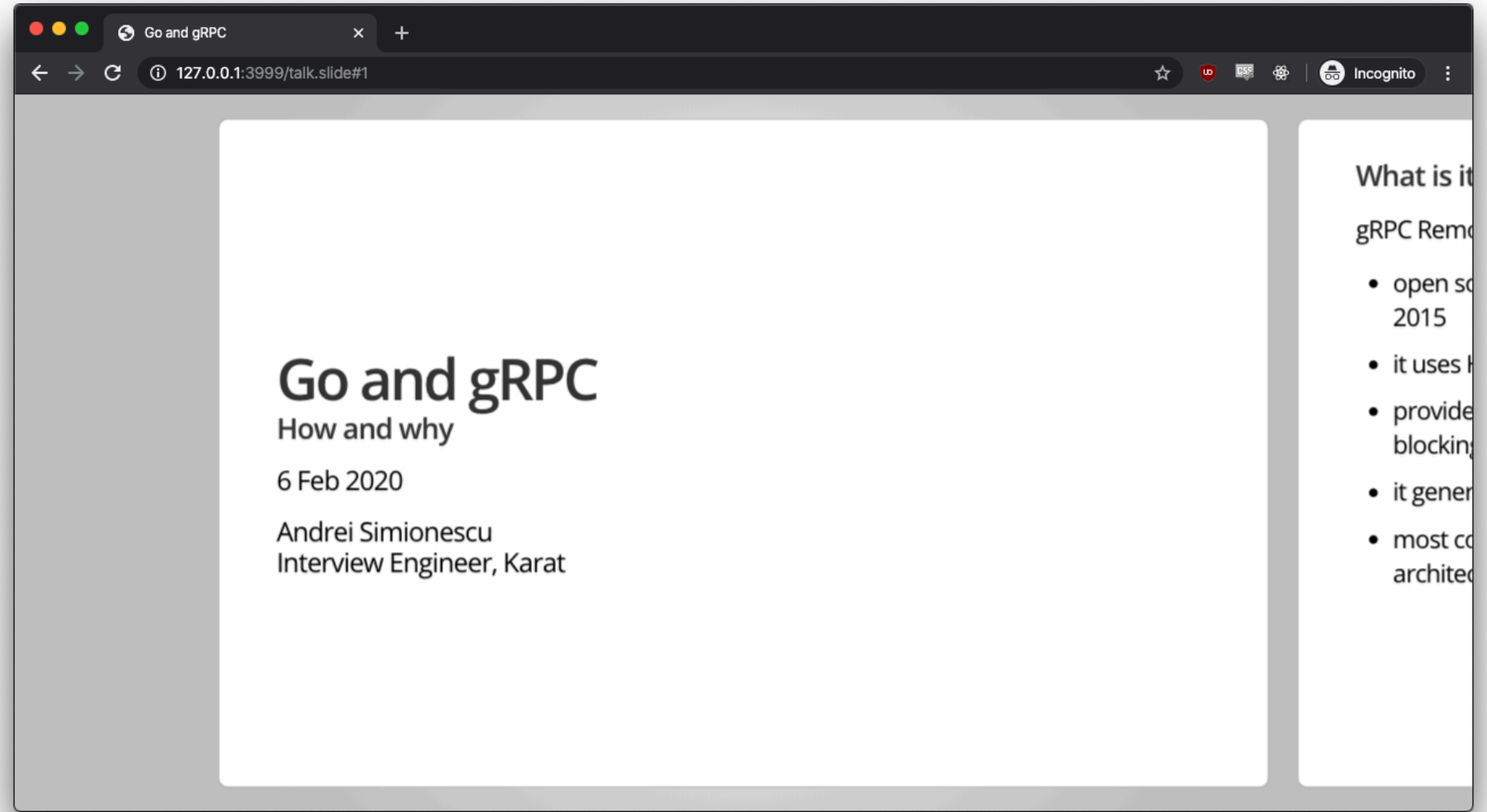
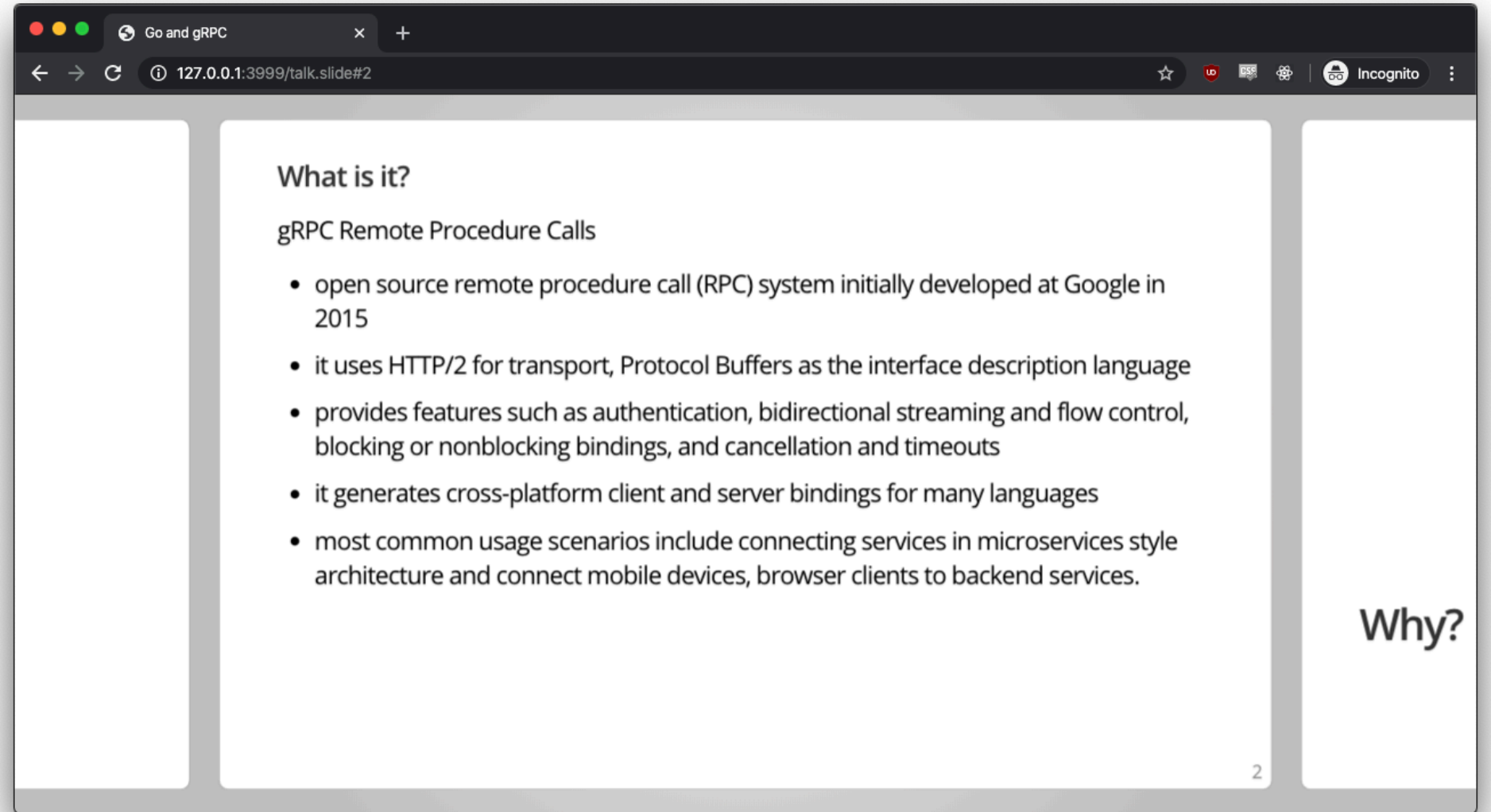
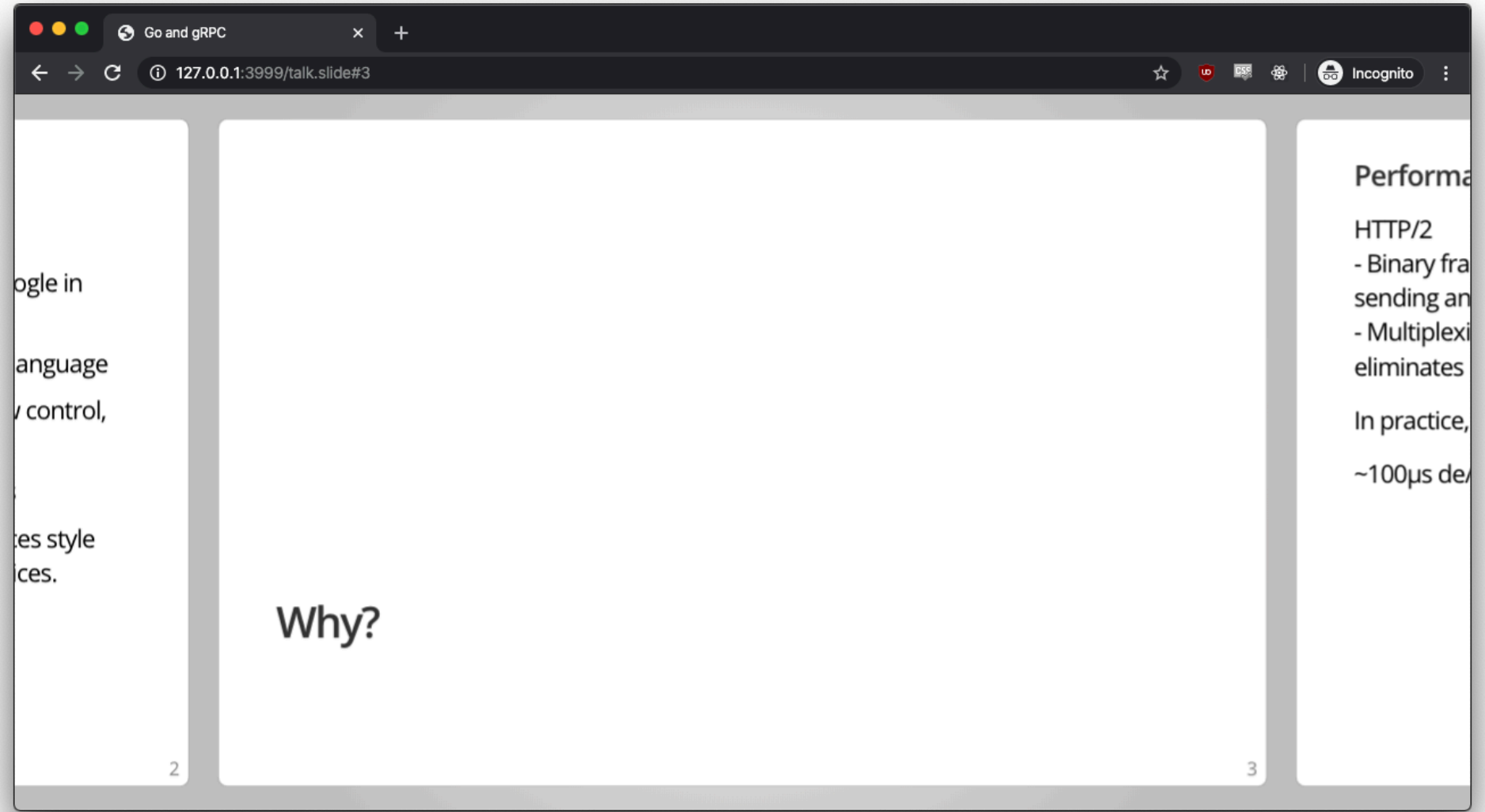
