goto; chicago

Adopting gRPC

Overcoming Team and Technical Hurdles

Joshua Humphries
Software Engineer at FullStory

jhump on GitHub

Adopting gRPC: Agenda

- Introduction to gRPC (“What?”)
- Benefits & Tradeoffs (“Why?”)
- Development Flow & Hurdles (“How?”)
- Overcoming Hurdles, Part 1: Best Practices
- Overcoming Hurdles, Part 2: Tools

Adopting gRPC: Agenda

- Introduction to gRPC (“What?”)
- Benefits & Tradeoffs (“Why?”)
- Development Flow & Hurdles (“How?”)
- Overcoming Hurdles, Part 1: Best Practices
- Overcoming Hurdles, Part 2: Tools

Intro: What is gRPC?

Ten thousand foot view: gRPC is

1. Specification for Remote Procedure Calls
2. Code generation tools
    - compile ***Protocol Buffers*** into programming language of your choice
3. Runtime libraries for programming language of your choice

Intro: What is gRPC?

Wait, what are **Protocol Buffers**?

1. An IDL (Interface Definition Language)
2. Code generation tools
    - The protobuf compiler, `protoc`
3. Runtime libraries for programming language of your choice

AKA *“protobufs”, “protos” for short*

Intro: What is gRPC?

Close up view: **Specification for RPCs**

- Language-agnostic Semantics
  - Unary RPCs vs. Streaming RPCs
  - Metadata
  - Cancellation/Deadlines
  - Response messages and Errors
- Spec for wire protocol
  - Maps RPC semantics to HTTP/2 protocol

Intro: What is gRPC?

Close up view: **Code Generation Tools**

- `protoc`
  - Generates data structures that correspond to IDL data model
  - Request and response types
- Plugins for `protoc`
  - Generate interfaces and client stubs

Intro: What is gRPC?

Close up view: **Runtime Libraries**

- Implements Wire Protocol
- Server Support
  - Expose service implementations via gRPC
- Client Support
  - Machinery for connecting to servers, sending RPCs
  - Service discovery, load balancing, connection management

Intro: What is gRPC?

Want to know more?

1. gRPC Home Page:
   https://grpc.io/
2. Protocol Buffers Home Page:
   https://developers.google.com/protocol-buffers/
3. Online book: “Practical gRPC”
   https://bleedingedgepress.com/practical-grpc/

Adopting gRPC: Agenda

- Introduction to gRPC (“What?”)
- Benefits & Tradeoffs (“Why?”)
- Development Flow & Hurdles (“How?”)
- Overcoming Hurdles, Part 1: Best Practices
- Overcoming Hurdles, Part 2: Tools

Benefits & Tradeoffs of gRPC

1. Developer productivity
    - Abstracts away networking details
    - “Procedure Call” syntax
    - Action-centric, not resource-centric

Benefits & Tradeoffs of gRPC

2. Strongly Typed Message Schemas
    - Clear “contract” of what messages can look like
    - Compile-time type checking (in languages that allow it)
    - Object-oriented APIs facilitate IDE support

    - *Tradeoff*: less flexibility

Benefits & Tradeoffs of gRPC

3. Efficiency/Performance
    - HTTP/2
    - Compact binary format

    - *Tradeoff*: less human-consumable than JSON+REST
    - *Tradeoff*: lack of browser support

Benefits & Tradeoffs of gRPC

4. Language agnostic
    - Official libraries support many languages
        - C++, Java, Go, JavaScript, Python, Ruby, Objective-C, C#, PHP, Dart
    - Open-source libraries bring support to many others!
    - Interoperability across heterogenous clients and server

Benefits & Tradeoffs of gRPC

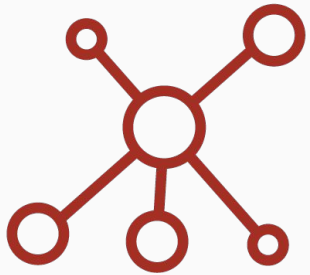
5. Many modern features
    - Flow control
    - Full-duplex bidi streams; not just request-response!
    - Request and response metadata (headers, trailers)
    - Call cancellation, deadline propagation
    - Interceptors (middleware) *
    - Service discovery and load balancing *
    - Automatic retries, hedging *

