Netflix Play API

Why we built an Evolutionary Architecture

Suudhan Rangarajan (@suudhan)
Senior Software Engineer
#netflixeverywhere

Download & Go

Look for this symbol to download movies and TV episodes to watch on the go without using data.

FIND SOMETHING TO DOWNLOAD

OK

Q1 2016  Q2 2016  Q3 2016  Q4 2016
Increase in Rollbacks

Decrease in deployments

Rollbacks per month
Dependments per week
Netflix Play API
Why we built an Evolutionary Architecture

Suudhan Rangarajan (@suudhan)
Senior Software Engineer
Previous Architecture Workflow

API Proxy Service

API Service

Sign-up
Content Discovery
Playback

Devices

← Services hosted in AWS →

Domain specific Microservices
Content Discovery Workflow

- Devices
- Services hosted in AWS

API Proxy Service → Discovery API → Content Discovery → Playback

Sign-up

Domain specific Microservices
Playback Workflow

 Devices

 ← Services hosted in AWS →

 API Proxy Service

 API Service

 Sign-up
 Content Discovery
 Playback

 Domain specific Microservices
Identity

Type 1/2 Decisions

Evolvability
Start with WHY: Ask why your service exists
Lead the Internet TV revolution to entertain billions of people across the world

Maximize customer engagement from signup to streaming

Enable acquisition, discovery, playback functionality 24/7
API Identity: Deliver Acquisition, Discovery and Playback functions with high availability
Single Responsibility Principle: Be wary of multiple-identities rolled up into a single service
### Previous Architecture

- Signup API
- Discovery API
- Play API

**One API Service**

### Current Architecture

- Signup API
- Discovery API
- Play API

**API Service Per function**
Lead the Internet TV revolution to entertain billions of people across the world

Maximize user engagement of Netflix customer from signup to streaming

Enable non-member, discovery, playback functionality 24/7

Deliver Playback Lifecycle 24/7
Decide best playback experience

Authorize playback experience

Track events to measure playback experience

Devices

API Proxy Service

Play API
Decide best playback experience

Authorize playback experience

Track events to measure playback experience

High Coupling, Low Evolvability

Devices

API Proxy Service
Let's all give it up for Monolith, who is about to retire. I'd also like to take this opportunity to proudly introduce...

THE MICROSERVICES

CLAP CLAP CLAP
CLAP CLAP CLAP CLAP
CLAP YEAE CLAP YEAE YEAE

Gimme five, buddy.

Talk to my API gateway please...

I'm not sharing a database with you!

Well, they might need some ORCHESTRATION first.

Daniel Stori {turnoff.us}
Play API Identity: Orchestrate Playback Lifecycle with stable abstractions
Guiding Principle: We believe in a simple singular identity for our services. The identity relates to and complements the identities of the company, organization, team and its peer services.
Identity

Type 1/2 Decisions

Evolvability
“Some decisions are consequential and irreversible or nearly irreversible – one-way doors – and these decisions must be made methodically, carefully, slowly, with great deliberation and consultation [...] We can call these Type 1 decisions…”

Quote from Jeff Bezos
“...But most decisions aren’t like that – they are changeable, reversible – they’re two-way doors. If you’ve made a suboptimal Type 2 decision, you don’t have to live with the consequences for that long [...] Type 2 decisions can and should be made quickly by high judgment individuals or small groups.”

Quote from Jeff Bezos
Three Type 1 Decisions to Consider

Appropriate Coupling
Synchronous & Asynchronous
Data Architecture
Two types of Shared Libraries

- **Shared Libraries with common functions**
  - Utilities
  - cache
  - Metrics

- **Client Libraries used for inter-service communications**
  - Client 1
  - Client 2
  - Client 3
1) Binary Coupling

“Thick” shared libraries with 100s of dependent libraries (e.g. utilities jar)
Binary coupling => Distributed Monolith

Hundreds of shared libraries spanning services across network boundaries

Previous Architecture
“The evils of too much coupling between services are far worse than the problems caused by code duplication”

- Sam Newman (Building Microservices)
Play API Service

Playback Decision Client

Playback Decision Service

Previous Architecture
Clients with heavy Fallbacks

Requests Per Second of API Service

Increase in Latencies from the API Service

Execution of Fallback via Play Decision Client
2) Operational Coupling

Previous Architecture

Play API Service

Playback Decision Client

Playback Decision Service
“Operational Coupling” might be an ok choice, if some services/teams are not yet ready to own and operate a highly available service.
Many of the client libraries had the potential to bring down the API Service.
3) Language Coupling