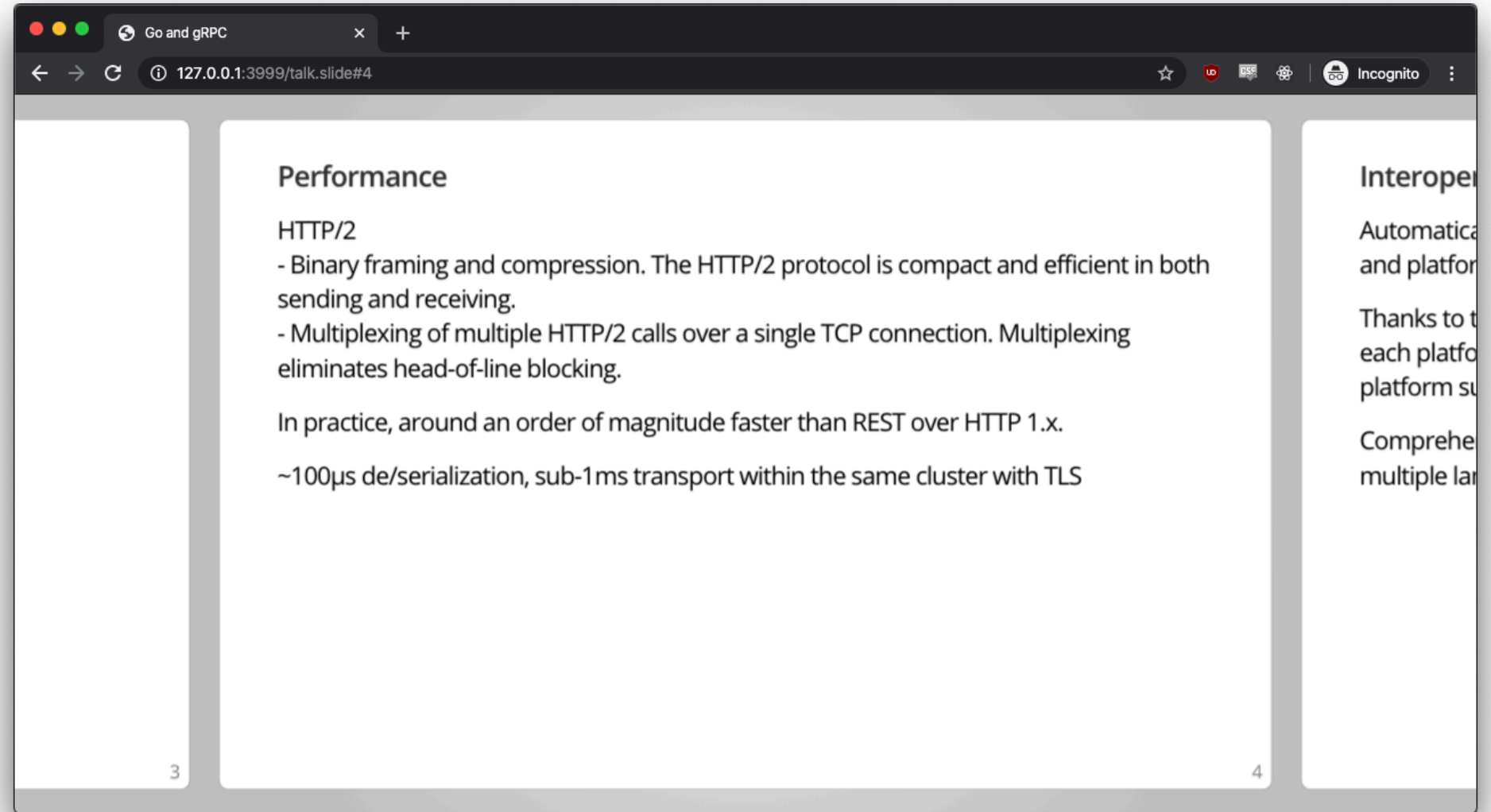
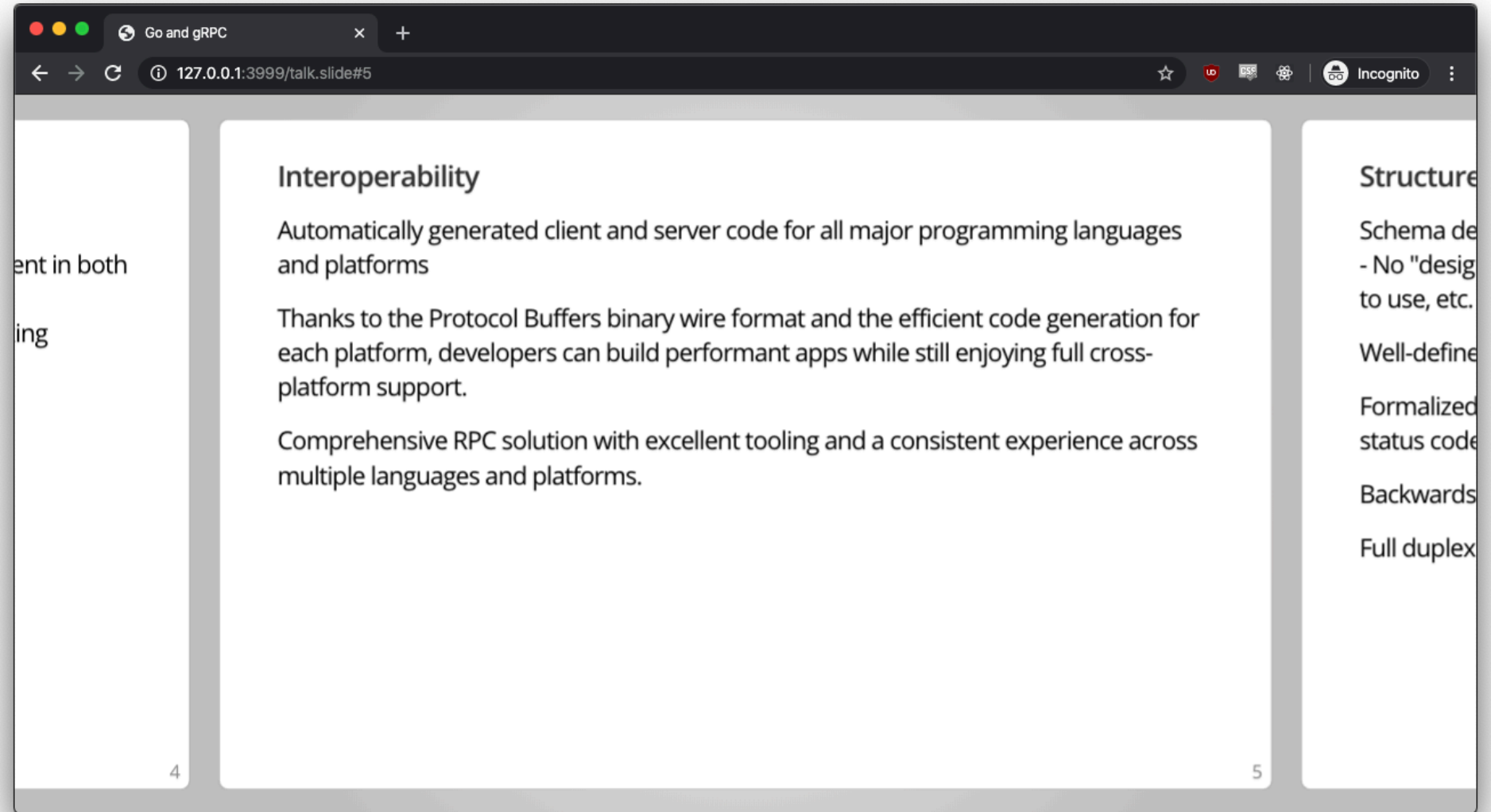
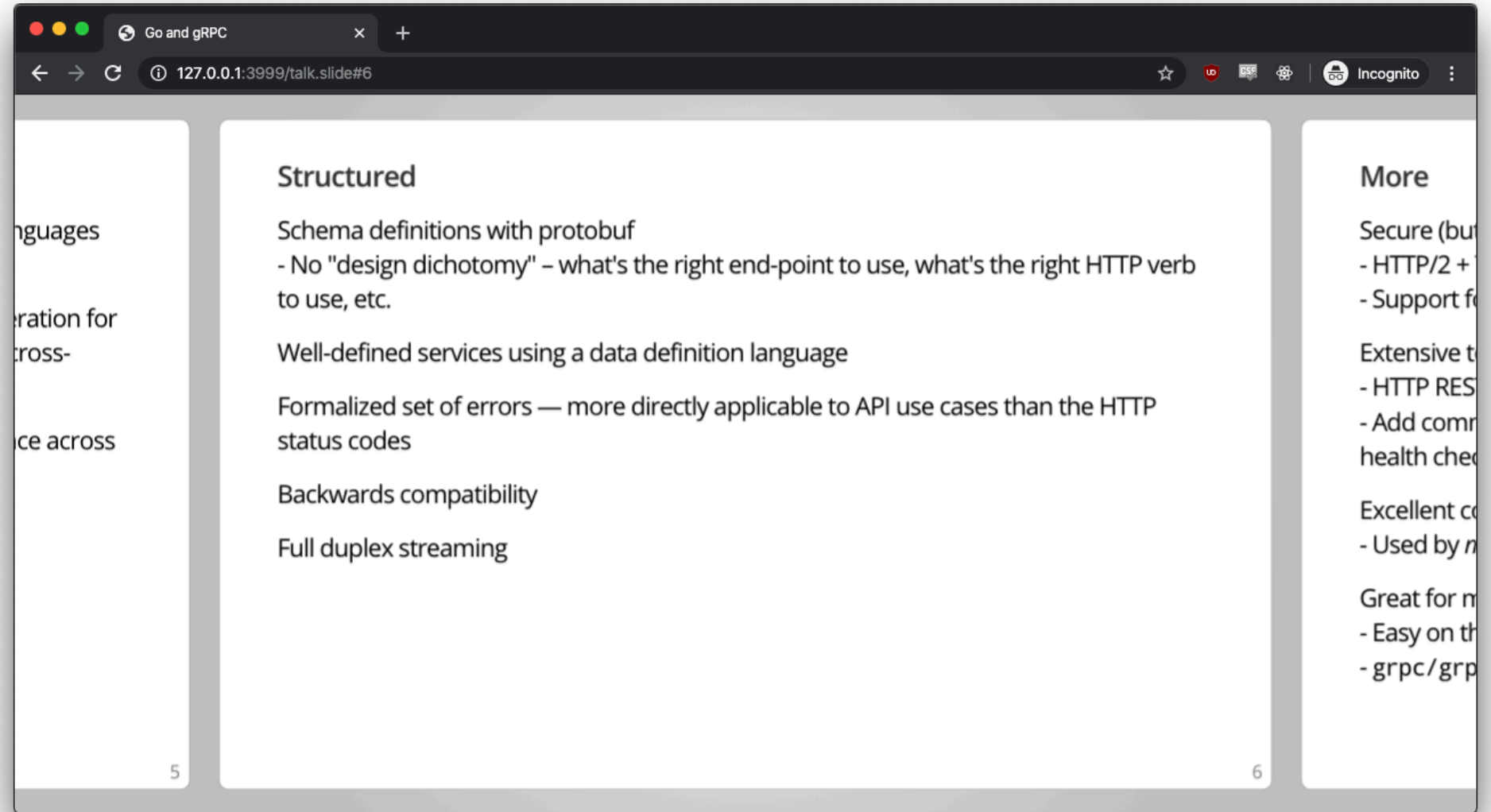