Benefits & Tradeoffs of gRPC

6. Opinionated
    - gRPC is end-to-end service comms framework
        - IDL for modeling data and interfaces (protobuf)
        - Prescriptive about the shape of client stubs and semantics
        - Transport layer (HTTP/2 + protobuf)
        - Server and client library
    - Few decisions to make
    - *Tradeoff*: must buy in 100% *

Benefits & Tradeoffs of gRPC

7. Ecosystem
    - Many tools and extensions: “Awesome gRPC”
    - grpc-gateway
        - Exposes gRPC services as REST APIs
    - grpc-web
        - Adapts gRPC to work with browser clients

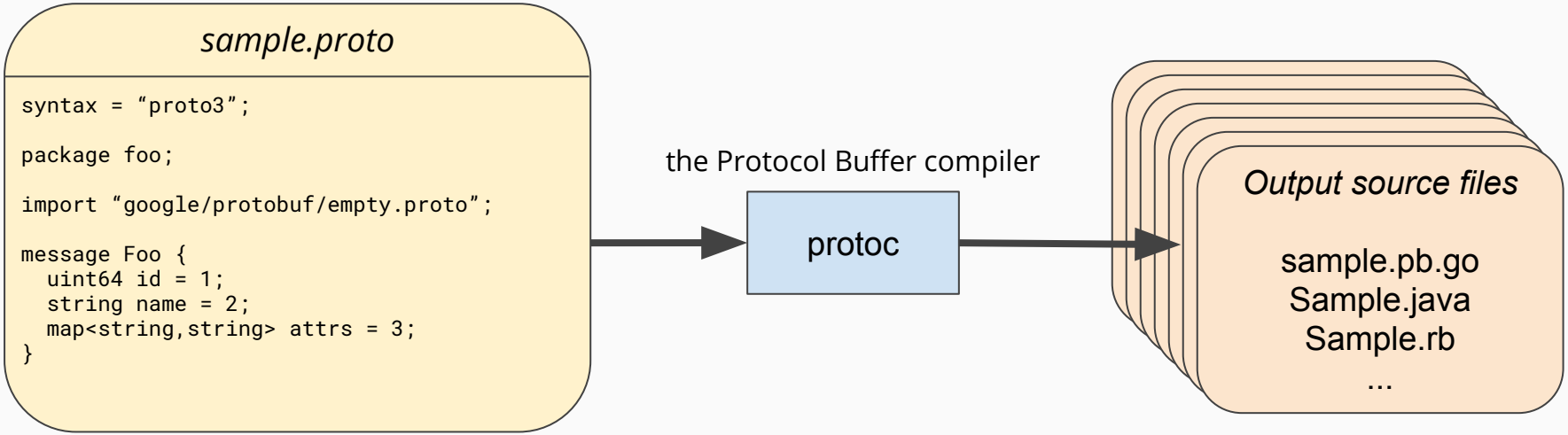


Adopting gRPC: Agenda

- Introduction to gRPC (“What?”)
- Benefits & Tradeoffs (“Why?”)
- Development Flow & Hurdles (“How?”)
- Overcoming Hurdles, Part 1: Best Practices
- Overcoming Hurdles, Part 2: Tools

Development Flow, Hurdles

It's all about code generation...



Development Flow, Hurdles

1. Define the API in “language agnostic” proto sources

foo/model.proto

```
syntax = “proto3”;

package foo;

message Foo {
  uint64 id = 1;
  string name = 2;
  map<string,string> attrs = 3;
}
```

foo/service.proto

```
syntax = “proto3”;

package foo;

import “foo/types.proto”;
import “google/protobuf/empty.proto”;

service FooService {
  rpc Query (QueryRequest) returns
      (stream QueryResponse);
  rpc Update (UpdateRequest) returns
      (UpdateResponse);
  rpc Tickle (TickleRequest) returns
      (google.protobuf.Empty);
}
```

