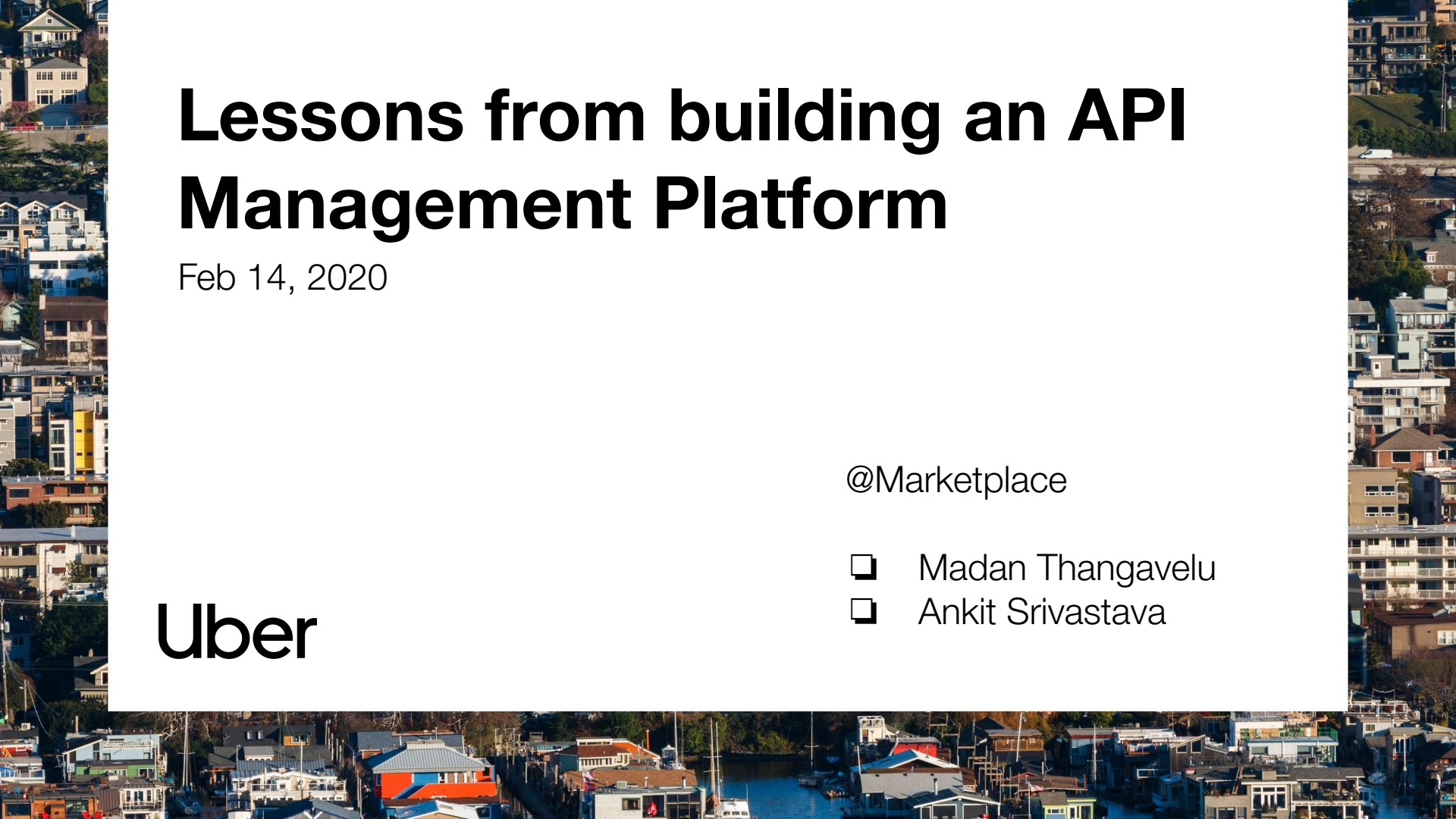


Lessons from building an API Management Platform

Feb 14, 2020

@Marketplace

- Madan Thangavelu
- Ankit Srivastava

Uber

Madan Thangavelu

Senior Manager. I head the Fulfillment Platform group at Uber, and previously been responsible for building the API Gateway Platform. Come talk to me about the future of Fulfillment at Uber.

Recovering engineer, nodeJS enthusiast, hacker.

Ankit Srivastava

Staff Engineer. Work on the Fulfillment Platform group at Uber, and focussing on development of a new age fulfillment system for unlocking new potentials.

Distributed Systems enthusiast.