Play API Service

client

Playback Decisions Service

Java

Previous Architecture
Communication Protocol

**Play API Service**

- Client

**Jersey Framework**

- Playback Decisions Service

REST over HTTP 1.1
- Unidirectional (Request/Response type APIs)

*Previous Architecture*
## Requirements

<table>
<thead>
<tr>
<th>Operationally “thin” Clients</th>
<th>No or limited shared libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto-generated clients for Polyglot support</td>
<td>Bi-Directional Communication</td>
</tr>
</tbody>
</table>
REST vs RPC

- At Netflix, most use-cases were modelled as Request/Response
  - REST was a simple and easy way of communicating between services; so choice of REST was more incidental rather than intentional
- Most of the services were not following RESTful principles.
  - The URL didn’t represent a unique resource, instead the parameters passed in the call determined the response - effectively made them a RPC call
- So we were agnostic to REST vs RPC as long as it meets our requirements
### Previous Architecture

- **Play API Service**
  - Playback Decisions
  - Playback Authorize
  - Playback Events

1. Operationally Coupled Clients
2. High Binary Coupling
3. Only Java
4. Unidirectional communication

### Current Architecture

- **Play API Service**
  - Playback Decisions
  - Playback Authorize
  - Playback Events

1. Minimal Operational Coupling
2. Limited Binary Coupling
3. Beyond Java
4. Beyond Request/Response
Type 1 Decision: Appropriate Coupling

Consider “thin” auto-generated clients with bi-directional communication and minimize code reuse across service boundaries.
Three Type 1 Decisions to Consider

- Appropriate Coupling
- Synchronous vs Asynchronous
- Data Architecture
PlayData getPlayData(string customerId, string titleId, string deviceId) {
    CustomerInfo custInfo = getCustomerInfo(customerId);
    DeviceInfo deviceInfo = getDeviceInfo(deviceId);
    PlayData playdata = decidePlayData(custInfo, deviceInfo, titleId);
    return playdata;
}
Typical Synchronous Architecture

Request Handler
Thread pool

Client Thread pool
Typical Synchronous Architecture

Request Handler
- Thread pool
- getPlayData
- getCustomerInfo
- getDeviceInfo
- decidePlayData
- Return

Client Thread pool

One thread per request

Customer Service
Device Service
Play Data Decision Service
Typical Synchronous Architecture

- Request Handler
- Thread pool
- Client Thread pool
- getPlayData
- getCustomerInfo
- getDeviceInfo
- decidePlayData

One thread per request

Blocking Request Handler

Blocking Client I/O
Typical Synchronous Architecture

- One thread per request
- Blocking Request Handler
- Blocking Client I/O

- Request Handler Thread pool
- Client Thread pool

Functions:
- getPlayData
- getCustomerInfo
- getDeviceInfo
- decidePlayData

Works for Simple Request/Response
Works for Limited Clients
Beyond Request/Response

One Request - One Response

Request Play-data for Title X
Receive Play-data for Title X

One Request - Stream Response

Request Play-data for Titles X,Y,Z
Receive Play-data for Title X
Receive Play-data for Title Y
Receive Play-data for Title Z

Stream Request - One Response

Request Play-data for Title X
Request Play-data for Title Y
Request Play-data for Title Z
Receive Play-data for Titles X,Y,Z

Stream Request - Stream Response

Request Play-data for Title X
Request Play-data for Title Y
Get Play-data for Title Z
Receive Play-data for Title Y
Receive Play-data for Title Z
Asynchronous Architecture

- Request/Response Event Loop
- Worker Threads
- Outgoing Event Loop per client
PlayData getPlayData(string customerId, string titleId, string deviceId){
    Zip(getCustomerInfo(customerId),
        getDeviceInfo(deviceId),
        (custInfo, deviceInfo) ->
            return decidePlayData(custInfo, deviceInfo, titleId)
    );
}
Asynchronous Architecture

1. setup

Request/Response Event Loop
Workflow spans many worker threads

Outgoing Event Loop per client

Customer Service
Device Service
PlayData Service
Asynchronous Architecture

Request/Response
Event Loop

Workflow spans many worker threads

Outgoing Event Loop per client

2 getCustomerInfo

Customer Service

Device Service

PlayData Service
Asynchronous Architecture

Request/Response
Event Loop

Outgoing Event Loop
per client

Workflow spans many
worker threads

getDeviceInfo

Customer Service

Device Service

PlayData Service
Asynchronous Architecture

Request/Response Event Loop

Workflow spans many worker threads

Outgoing Event Loop per client

Customer Service

Device Service

PlayData Service
Asynchronous Architecture

Request/Response Event Loop

Workflow spans many worker threads

decidePlayData

Outgoing Event Loop per client

Customer Service

Device Service

PlayData Service
Workflow spans multiple threads

- All context is passed as messages from one processing unit to another.
- If we need to follow and reason about a request, we need to build tools to capture and reassemble the order of execution units.
- None of the calls can block.
Asynchronous Architecture