foo/types.proto

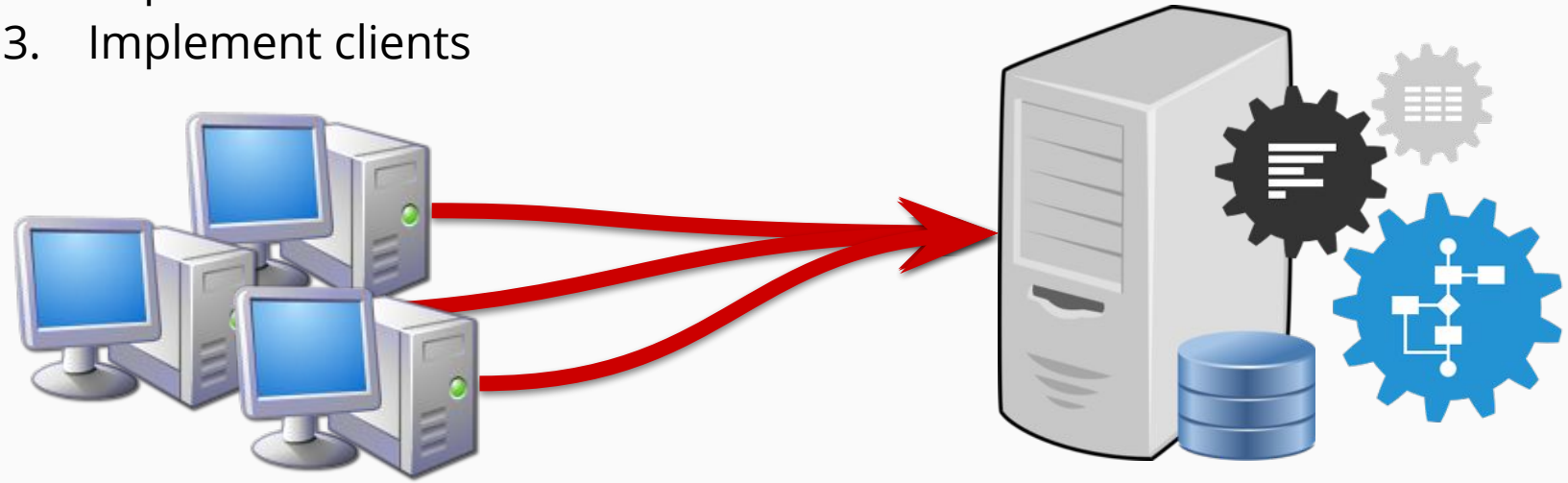
```
syntax = “proto3”;

package foo;

import “foo/model.proto”;

message QueryRequest {
  oneof criteria {
    uint64 by_id = 1;
    string by_name = 2;
    AttrCriteria by_attr = 3;
  }
}
```

Development Flow, Hurdles

1. Define the API in “language agnostic” proto sources
2. Implement server

Development Flow, Hurdles

1. Define the API in “language agnostic” proto sources
2. Implement server
3. Implement clients



Development Flow, Hurdles

1. Define the API in “language agnostic” proto sources
2. Implement server
3. Implement clients

Easy, right?

Development Flow, Hurdles

What about...

- How should I organize proto sources?
- How do I share proto sources across clients and servers?
- How do I test/explore APIs?
- How do I change/evolve my APIs?
- How do I monitor/observe services?

Development Flow, Hurdles

Other Obstacles

- Learning curve: new toolchain, technologies
- Can't use familiar tools like WireShark, `curl`, `wget`, `postman`, etc.
- *Advanced:* Writing new tools

See Gophercon 2018 talk on `grpcurl`

Adopting gRPC: Agenda

- Introduction to gRPC (“What?”)
- Benefits & Tradeoffs (“Why?”)
- Development Flow & Hurdles (“How?”)
- Overcoming Hurdles, Part 1: Best Practices
- Overcoming Hurdles, Part 2: Tools
  - Providing a “gRPC skeleton”
  - Organizing proto sources
  - Evolving/versioning APIs