Agenda

01 API Gateway Features
02 Self Service API Gateway
03 Principles
04 Making Choices
05 Learnings

20+ Apps

1600+ APIs

3000+ services

API Gateway Features

Security

CORS
Data validation
Rate limit policies
Client access policies
Data encryption
Access Logging
Token Authentication

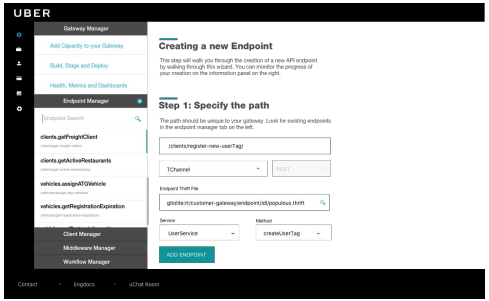
Operations

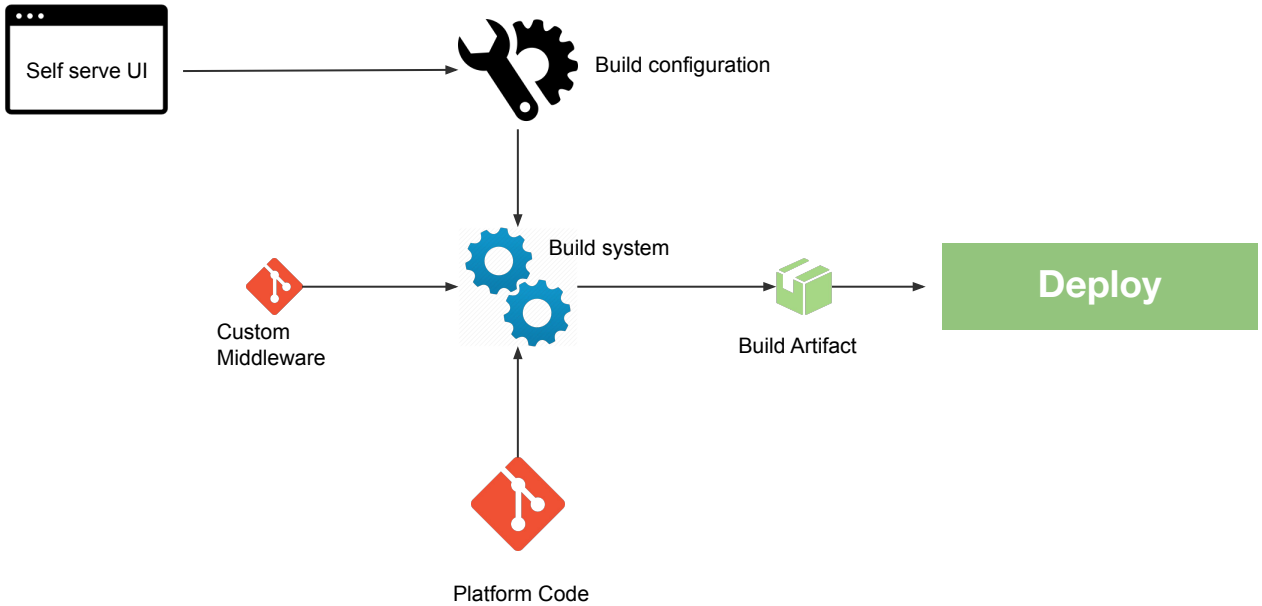
Tracing
Debugging knobs
Debugging tools
Global Rate Limiting
Local Rate Limiting
Isolation
Auto alerts
Monitoring & SLA

API Management

API creation/modification
Request/Response Tx
Protocol transformations
Versioning
Migrations
SDK generations
Document generation
DC Redirection
Request Shadowing

Self Service API Gateway





Principles

Reliability

- API isolation - one does not affect another
- Performance, Tooling and Documentation are first class citizens
- Bad code is prevented with automated processes

Build Artifact

- A build should be reproducible and completely deterministic
- Deploy artifact should be 100% generated, no custom code.
- Local development possible

Features

- Less magical features preferred over feature bloat
- Easy to switch from magical automation to full control

Making choices

[1] Protocol

HTTP <> HTTP
HTTP <> Tchannel
HTTP <> gRPC
gRPC <> gRPC
gRPC <> HTTP

[2] Schema

OpenAPI
Thrift
Proto
YAML

[3] Serialization

JSON <> JSON
JSON <> Thrift
Proto <> Thrift
Proto <> JSON

[4] Customization

Incoming Headers
Incoming Body
Response Headers
Response Body

[5] Language

GoLang
NodeJs
Java

[6] Config Mgmt

RDBMS
GIT

[7] Self-Serve

Developer Experience
Rich UI
YAML / JSON based
config

[8] Scaling

Single binary
Multiple binary

[9] Migration

Move all 1600 APIs
Allow only new APIs

[10] Features

Uber

API Gateway Platform

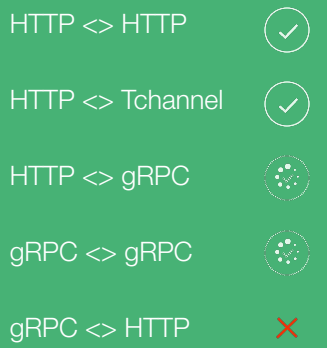
[1] Hybrid Protocols

Pros

- Protocol transformation delinks development of mobile to server communication from internal Uber service implementation.
- Rich gateway compatible with 99% of Uber protocols