Go and gRPC
127.0.0.1:3999/talk.slide#1
Incognito
Go and gRPC
How and why
6 Feb 2020
Andrei Simionescu
Interview Engineer, Karat
What is it
gRPC Remo
open so
2015
it uses H
provide
blockin
it gener
most co
architec


Go and gRPC
127.0.0.1:3999/talk.slide#2
Incognito
What is it?
gRPC Remote Procedure Calls
open source remote procedure call (RPC) system initially developed at Google in 2015
it uses HTTP/2 for transport, Protocol Buffers as the interface description language
provides features such as authentication, bidirectional streaming and flow control, blocking or nonblocking bindings, and cancellation and timeouts
it generates cross-platform client and server bindings for many languages
most common usage scenarios include connecting services in microservices style architecture and connect mobile devices, browser clients to backend services.
2
Why?


Go and gRPC
127.0.0.1:3999/talk.slide#3
Incognito
ogle in
anguage
control,
es style
ices.
2
Why?
3
Performa
HTTP/2
- Binary fra
sending an
- Multiplexi
eliminates
In practice,
~100µs de/


Go and gRPC
127.0.0.1:3999/talk.slide#4
Incognito
Performance
HTTP/2
- Binary framing and compression. The HTTP/2 protocol is compact and efficient in both sending and receiving.
- Multiplexing of multiple HTTP/2 calls over a single TCP connection. Multiplexing eliminates head-of-line blocking.
In practice, around an order of magnitude faster than REST over HTTP 1.x.
~100µs de/serialization, sub-1ms transport within the same cluster with TLS
3
4


Go and gRPC
127.0.0.1:3999/talk.slide#5
Incognito
ent in both
ing
4
Interoperability
Automatically generated client and server code for all major programming languages and platforms
Thanks to the Protocol Buffers binary wire format and the efficient code generation for each platform, developers can build performant apps while still enjoying full cross-platform support.
Comprehensive RPC solution with excellent tooling and a consistent experience across multiple languages and platforms.
5
Structure
Schema de
- No "desig
to use, etc.
Well-define
Formalized
status code
Backwards
Full duplex


Go and gRPC
127.0.0.1:3999/talk.slide#6
Incognito
nguages
eration for
cross-
ice across
5
Structured
Schema definitions with protobuf
- No "design dichotomy" – what's the right end-point to use, what's the right HTTP verb to use, etc.
Well-defined services using a data definition language
Formalized set of errors — more directly applicable to API use cases than the HTTP status codes
Backwards compatibility
Full duplex streaming
6
More
Secure (but
- HTTP/2 +
- Support f
Extensive t
- HTTP RES
- Add comr
health chec
Excellent c
- Used by n
Great for m
- Easy on th
- grpc/grp

TTP verb

HTTP

More

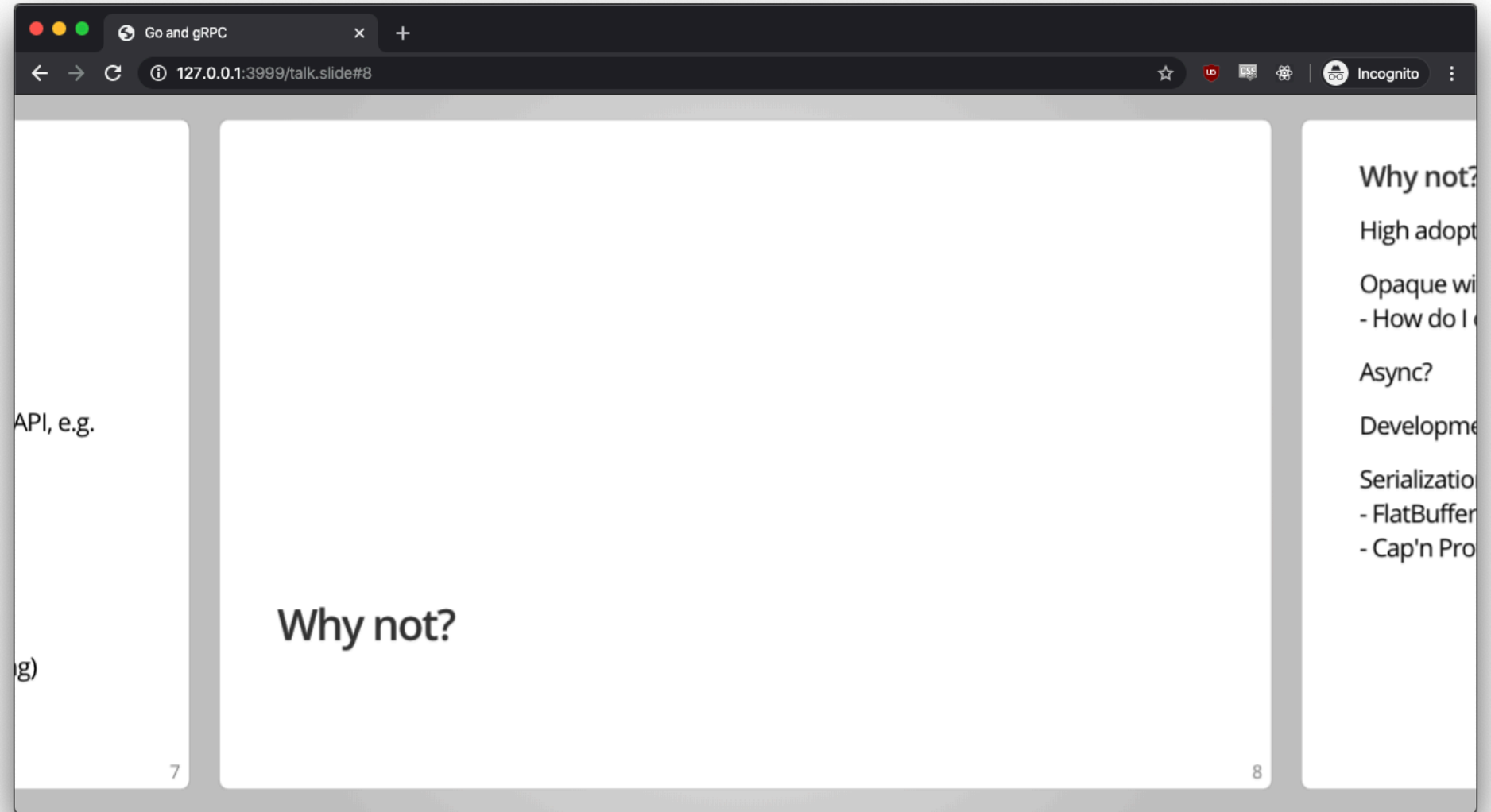
Secure (but not by default)
- HTTP/2 + TLS
- Support for Client Certificate authentication

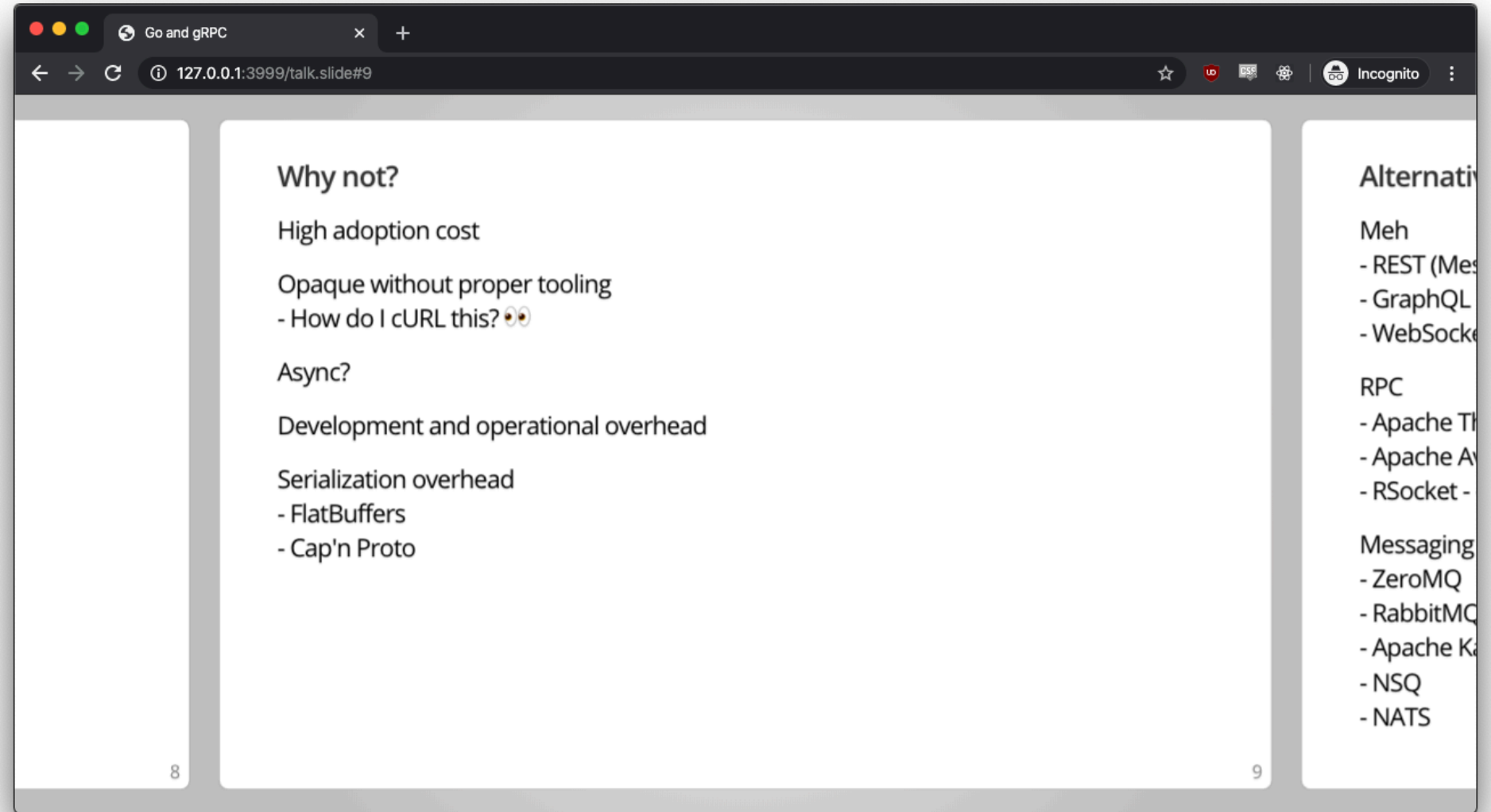
Extensive tooling
- HTTP REST interface via annotations
- Add common functionality to multiple service endpoints via the Interceptor API, e.g. health checks, authentication