Adopting gRPC: Agenda

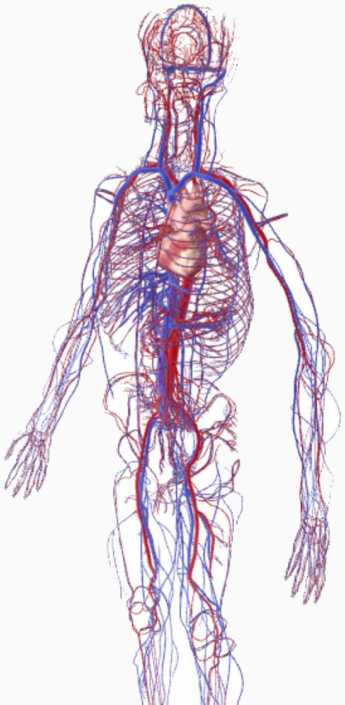
- Introduction to gRPC (“What?”)
- Benefits & Tradeoffs (“Why?”)
- Development Flow & Hurdles (“How?”)
- Overcoming Hurdles, Part 1: Best Practices
- Overcoming Hurdles, Part 2: Tools

  - Providing a “gRPC skeleton”
  - Organizing proto sources
  - Evolving/versioning APIs

Best Practices: gRPC Skeleton

Layers of a gRPC server|client

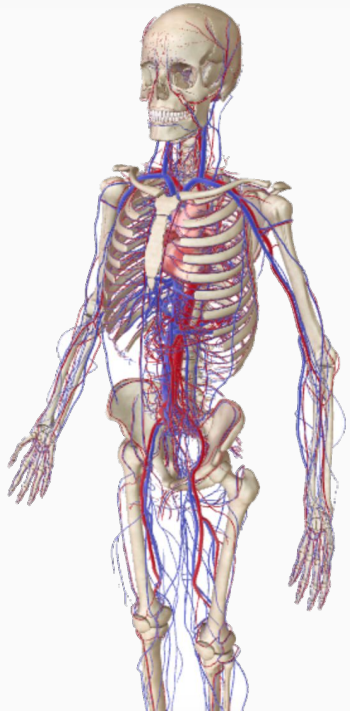
1. Internals: gRPC runtime library
   (nervous system)



Best Practices: gRPC Skeleton

Layers of a gRPC server|client

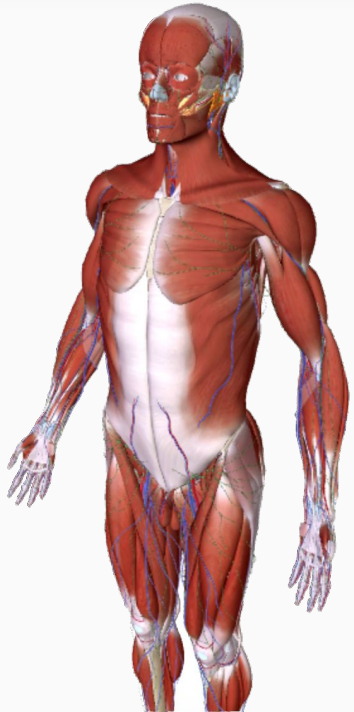
1. Internals: gRPC runtime library
   (nervous system)
2. Cross-cutting concerns: middleware
   (skeletal system)



Best Practices: gRPC Skeleton

Layers of a gRPC server|client

1. Internals: gRPC runtime library
   (nervous system)
2. Cross-cutting concerns: middleware
   (skeletal system)
3. Application code: business logic
   (muscular system)



Best Practices: gRPC Skeleton

Skeleton Specifics

- Wrapper around gRPC server|client library
- Automatically configures server|client in consistent way

Recommended:

Interceptor
***Server only*: Default services**
Server only: Maximum concurrent streams
Server only: Unknown method handler
Client only: Load balancing strategy
Client only: Service discovery strategy

Optional:

Flow control window/buffer sizes
TCP keep-alive policy
Default compression algorithm
Maximum message size
(recommended for receive)