Cons

- No application headers in Tchannel & gRPC
- No notion of REST verbs in gRPC
- Error conversion from one protocol to another
- HTTP -> gRPC transcends free form to typed requests
- gRPC Stream incompatible with other protocols
- The overhead to know the different compatibilities during config



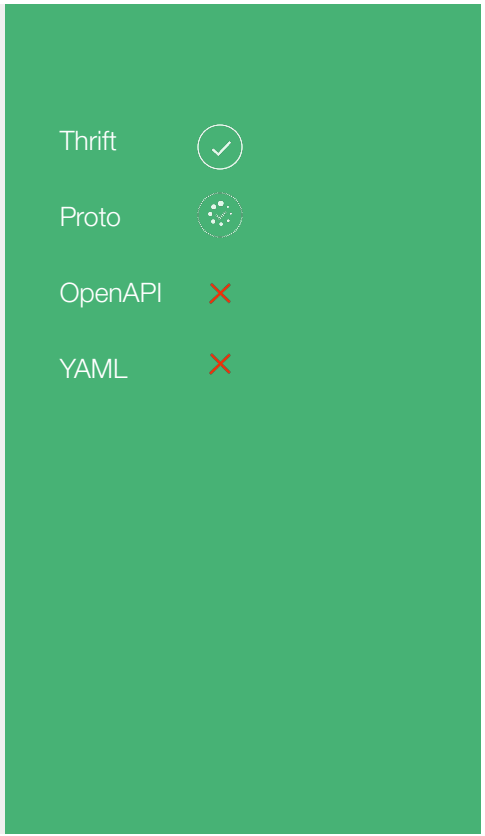
[2] Schema Management

Pros

- Strict Typing - a single field cannot be number & string like OpenAPI
- Expressive
- Annotation support to allow building complex representations

Cons

- Fundamental mismatch between schemas across protocols
  - Union, i64, i32 in thrift to JSON/HTTP
- No notion to represent Headers, Query, Path parameters
- What is ok in Thrift is not ok in GoLang
- Proto only services now needed a thrift file too (ouch)
- No support for meta validation like OpenAPI
- No doc generation support
- Nested imports are hard to resolve & maintain green build



[3] Serialization Interoperability

Pros

- Rich & versatile support for multiple serializations, developers do not need to be aware of on-the-wire serializations.
- Proto (binary) ingress reduces payload size
- Migrate old APIs without migration

Cons

- Fundamental incompatibility in representations
  - Union/Set/List/Map Thrift representation in JSON & GoLang
  - UInt32Value, UInt64Value, Timestamp, Int32Value, in JSON?
- Payload Versioning
- Debug tooling for proto

JSON <> JSON ✓

JSON <> Thrift ✓

JSON <> Proto ✓

Proto <> Thrift ✓

Proto <> JSON ✓

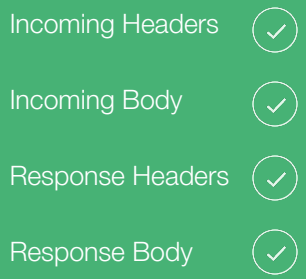
[4] API Customization

Pros

- Rich support built on header/payload interception
- Support for payload / metadata transformation, cross-protocol mappings
  - Header to Body
  - Error mapping
- Business agnostic feature rich custom middlewares
  - Filter fields, localization, rate limiting based on request

Cons

- Cost of serialization/deserialization
- Numerous bugs in mapping logic (e.g., nested list of map or sets)
- Magical auto-mapping based on field name resulted in bugs



[5] Language

Pros

- Dynamic (node) → Static (golang) language, strict typing
- Significantly performant ~ 1000 QPS/Core
- Aligned with company infrastructure

Cons

- A dynamic environment based on typed language
- No support of generics leads to too much codegen
- Large binary - compile time & hard failures
- Language naming conventions & reserved keywords
- Goimport non-determinism

[6] Config Management

Pros

- Clear version history & rollback
- Federation of config is straightforward
- Generated code & config live together
- Ability to test generated binary before config is committed

Cons

- Merge conflicts
- Slow to git commit and push (numerous systems)
- Hacky GIT binding with no native control from GoLang
- Slow checkout to apply a new config
- Polluted review of config with generated code