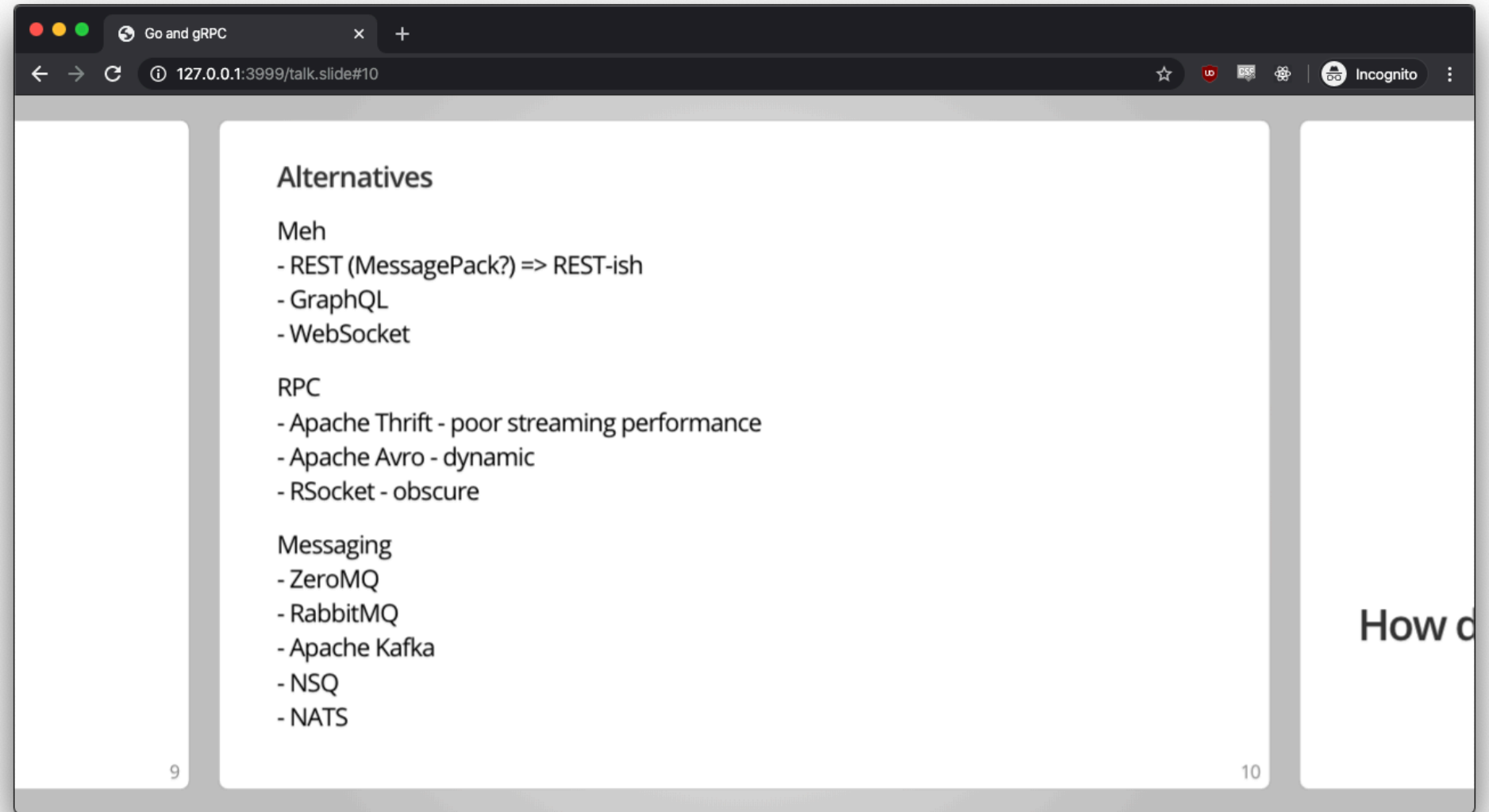
Excellent community support
- Used by *many* large projects and companies

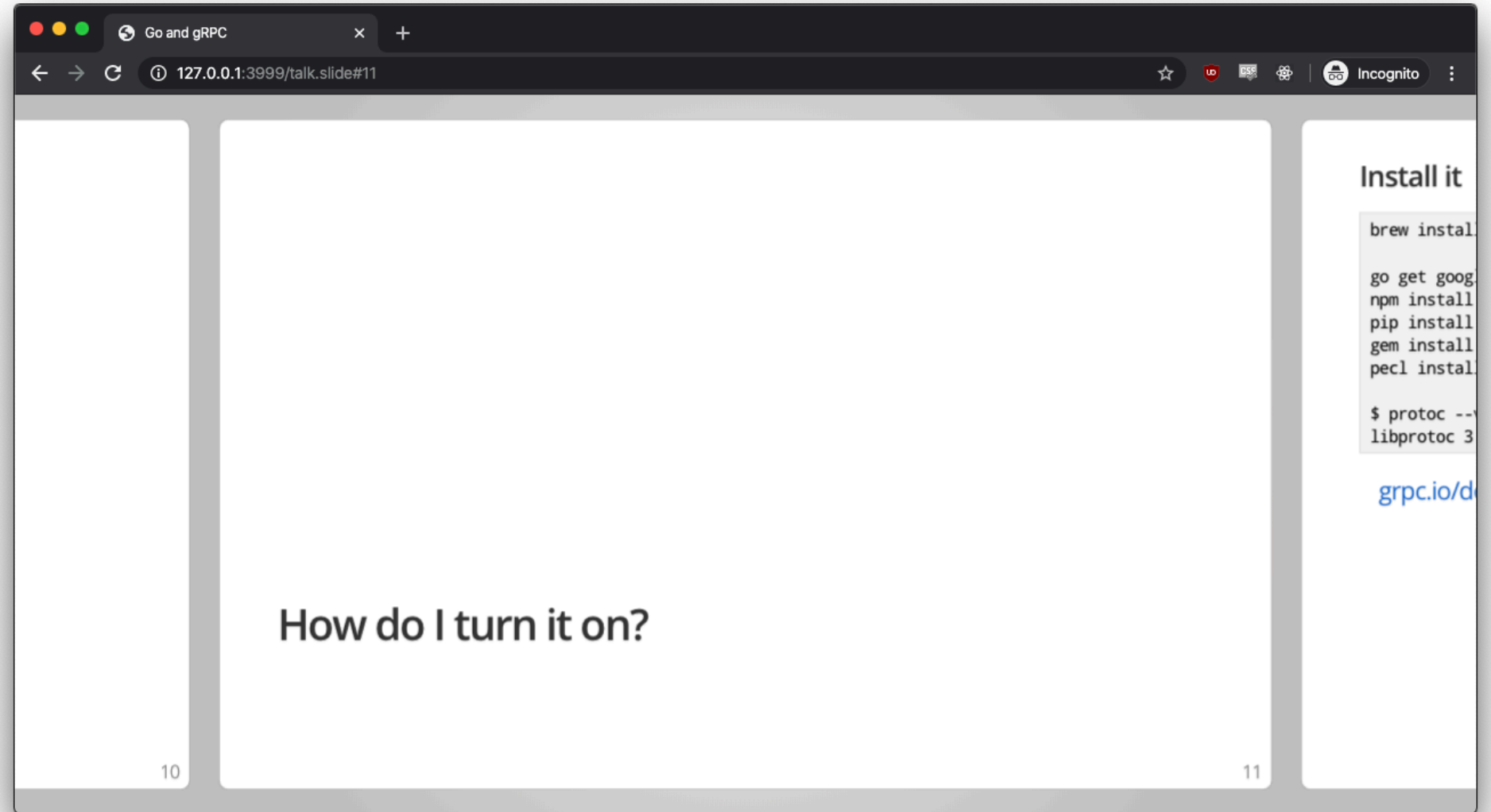
Great for mobile and even web clients
- Easy on the battery
- `grpc/grpc-web` and `improbable-eng/grpc-web` (neither support streaming)

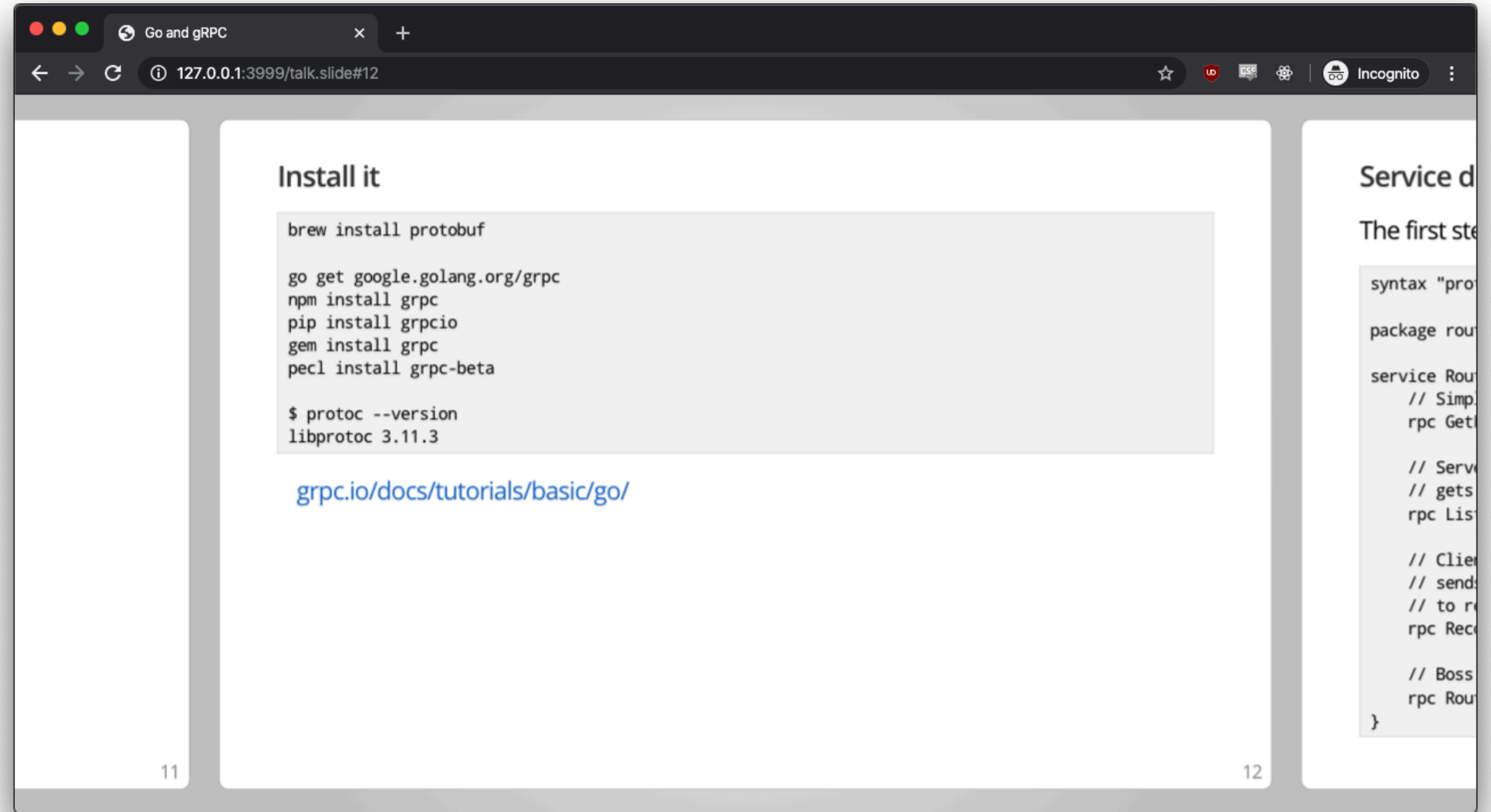
Why n


Go and gRPC
127.0.0.1:3999/talk.slide#8
Incognito
API, e.g.
g)
7
Why not?
8
Why not?
High adopt
Opaque wi
- How do I
Async?
Developme
Serializatio
- FlatBuffer
- Cap'n Pro