Best Practices: gRPC Skeleton

```
// Client interceptor
func intercept(ctx context.Context, method string,
      req, reply interface{}, cc *grpc.ClientConn,
      invoker grpc.UnaryInvoker,
      opts ...grpc.CallOption) (err error) {
  // Metrics instrumentation
  start := time.Now()
  defer func() {
    duration := time.Since(start)
    code := status.FromError(err).Code()
    labels := map[string]string{
      "method": method,
      "code":   code.String(),
    }
    clientLatencyHistogram.Observe(labels, duration)
    if reqPb, ok := req.(proto.Message); ok {
      clientRequestSizes.Observe(labels,
          proto.Size(reqPb))
    }
    if respPb, ok := reply.(proto.Message);
       ok && err == nil {
      clientResponseSizes.Observe(labels,
          proto.Size(respPb))
    }
  }()

  // Tracing
  span, ctx := tracing.CreateClientSpan(ctx, method)
  defer span.Complete()

  // Authn|Authz
  credentials := auth.CredentialsFromContext(ctx)
  if credentials != nil {
    ctx = addCredsToMetadata(ctx)
  }

  // Optional: circuit-breaker
  brk := circuits.Get(ctx, cc, method)
  if brk != nil {
    return brk.Run(func() error {
      return invoker(ctx, method, req, reply,
          cc, opts...)
    })
  }

  return invoker(ctx, method, req, reply, cc, opts...)
}
```

Best Practices: gRPC Skeleton

```
// Server interceptor
func intercept(ctx context.Context, req interface{},
    info *grpc.UnaryServerInfo,
    handler grpc.UnaryHandler) (resp interface{},
        err error) {
  // Request ID (for structured logging)
  ctx = requests.ContextWithRequestID(ctx,
      uuid.GenerateUUID())
  method := info.FullMethod

  // Metrics instrumentation, Logging
  start := time.Now()
  defer func() {
    duration := time.Since(start)
    code := status.FromError(err).Code()
    labels := map[string]string{
      "method": method,
      "code":   code.String(),
    }
    serverLatencyHistogram.Observe(labels, duration)
    if reqPb, ok := req.(proto.Message); ok {
      serverRequestSz.Observe(labels, proto.Size(reqPb))
    }
    if respPb, ok := reply.(proto.Message);
        ok && err == nil {
      serverResponseSz.Observe(labels, proto.Size(respPb))
    }
    if err != nil {
      log.Warningf("error handling %s: %v", method, err)
    }
    log.Infof("grpc:%s code:%v %v %v",
        method, code, duration, peer.FromContext(ctx))
  }()

  // Tracing
  span, ctx := tracing.CreateServerSpan(ctx, method)
  defer span.Complete()

  // Authn|Authz
  requester, scopes := auth.GetRequester(ctx)
  if requester == nil {
    return status.Error(codes.Unauthenticated,
        "unknown requester")
  }
  if !auth.IsAllowed(scopes, method) {
    return status.Error(codes.PermissionDenied,
        "insufficient scopes")
  }

  // Optional: traffic control (server-only)
  limit := ratelimits.Get(ctx, requester, method)
  if limit != nil {
    if err := limit.Take(ctx, 1); err != nil {
      return status.Error(codes.ResourceExhausted,
          "too many requests")
    }
  }

  return handler(ctx, req)
}
```

Best Practices: gRPC Skeleton

Default services

- **Server Reflection!**
  - Makes it possible for tools to automatically discover your servers' RPC schemas
- Health checking
  - Provides entry point for asking the server which services are functioning
- Custom services
  - Diagnostic services for examining server internal state
    - Scheduled tasks/job queue, cluster coordination/leader election/leases, connected clients, outbound connections, circuit breaker stats, other metrics

Adopting gRPC: Agenda

- Introduction to gRPC (“What?”)
- Benefits & Tradeoffs (“Why?”)
- Development Flow & Hurdles (“How?”)
- Overcoming Hurdles, Part 1: Best Practices
  - Providing a “gRPC skeleton”
  - Organizing proto sources
  - Evolving/versioning APIs
- Overcoming Hurdles, Part 2: Tools

Best Practices: Code Organization

Q: How do I organize proto sources?

A: It depends on your repo organization and existing languages and toolchains you use.