GIT 

RDBMs 

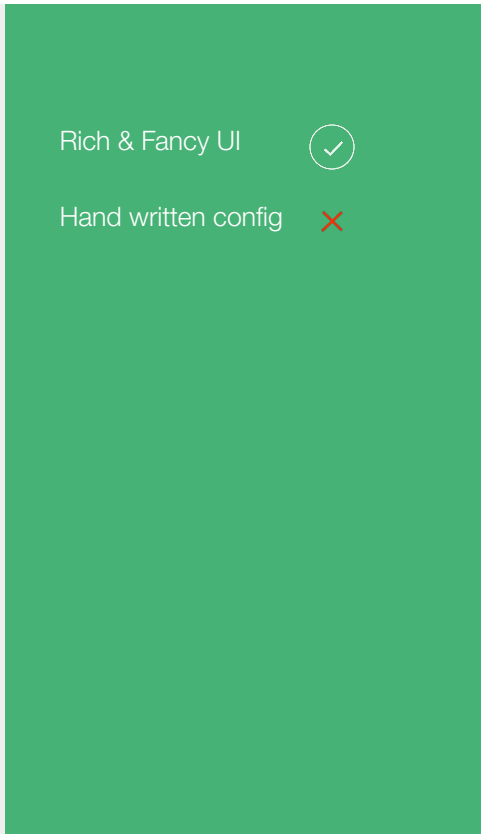
[7] Self-Serve

Pros

- Guides the user across tons of configurations & validations
- Fancy* & magical
- Less option for hacking features from customers (engineers)
- Feature rich - alerts, search fields, api docs, management
- Provided sessions for each user - shareable error states

Cons

- Batch/Multi edit flows became complex via the UI
- Surfacing complex errors was hard and at the mercy of the UI
- Dedicated frontend eng
- Introduced RDBMs for UI states & sessions



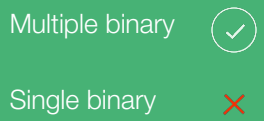
[8] Scaling

Pros

- Multi-binary support to scale 1000s of APIs across binaries, provides isolation & independent scaling.
- Horizontally shard API deployments based on a url routing proxy layer.
- Faster isolated deploy for in development API binary

Cons

- Significant config overhead to resolve module dependencies
- Maintenance of multiple binaries



[9] Migration

Pros

- Deprecate the old API gateway with no maintenance overhead
- Significantly performant API gateway
- Validation that the new API gateway is feature rich
- Security & latest fixes
- An opportunity to deprecate

Cons

- 1.5 years of migration support & company wide effort
- Forced to support features we did not want in the new gateway

Move all 1600 APIs ✓

Allow only new APIs ✓

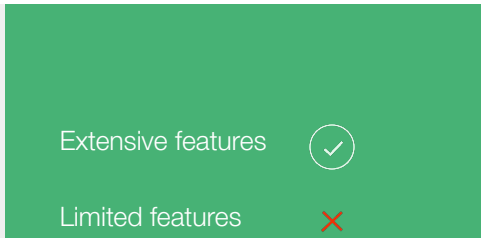
[10] Centralized Feature Support

Pros

- Rich Feature support for customers to choose from
  - Rate limit by body/query
  - Decrypt request
  - Encrypt response
  - Transform Request
  - Transform Response
  - Load shedding
  - Localization
  - Authentication per API call
  - Streaming APIs
  - UI Features

Cons

- Development & maintenance overhead



Making choices

[1] Protocol

HTTP <> HTTP
HTTP <> Tchannel
HTTP <> gRPC
gRPC <> gRPC
gRPC <> HTTP

[2] Schema

OpenAPI
Thrift
Proto
YAML

[3] Serialization

JSON <> JSON
JSON <> Thrift
Proto <> Thrift
Proto <> JSON

[4] Customization

Incoming Headers
Incoming Body
Response Headers
Response Body

[5] Language

GoLang
NodeJs
Java

[6] Config Mgmt

RDBMS
GIT

[7] Self-Serve

Developer Experience
Rich UI
YAML / JSON based config

[8] Scaling

Single binary
Multiple binary

[9] Migration

Move all 1600 APIs
Allow only new APIs

[10] Features

Uber

API Gateway Platform

Learnings

- Stick with a single protocol across the company
- Stick with a single serialization across the company
- Decision to serialize/deserialize is not for the faint hearts
- UI can significantly assist users but add lot of maintenance overhead
- Developers love faster iteration - choose the tech stack wisely
- GIT for configuration is a double edged sword
- There is no perfect programming language to build a API gateway system
- Scale :

1.5k

endpoints

1M rps

requests to the API layer

Thank you!

- Madan Thangavelu
  @hackinghabits
- Ankit Srivastava
  @_ankits