Go and gRPC
127.0.0.1:3999/talk.slide#9
Incognito
8
Why not?
High adoption cost
Opaque without proper tooling
- How do I cURL this? 👀
Async?
Development and operational overhead
Serialization overhead
- FlatBuffers
- Cap'n Proto
9
Alternati
Meh
- REST (Mes
- GraphQL
- WebSocke
RPC
- Apache Th
- Apache Av
- RSocket -
Messaging
- ZeroMQ
- RabbitMQ
- Apache Ka
- NSQ
- NATS


Go and gRPC
127.0.0.1:3999/talk.slide#10
Incognito
Alternatives
Meh
- REST (MessagePack?) => REST-ish
- GraphQL
- WebSocket
RPC
- Apache Thrift - poor streaming performance
- Apache Avro - dynamic
- RSocket - obscure
Messaging
- ZeroMQ
- RabbitMQ
- Apache Kafka
- NSQ
- NATS
9
10
How d


Go and gRPC
127.0.0.1:3999/talk.slide#11
Incognito
10
How do I turn it on?
11
Install it
brew instal
go get goog
npm install
pip install
gem install
pecl instal
$ protoc --
libprotoc 3
grpc.io/d


Go and gRPC
127.0.0.1:3999/talk.slide#12
Incognito
Install it
brew install protobuf
go get google.golang.org/grpc
npm install grpc
pip install grpcio
gem install grpc
pecl install grpc-beta
$ protoc --version
libprotoc 3.11.3
grpc.io/docs/tutorials/basic/go/
11
12
Service d
The first ste
syntax "pro
package rou
service Rou
// Simp
rpc Get
// Serv
// gets
rpc Lis
// Clie
// send
// to r
rpc Rec
// Boss
rpc Rou
}

Service definition

The first step is to define a gRPC *service* and the method *request* and *response* types.

```
syntax "proto3";

package routeguide;

service RouteGuide {
    // Simple RPC, the client sends a request and waits for the response to come back.
    rpc GetFeature(Point) returns (Feature) {}

    // Server-side streaming RPC where the client sends a request to the server and
    // gets a stream to read a sequence of messages back, until there are no more messages.
    rpc ListFeatures(Rectangle) returns (stream Feature) {}

    // Client-side streaming RPC where the clients writes a sequence of messages and
    // sends them to the server using a provided stream. Then, it waits for the server
    // to read them all and return its response.
    rpc RecordRoute(stream Point) returns (RouteSummary) {}

    // Boss fight.
    rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}
}
```


Message

Our .proto
request an

```
message Poi
    int32 l
    int32 l
}

// A latitu
// points "
message Rec
    // One
    Point l

    // The
    Point h
}
```

types.

```
ages.
```


Message definition

Our `.proto` file also contains protocol buffer message type definitions for all the request and response types used in our service methods.

```
message Point {
    int32 latitude = 1;
    int32 longitude = 2;
}

// A latitude-longitude rectangle, represented as two diagonally opposite
// points "lo" and "hi".
message Rectangle {
    // One corner of the rectangle.
    Point lo = 1;

    // The other corner of the rectangle.
    Point hi = 2;
}
```

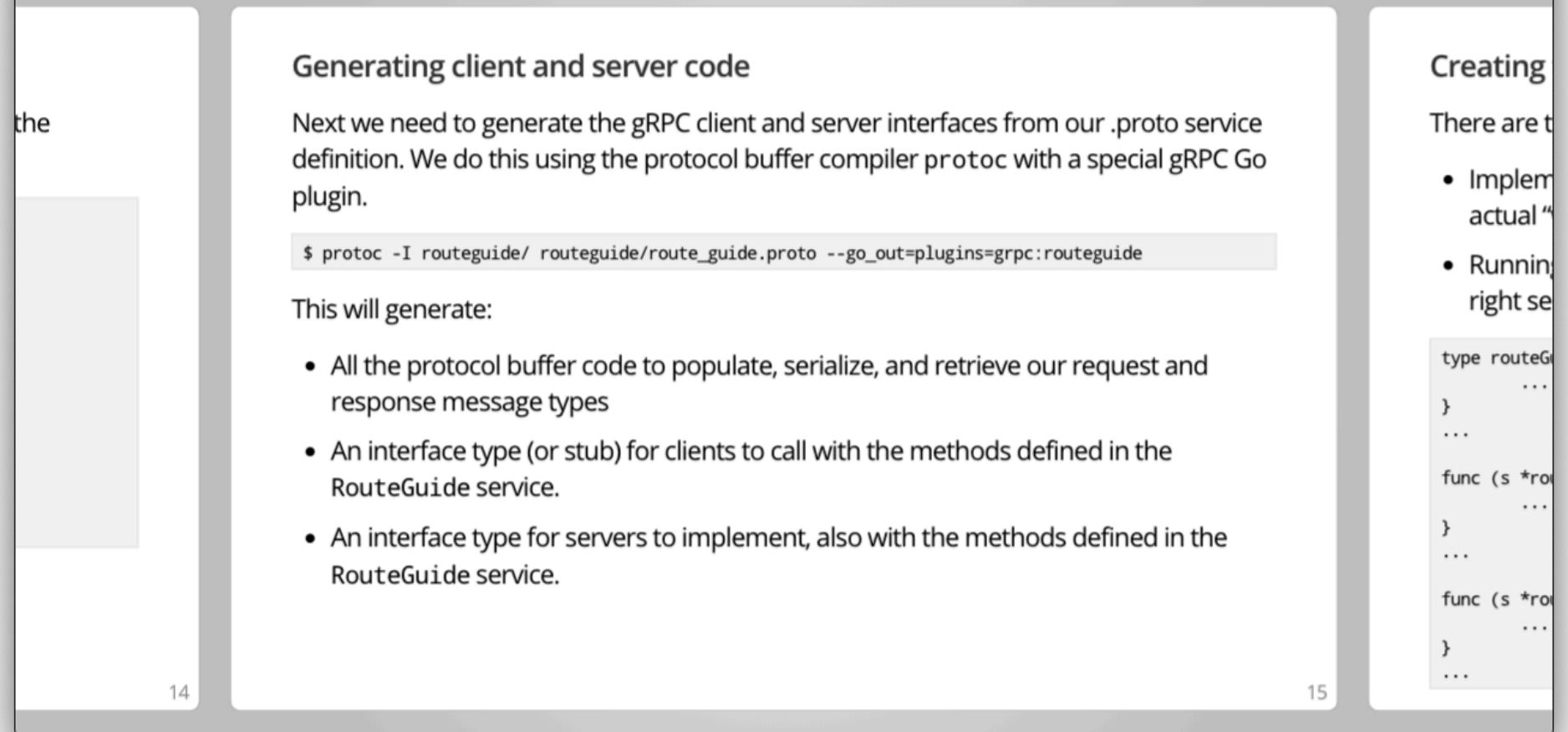

Generati

Next we ne
definition. V
plugin.

```
$ protoc -I
```

This will ge

- All the p
  respons
- An inter
  RouteG
- An inter
  RouteG


Go and gRPC
127.0.0.1:3999/talk.slide#15
Incognito
the
14
Generating client and server code
Next we need to generate the gRPC client and server interfaces from our .proto service definition. We do this using the protocol buffer compiler protoc with a special gRPC Go plugin.
$ protoc -I routeguide/ routeguide/route_guide.proto --go_out=plugins=grpc:routeguide
This will generate:
All the protocol buffer code to populate, serialize, and retrieve our request and response message types
An interface type (or stub) for clients to call with the methods defined in the RouteGuide service.
An interface type for servers to implement, also with the methods defined in the RouteGuide service.
15
Creating
There are t
Implem
actual "
Running
right se
type routeG
...
}
...
func (s *ro
...
}
...
func (s *ro
...
}
...

to service
l gRPC Go

t and

ne

l in the

15

Creating the server

There are two parts to making our RouteGuide service do its job:

- Implementing the service interface generated from our service definition: doing the actual "work" of our service.
- Running a gRPC server to listen for requests from clients and dispatch them to the right service implementation.

```
type routeGuideServer struct {
        ...
}
...

func (s *routeGuideServer) GetFeature(ctx context.Context, point *pb.Point) (*pb.Feature, error) {
        ...
}
...

func (s *routeGuideServer) ListFeatures(rect *pb.Rectangle, stream pb.RouteGuide_ListFeaturesServer
        ...
}
...
```

16

Simple RI

routeGuid
Point from
database ir

```
func (s *ro
    for _,
        if
        }
    }
    // No f
    return
}
```

The metho
buffer requ
return it alo
the data ca

doing the

m to the

```
re, error) {
```

```
eaturesServer
```


Simple RPC

`routeGuideServer` implements all our service methods. `GetFeature` simply gets a `Point` from the client and returns the corresponding feature information from its database in a `Feature`.

```
func (s *routeGuideServer) GetFeature(ctx context.Context, point *pb.Point) (*pb.Feature, error) {
    for _, feature := range s.savedFeatures {
        if proto.Equal(feature.Location, point) {
            return feature, nil
        }
    }
    // No feature was found, return an unnamed feature
    return &pb.Feature{"", point}, nil
}
```

The method is passed a context object for the RPC and the client's `Point` protocol buffer request. After populating the `Feature` with the appropriate information, we return it along with a `nil` error to tell gRPC that we've finished dealing with the RPC and the data can be returned to the client.

Server-si

Now, we n

```
func (s *ro
    for _,
        if
        }
    }
    return
}
```

Instead of
time we ge
and a spec

Finally, as i
responses.

gets a
m its

```
re, error) {
```

tocol
n, we
e RPC and

Server-side streaming RPC

Now, we need to send back multiple `Features` to the client.

```
func (s *routeGuideServer) ListFeatures(rect *pb.Rectangle, stream pb.RouteGuide_ListFeaturesServer
    for _, feature := range s.savedFeatures {
        if inRange(feature.Location, rect) {
            if err := stream.Send(feature); err != nil {
                return err
            }
        }
    }
    return nil
}
```

Instead of getting simple request and response objects in our method parameters, this time we get a request object (the `Rectangle` in which our client wants to find `Features`) and a special object to write our responses to.