Best Practices: Code Organization

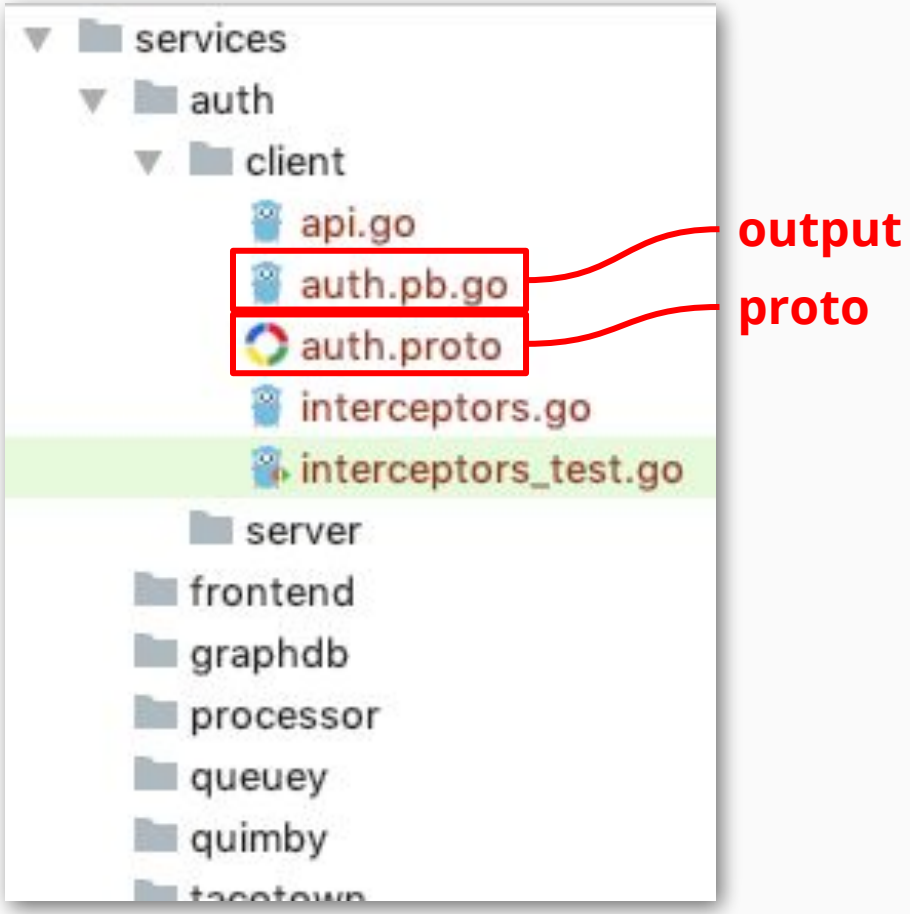
Mono-repo: Single repo with all code

- All libraries, all services
- Possibly 3rd-party/vendor'ed code
- Possibly polyglot

Monoglot? Keep proto files in same directory as generated output.
Polyglot? Create higher-level folder for proto sources, adjacent to other language-specific folder(s).

