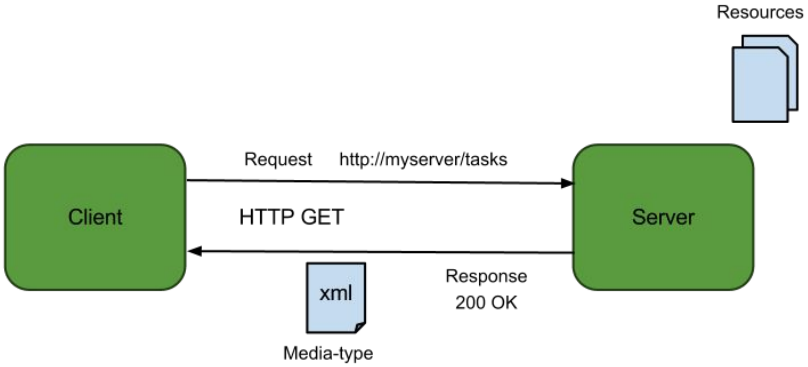
checkKYC()

REST

Representational **S**tate **T**ransfer

Architectural style → set of constraints

Build over HTTP



1. Client and Server model
2. Stateless
3. Cache
4. Uniform Interface
    a. Resource → Invoice (GET /invoices/{invoiceId}
    b. Representations
    c. Self Descriptive Messages
    d. HATEOAS

Challenge Definition

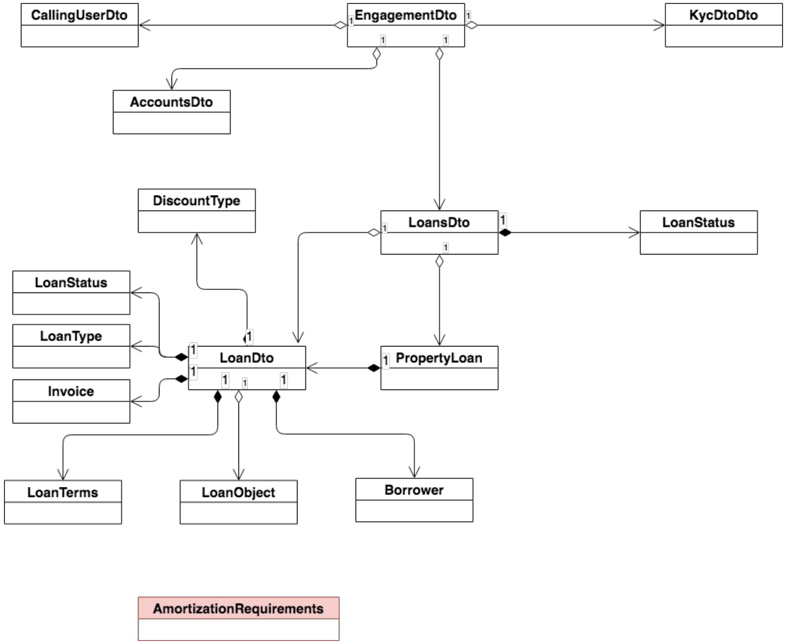
We need to provide → Amortization Requirements

Different Views of the “System”

- Domain Data
- Complexity
- Network (performance)
- REST API Granularity
- Error Handling
- Present VS Future
- User Experience

Challenge Definition: Domain Data

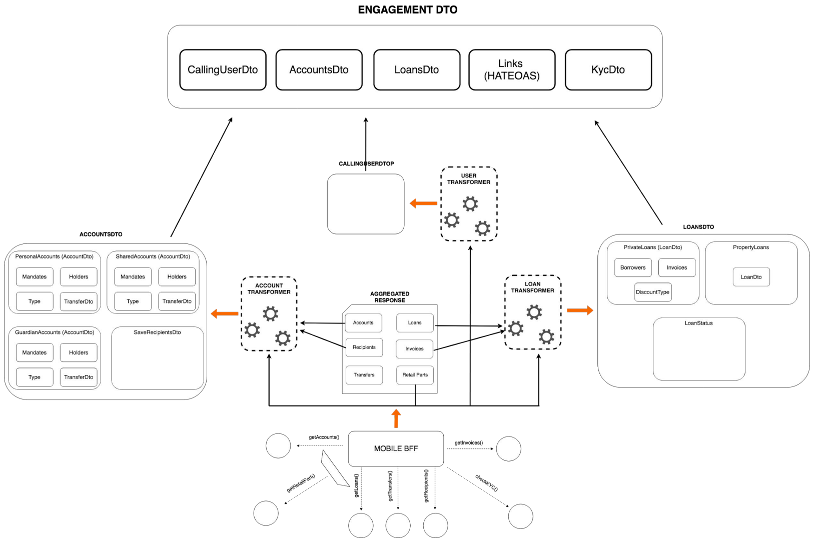
Where should we place the data?