Finally, as in our simple RPC, we return a nil error to tell gRPC that we've finished writing responses.

Client-sid

```
func (s *ro
    var poi
    var las
    for {
        poi
        if
        }
        if
        }
        poi
        for
        }
        if
        }
        las
    }
}
```

eaturesServer

eters, this
Features)

ed writing

Client-side streaming RPC

```
func (s *routeGuideServer) RecordRoute(stream pb.RouteGuide_RecordRouteServer) error {
    var pointCount, featureCount, distance int32
    var lastPoint *pb.Point
    for {
        point, err := stream.Recv()
        if err == io.EOF {
            endTime := time.Now()
            return stream.SendAndClose(&pb.RouteSummary{/*...*/})
        }
        if err != nil {
            return err
        }
        pointCount++
        for _, feature := range s.savedFeatures {
            if proto.Equal(feature.Location, point) {
                featureCount++
            }
        }
        if lastPoint != nil {
            distance += calcDistance(lastPoint, point)
        }
        lastPoint = point
    }
}
```


Bidirectic

```
func (s *ro
    for {
        in,
        if
        }
        if
        }
        key
        for
        }
    }
}
```

The syntax
except the
because it's

Bidirectional streaming RPC

```
func (s *routeGuideServer) RouteChat(stream pb.RouteGuide_RouteChatServer) error {
    for {
        in, err := stream.Recv()
        if err == io.EOF {
            return nil
        }
        if err != nil {
            return err
        }
        key := serialize(in.Location)
               ... // look for notes to be sent to client
        for _, note := range s.routeNotes[key] {
            if err := stream.Send(note); err != nil {
                return err
            }
        }
    }
}
```

The syntax for reading and writing here is very similar to our client-streaming method, except the server uses the stream's `Send()` method rather than `SendAndClose()` because it's writing multiple responses.

Starting t

Once we've
that clients

```
flag.Parse(
lis, err :=
if err != n
        log
}
grpcServer
pb.Register
... // dete
grpcServer.
```

method,
se()

Starting the server

Once we've implemented all our methods, we also need to start up a gRPC server so that clients can actually use our service.

```
flag.Parse()
lis, err := net.Listen("tcp", fmt.Sprintf(":%d", *port))
if err != nil {
        log.Fatalf("failed to listen: %v", err)
}
grpcServer := grpc.NewServer()
pb.RegisterRouteGuideServer(grpcServer, &routeGuideServer{})
... // determine whether to use TLS
grpcServer.Serve(lis)
```


Creating

```
conn, err :
if err != n
    ...
}
defer conn.
```

Once the g

```
client := p
```

Calling the
method.

```
feature, er
if err != n
        ...
}
```

rver so

21

Creating the client

```
conn, err := grpc.Dial(*serverAddr)
if err != nil {
    ...
}
defer conn.Close()
```

Once the gRPC *channel* is set up, we need a client *stub* to perform RPCs.

```
client := pb.NewRouteGuideClient(conn)
```

Calling the simple RPC `GetFeature` is nearly as straightforward as calling a local method.

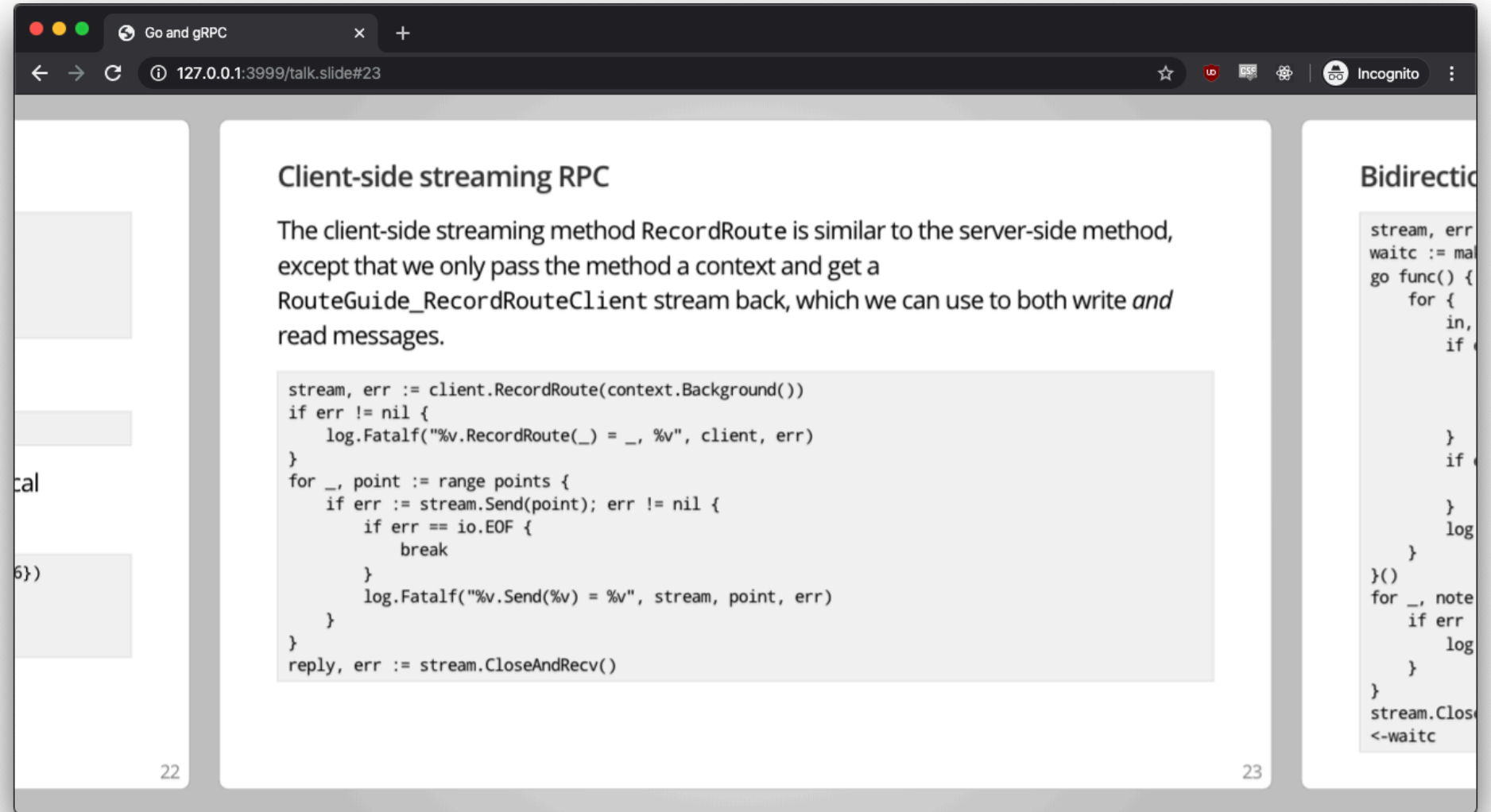
```
feature, err := client.GetFeature(context.Background(), &pb.Point{409146138, -746188906})
if err != nil {
        ...
}
```

22

Client-sid

The client-s
except that
RouteGuid
read messa

```
stream, err
if err != n
    log.Fat
}
for _, poin
    if err
        if
        }
        log
    }
}
reply, err
```


Go and gRPC
127.0.0.1:3999/talk.slide#23
Incognito
cal
6})
22
Client-side streaming RPC
The client-side streaming method RecordRoute is similar to the server-side method, except that we only pass the method a context and get a RouteGuide_RecordRouteClient stream back, which we can use to both write and read messages.
stream, err := client.RecordRoute(context.Background())
if err != nil {
log.Fatalf("%v.RecordRoute(_) = _, %v", client, err)
}
for _, point := range points {
if err := stream.Send(point); err != nil {
if err == io.EOF {
break
}
log.Fatalf("%v.Send(%v) = %v", stream, point, err)
}
}
reply, err := stream.CloseAndRecv()
23
Bidirectio
stream, err
waitc := ma
go func() {
for {
in,
if
}
if
}
log
}
}()
for _, note
if err
log
}
}
stream.Clos
<-waitc

ıethod,

rite *and*

Bidirectional streaming RPC

```
stream, err := client.RouteChat(context.Background())
waitc := make(chan struct{})
go func() {
    for {
        in, err := stream.Recv()
        if err == io.EOF {
            // read done.
            close(waitc)
            return
        }
        if err != nil {
            log.Fatalf("Failed to receive a note : %v", err)
        }
        log.Printf("Got message %s at point(%d, %d)", in.Message, in.Location.Latitude, in.Locatior
    }
}()
for _, note := range notes {
    if err := stream.Send(note); err != nil {
        log.Fatalf("Failed to send a note: %v", err)
    }
}
stream.CloseSend()
<-waitc
```


Nice t


Go and gRPC
127.0.0.1:3999/talk.slide#25
Incognito
, in.Locatior
24
Nice to know
25
Well kno
Package go
Any (messag
DoubleValue
EnumValue (
Field.Kind
Int64Value
NullValue (
StringValue
Type (messa
.proto
import "goo
import "goo
message Nam
string na
google.pr
google.pr
}

25

Well known types

Package google.protobuf

```
Any (message)  Api (message)  BoolValue (message)  BytesValue (message)
DoubleValue (message)  Duration (message)  Empty (message)  Enum (message)
EnumValue (message)  Field (message) Field.Cardinality (enum)
Field.Kind (enum)  FieldMask (message)  FloatValue (message)  Int32Value (message)
Int64Value (message) ListValue (message)  Method (message)  Mixin (message)
NullValue (enum) Option (message)  SourceContext (message)
StringValue (message) Struct (message)  Syntax (enum) Timestamp (message)
Type (message)  UInt32Value (message)  UInt64Value (message)  Value (message)
```

.proto

```
import "google/protobuf/struct.proto"
import "google/protobuf/timestamp.proto"

message NamedStruct {
  string name = 1;
  google.protobuf.Struct definition = 2;
  google.protobuf.Timestamp last_modified = 3;
}
```

26

github.co

- fast ma
- more c
- goproto
- less typ
- peace c
- other s

```
$ go get gi
$ go get gi
$ go get gi
$ go get gi

$ protoc --
```

github.com/gogo/protobuf (!!)