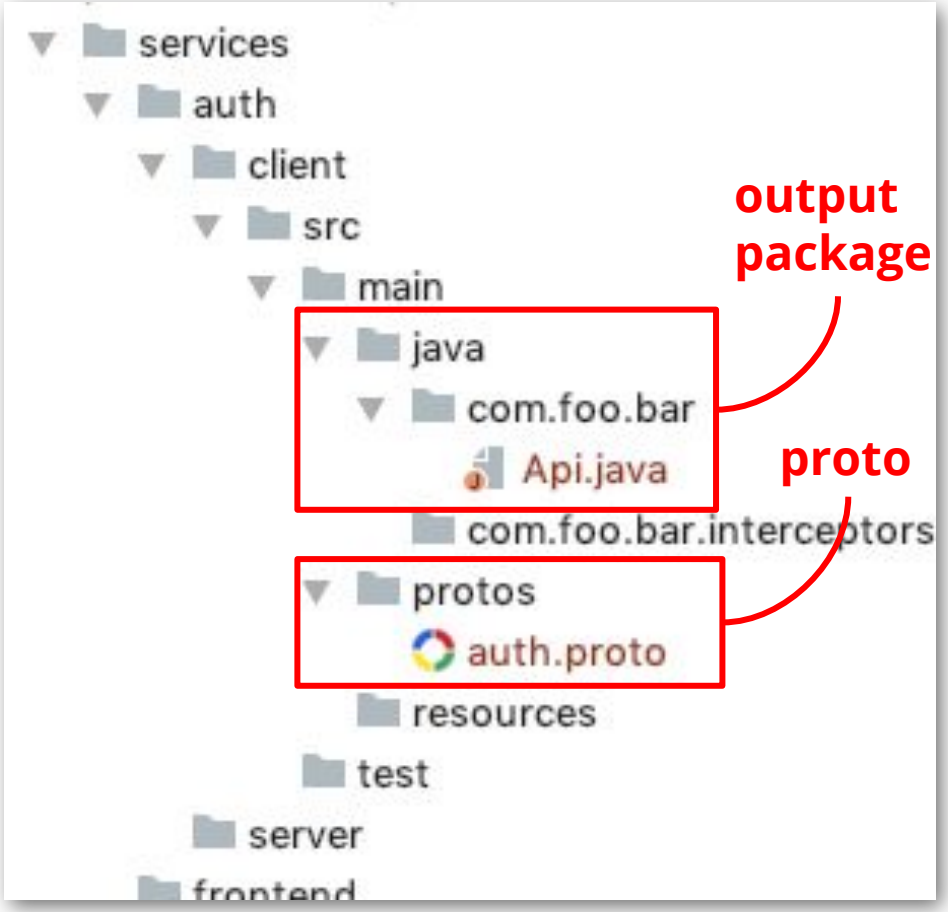
Code Organization

Protos next to outputs



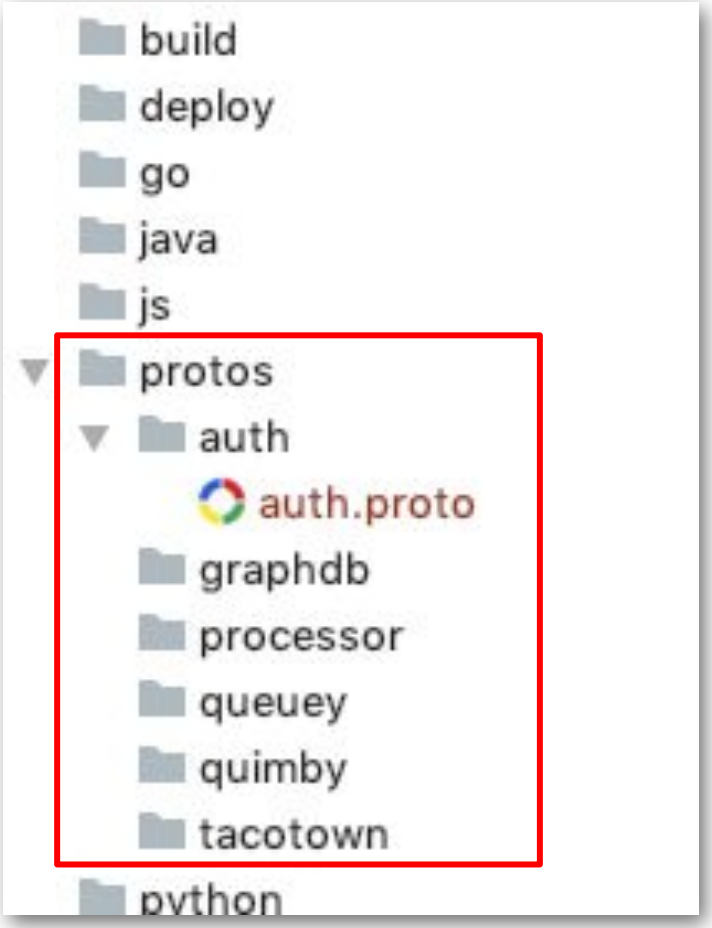
Code Organization

Protos next to outputs (Maven style)



Code Organization

Protos at higher level



Best Practices: Code Organization

Mono-repi: Multiple large “monolithic” repos

- Possibly a set of repos where each is an isolated mono-repo
- Possibly a mono-repo that is split by team, language, or other axis

Isolated? If no reason to share protos across repo, then use same strategy as for a single mono-repo.
Split by language? Create a “protos” mono-repo and use git submodules to inline protos into other repos.
Other? Could still use “protos” mono-repo. Or use multi-repo strategy.