Challenge Definition: Complexity

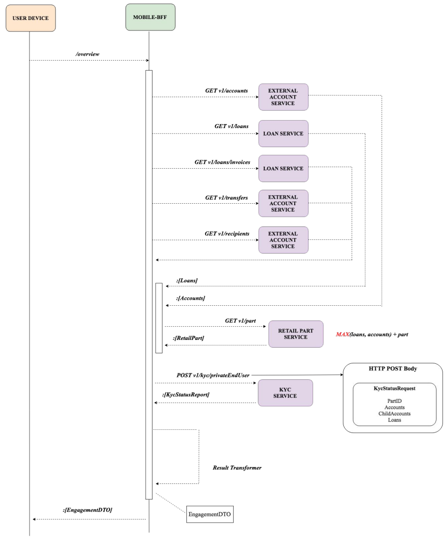
Client Complexity?

Server Complexity?



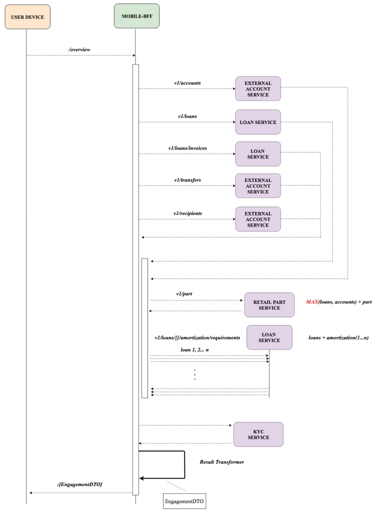
Challenge Definition: Network

ENGAGEMENT DIAGRAM



MAX((MAX(loans, accounts) + part) , transfers, invoices, recipients) + kyc

ENGAGEMENT DIAGRAM



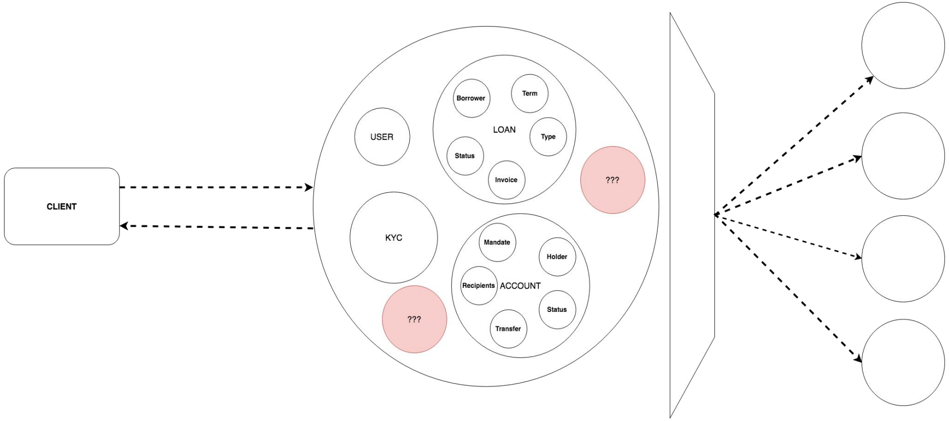
MAX((MAX(loans, accounts) + part) , transfers, invoices, recipients) + kyc

⇩

MAX(MAX((MAX(loans, accounts) + part), (loans+amortization(1...n))) , transfers, invoices, recipients) + kyc