- fast marshalling and unmarshalling
- more canonical Go structures
- goprotobuf compatibility
- less typing by optionally generating extra helper code
- peace of mind by optionally generating test and benchmark code
- other serialization formats

```
$ go get github.com/gogo/protobuf/proto
$ go get github.com/gogo/protobuf/jsonpb
$ go get github.com/gogo/protobuf/protoc-gen-gogo
$ go get github.com/gogo/protobuf/gogoproto

$ protoc --gofast_out=plugins=grpc:. my.proto
```


Please, m

```
protoc -I=.
-I=$GOPATH/
--{binary}_
```

That's not

```
protoc -I=.
Mgoogle/pro
Mgoogle/pro
Mgoogle/pro
Mgoogle/pro
Mgoogle/pro
myproto.pro
```

Please, make it stop

```
protoc -I=. -I=$GOPATH/src \
-I=$GOPATH/src/github.com/gogo/protobuf/protobuf \
--{binary}_out=. myproto.proto
```

That's not so bad. Wait, you want to use the files from `google/protobuf`?

```
protoc -I=. -I=$GOPATH/src -I=$GOPATH/src/github.com/gogo/protobuf/protobuf --{binary}_out=\
Mgoogle/protobuf/any.proto=github.com/gogo/protobuf/types,\
Mgoogle/protobuf/duration.proto=github.com/gogo/protobuf/types,\
Mgoogle/protobuf/struct.proto=github.com/gogo/protobuf/types,\
Mgoogle/protobuf/timestamp.proto=github.com/gogo/protobuf/types,\
Mgoogle/protobuf/wrappers.proto=github.com/gogo/protobuf/types:. \
myproto.proto
```


Intercept

A native wa

```
import "gith

myServer :=
    grpc.St
        grp
        grp
        grp
        grp
        grp
        grp
    )),
    grpc.Una
        grp
        grp
        grp
        grp
        grp
        grp
    )),
)
```

_out=\

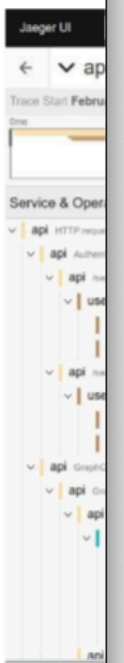
Interceptor API and go-grpc-middleware

A native way to add common functionality to either servers or clients.

```
import "github.com/grpc-ecosystem/go-grpc-middleware"

myServer := grpc.NewServer(
    grpc.StreamInterceptor(grpc_middleware.ChainStreamServer(
        grpc_ctxtags.StreamServerInterceptor(),
        grpc_opentracing.StreamServerInterceptor(),
        grpc_prometheus.StreamServerInterceptor,
        grpc_zap.StreamServerInterceptor(zapLogger),
        grpc_auth.StreamServerInterceptor(myAuthFunction),
        grpc_recovery.StreamServerInterceptor(),
    )),
    grpc.UnaryInterceptor(grpc_middleware.ChainUnaryServer(
        grpc_ctxtags.UnaryServerInterceptor(),
        grpc_opentracing.UnaryServerInterceptor(),
        grpc_prometheus.UnaryServerInterceptor,
        grpc_zap.UnaryServerInterceptor(zapLogger),
        grpc_auth.UnaryServerInterceptor(myAuthFunction),
        grpc_recovery.UnaryServerInterceptor(),
    )),
)
```


For exam

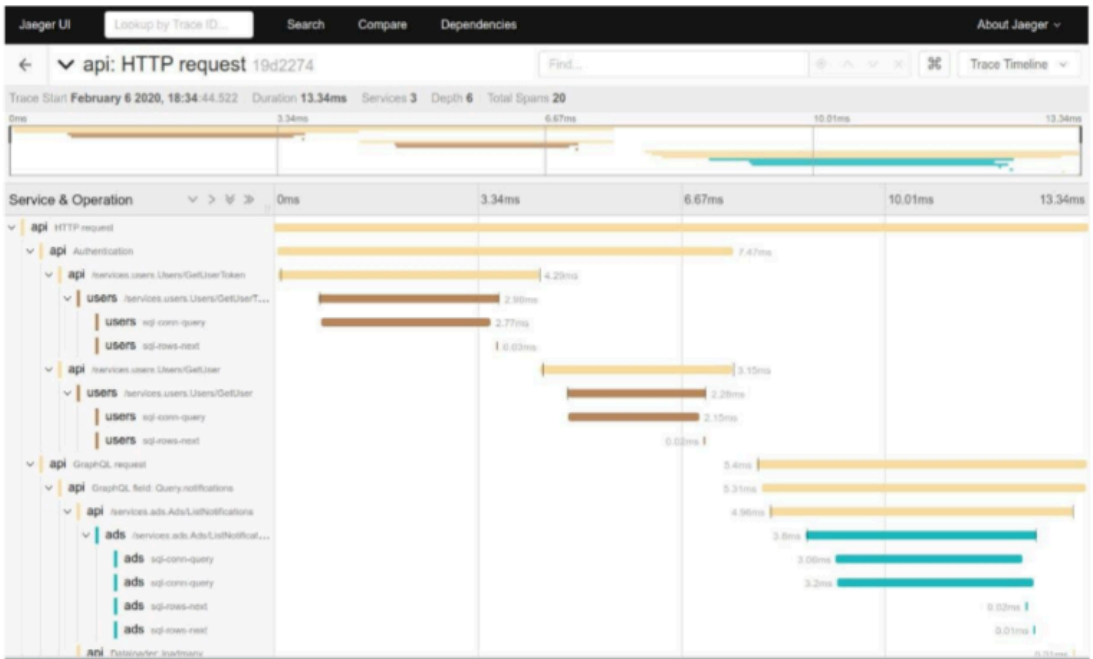


Go and gRPC

127.0.0.1:3999/talk.slide#30

Incognito

For example



29

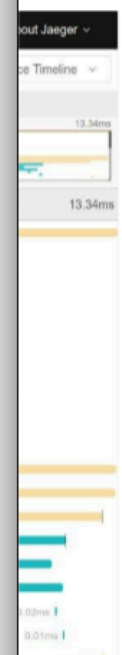
30

Does i

Go and gRPC

127.0.0.1:3999/talk.slide#31

Incognito



30

Does it speak REST though?

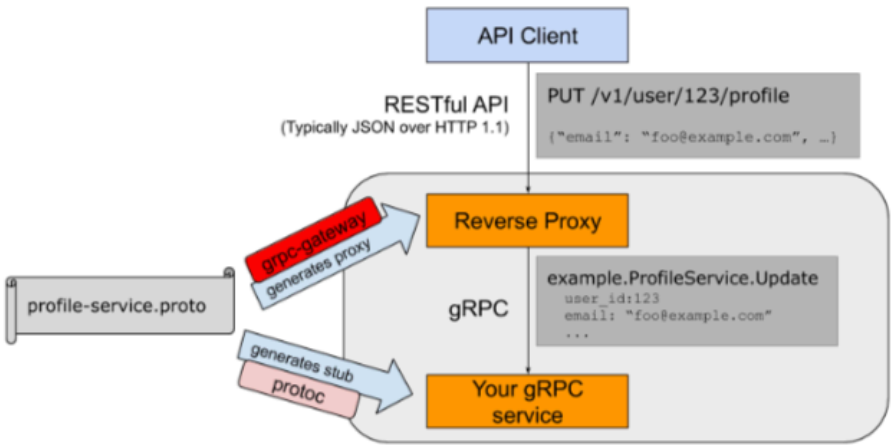
31

github.co

```
--grpc-gate
--swagger_o
```

"Simple"

github.com/grpc-ecosystem/grpc-gateway




```
--grpc-gateway_out=logtostderr=true,repeated_path_param_separator=ssv:.
--swagger_out=logtostderr=true,repeated_path_param_separator=ssv:.
```