Request/Response Event Loop

Worker Threads

Outgoing Event Loop per client

Asynchronous Request Handler

Non-Blocking I/O
Synchrony

Ask: Do you really have a need beyond Request/Response?
Network Event Loop

Outgoing Event Loop per client

Dedicated thread

Synchronous Execution + Asynchronous I/O

getPlayData
getCustomerInfo
deviceInfo
decidePlayData
Return

Blocking Request Handler

Non-Blocking I/O

Current Architecture
Type 1 Decision: Synchronous vs Asynchronous

If most of your APIs fit the Request/Response pattern, consider a synchronous request handler, with nonblocking I/O
Three Type 1 Decisions to Consider

- Appropriate Coupling
- Synchronous vs Asynchronous
- Data Architecture
Without an intentional Data Architecture, Data becomes its own monolith
What a Data Monolith looks like

Service 1

Data Source

Service 2

Data Source

Service 3

Data Source

Service 4
What a Data Monolith looks like

API Service

← Multiple Data sources loaded in memory →

Memory Load

↑

4 GB
1 GB
2 GB
400 MB
600 MB
What a Data Monolith looks like

Very small percentage of data actually accessed

Previous Architecture
What a Data Monolith looks like

API Service

Each Data Source models gets coupled across classes and libraries
What a Data Monolith looks like

API Service

Data Update

CPU Utilization

Unpredictable Performance Characteristics

Previous Architecture
What a Data Monolith looks like

API Service

Netflix was down

Data Update

Potential to bring down the service

Previous Architecture
"All problems in computer science can be solved by another level of indirection."

David Wheeler
(World’s first Comp Sci PhD)
Current Architecture

Data Source

Data Service

Data Store

Data Loader

Materialized View

Data Source

Data Source

Data Source

Data Source
Current Architecture

- Uses only the data it needs
- Predictable Operational Characteristics
- Reduced Dependency chain
Type 1 Decision: Data Architecture

Isolate Data from the Service. At the very least, ensure that data sources are accessed via a layer of abstraction, so that it leaves room for extension later.
Three Type 1 Decisions to Consider

- Appropriate Coupling
- Synchrony
- Data Architecture
For Type 2 decisions, choose a path, experiment and iterate
Guiding Principle: Identify your Type 1 and Type 2 decisions; Spend 80% of your time debating and aligning on Type 1 Decisions.
An Evolutionary Architecture supports **guided** and incremental change as first principle among multiple dimensions

- ThoughtWorks
Choosing a **microservices** architecture with **appropriate coupling** allows us to evolve across multiple dimensions
How evolvable are the Type 1 decisions

<table>
<thead>
<tr>
<th>Change Play API</th>
<th>Previous Architecture</th>
<th>Current Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Polyglot services?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bidirectional APIs?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Additional Data Sources?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Known
Unknowns
Potential Type 1 decisions in the future?

<table>
<thead>
<tr>
<th>Change Play API</th>
<th>Previous Architecture</th>
<th>Current Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Containers?</td>
<td></td>
<td>?</td>
</tr>
<tr>
<td>Serverless?</td>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>

And we fully expect that there will be Unknown Unknowns
As we evolve, how to ensure we are not breaking our original goals?
Use **Fitness Functions** to guide change
High Availability  Low Latency

Evolvability  Simplicity

Scalable  Reliability

Continuous Integration  High Throughput

Observability  Developer Productivity
Why **Simplicity** over Reliability?

- Increase in Operational Complexity
- Reliable Fallback when service is down
Why **Scalability** over Throughput?

New instances were added

Increase in Errors due to cache warming
Why Observability over Latency?

Decrease in latency by using a fully async executor

Cost of Async: Loss in Observability
Four 9s of availability

- Merge to Deploy Time
- Resilience to failures
- P99 latency
- Thin Clients
Guiding Principle: Define Fitness functions to act as your guide for architectural evolution
**Previous Architecture**

- Multiple Identities
- Operational Coupling
- Binary Coupling
- Synchronous communication
- Only Java
- Data Monolith

**Current Architecture**

- Singular Identities
- Operational Isolation
- No Binary Coupling
- Asynchronous communication
- Beyond Java
- Explicit Data Architecture
- Guided Fitness Functions
- No incidents in a year
- 4.5 deployments per week
- Just two rollbacks!
Build a Evolutionary Architecture

- Identity
- Type 1/2 Decisions
- Evolvability