Challenge Definition: REST Granularity

Fine-grained VS Coarse-grained

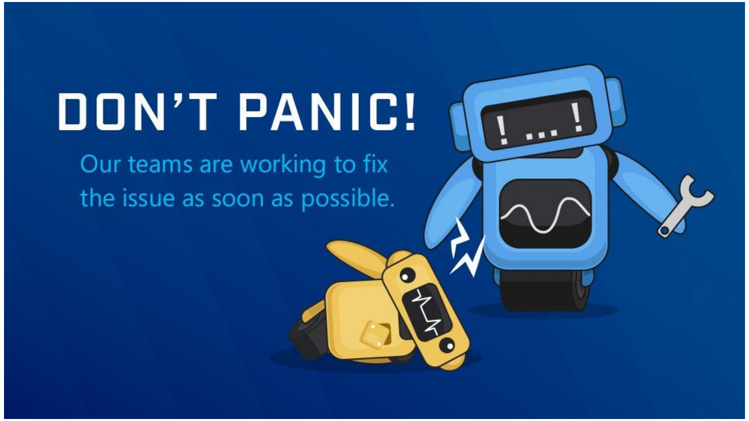


Challenge Definition: Error Handling

What if partial State error happens?

Should we work with partial State?

Probability of success?



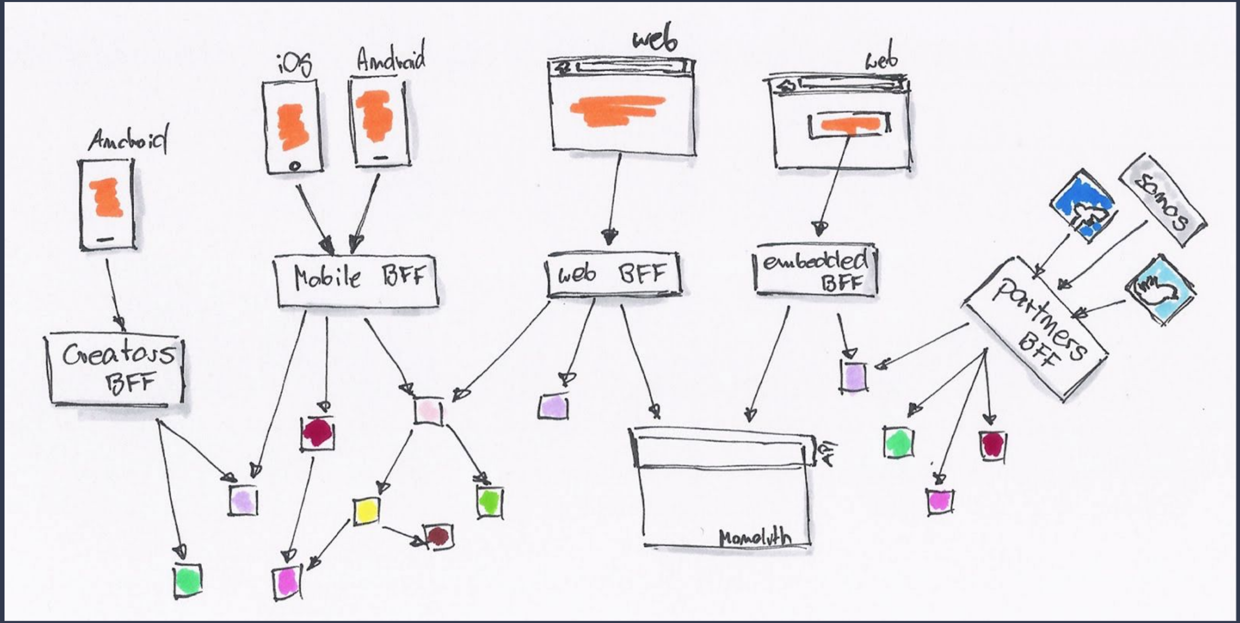
Challenge Definition: Present VS Future

Is it “just” a good decision for the current app state?

What features are we going to implement?

How this design decision can affect near future?

Questions, Reflections, Ideas?


Android
Creators BFF
iOS
Android
Mobile BFF
web
web BFF
web
embedded BFF
Sonos
partners BFF
API
Monolith