Best Practices: Code Organization

Multi-repo: Numerous repos

- Each service or satellite of services is in its own repo
- Possibly each library/package is in its own repo
- *Harder to be prescriptive*

Will need build tools that understand proto dependencies to run `protoc`

- Git submodule may be your friend

Create “protos” repo for each service or satellite of services.

- Embed protos in repo with server implementation

Best Practices: Code Organization

Multi-repo: Numerous repos

- Define convention for proto packages. Organize folders by package elements.

  example: package “`foo.bar`” -> folder “`foo/bar`”

Adopting gRPC: Agenda

- Introduction to gRPC (“What?”)
- Benefits & Tradeoffs (“Why?”)
- Development Flow & Hurdles (“How?”)
- Overcoming Hurdles, Part 1: Best Practices
  - Providing a “gRPC skeleton”
  - Organizing proto sources
  - Evolving/versioning APIs
- Overcoming Hurdles, Part 2: Tools

Best Practices: Evolving an API

- With protocol buffer binary format:
  Many changes are backwards compatible

- Exposing API via JSON (e.g. `grpc-gateway`)?
  Less flexibility.
- Exposing APIs *publicly* (e.g. clients outside of your control)?
  Even less flexibility.

Best Practices: Evolving an API

Backwards-Compatible Changes:

- Can safely rename messages and their packages *
- Can safely add, rename, and remove fields **
- Can safely make many kinds of data type changes:
  - Optional to repeated **
  - String to bytes, Message to bytes **
  - Enum to int32 **
  - Widen integer types
- Can**not** rename services, methods, or their packages

Best Practices: Evolving an API

Making Backwards-**In**compatible Changes

- Semantic versioning
- Multi-step changes

Best Practices: Evolving an API

Semantic Versioning

- Put version number in proto package (typically last element)
  - example: `foo.bar.v1`
- Backwards incompatible changes require new version
- Must continue supporting old version until old clients turned down

Best Practices: Evolving an API

Multi-Step Changes

- Break up an incompatible change into a sequence of changes
- Each step in the sequence is backwards-compatible
- Removing things is last step

Best Practices: Evolving an API

Multi-Step Changes Example: Changing Types & Structure

Clients

Send old request.
Accept old response.

Protocol:

```
message Request {
  OldStructure old = 1;
}

message Response {
  OldStructure old = 1;
}
```

Servers

Accept old request.
Send old response.

Best Practices: Evolving an API

Multi-Step Changes Example: Changing Types & Structure

Clients

Send old request.
Accept old response.

Protocol:

```
message Request {
  OldStructure old = 1;
  NewStructure new = 2;
}

message Response {
  OldStructure old = 1;
  NewStructure new = 2;
}
```

Servers

Accept old *or* new request.
Send both old *and* new response.

Best Practices: Evolving an API

Multi-Step Changes Example: Changing Types & Structure

Clients

Send new request.
Accept new response.

Protocol:

```
message Request {
  OldStructure old = 1;
  NewStructure new = 2;
}

message Response {
  OldStructure old = 1;
  NewStructure new = 2;
}
```

Servers

Accept old *or* new request.
Send both old *and* new response.

Best Practices: Evolving an API

Multi-Step Changes Example: Changing Types & Structure

Clients

Send new request.
Accept new response.

Protocol:

```
message Request {
  NewStructure new = 2;
}

message Response {
  NewStructure new = 2;
}
```

Servers

Accept new request.
Send new response.

Adopting gRPC

- Introduction to gRPC (“What?”)
- Benefits & Tradeoffs (“Why?”)
- Development Flow & Hurdles (“How?”)
- Overcoming Hurdles, Part 1: Best Practices
- Overcoming Hurdles, Part 2: Tools

FullStory Tools for gRPC

https://github.com/fullstorydev

`grpcurl`

- Command-line client, a la `curl` or `wget`

`grpcui`

- User interface, in the vein of `postman`

FullStory Tools for gRPC

Demonstrations!

https://ui.grpc.me/

goto; chicago

Please

Remember to rate this session

Thank you!

follow us @gotochgo