"Simple" example

Show,

Go and gRPC

127.0.0.1:3999/talk.slide#33

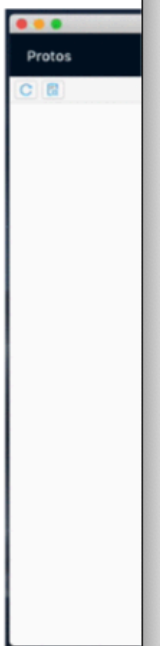
Incognito

32

Show, don't tell

33

https://gi



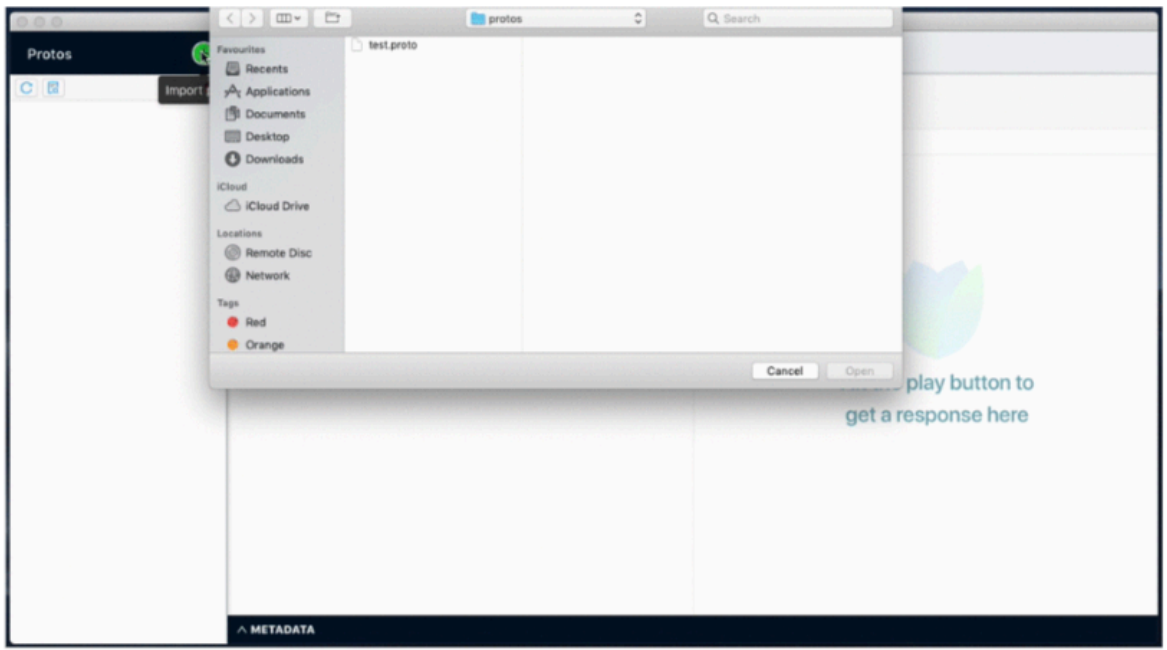
Go and gRPC

127.0.0.1:3999/talk.slide#34

Incognito

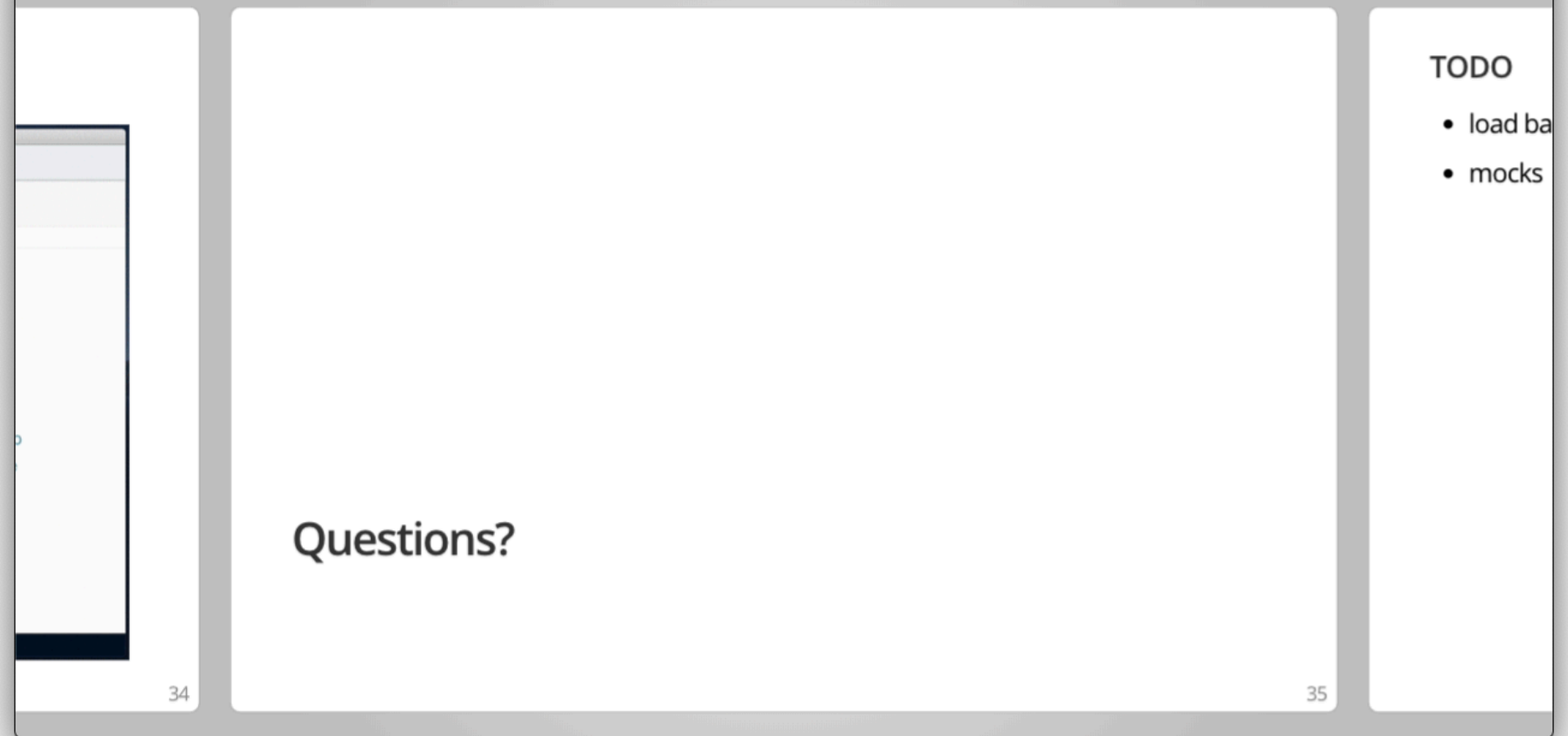
33

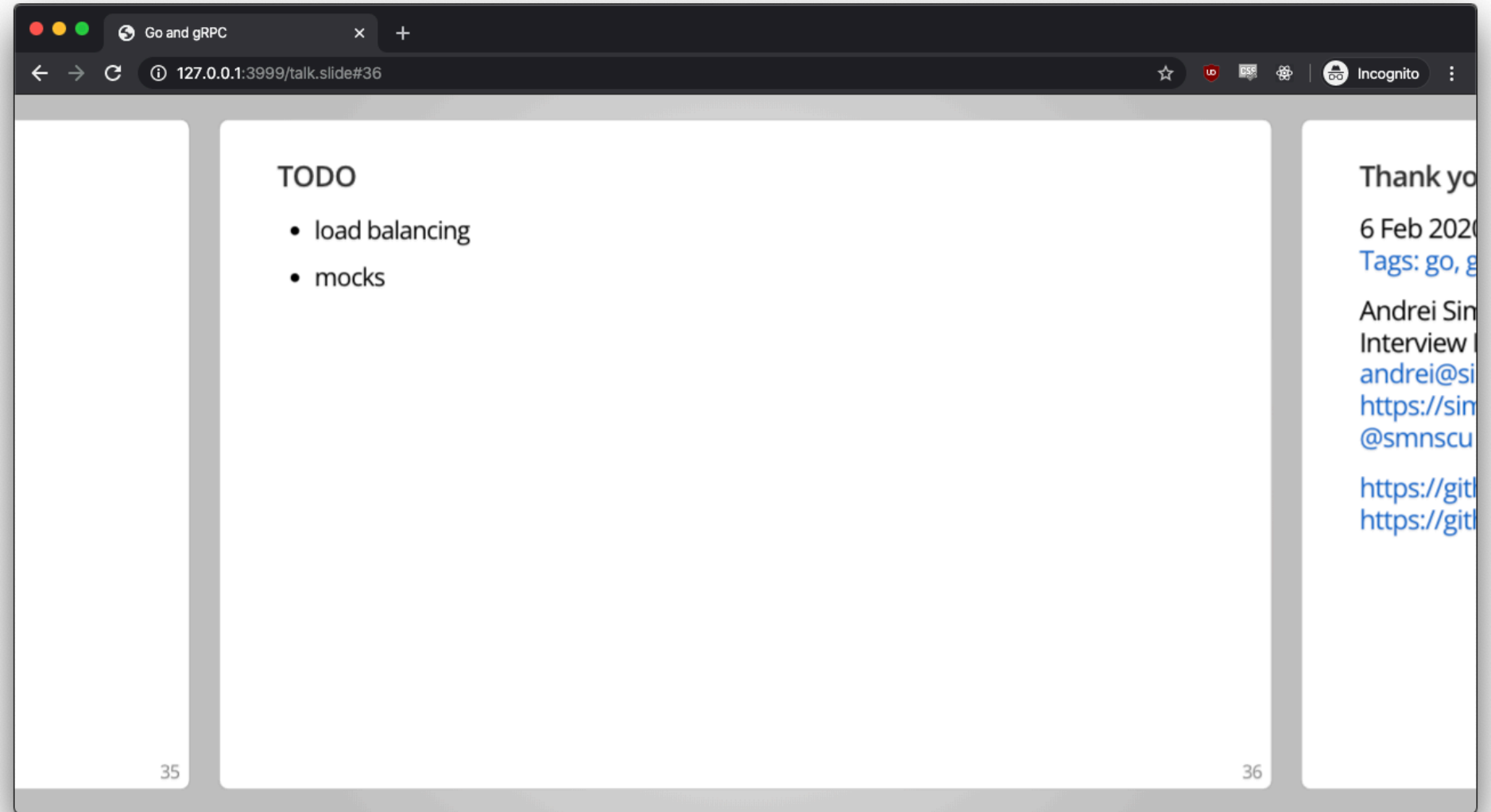
https://github.com/uw-labs/bloomrpc

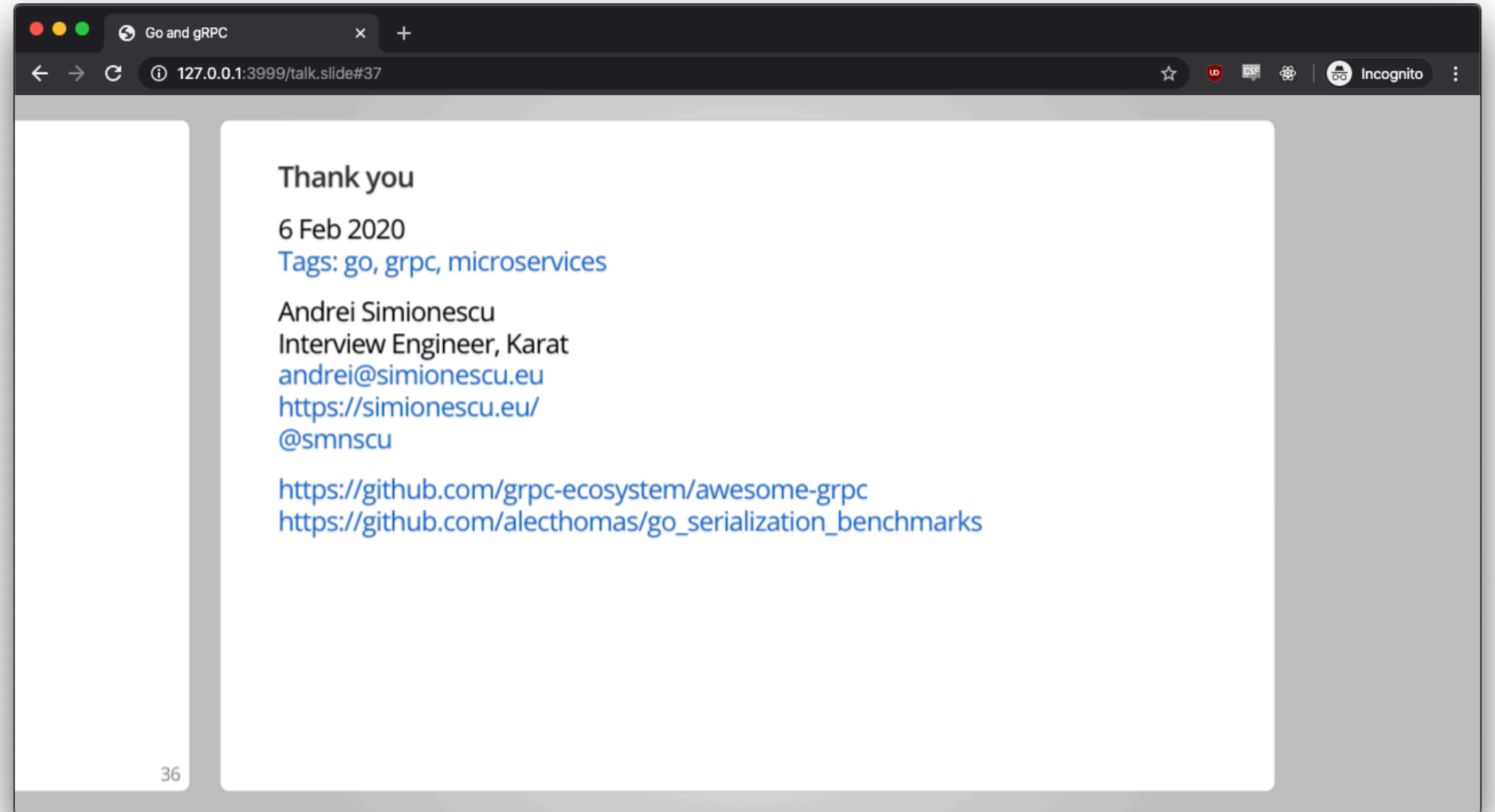


34

Quest


Go and gRPC
127.0.0.1:3999/talk.slide#35
Incognito
34
Questions?
35
TODO
load ba
mocks


Go and gRPC
127.0.0.1:3999/talk.slide#36
Incognito
35
TODO
load balancing
mocks
36
Thank yo
6 Feb 202
Tags: go, g
Andrei Sin
Interview
andrei@si
https://sim
@smnscu
https://gitl
https://gitl


Go and gRPC
127.0.0.1:3999/talk.slide#37
Incognito
Thank you
6 Feb 2020
Tags: go, grpc, microservices
Andrei Simionescu
Interview Engineer, Karat
andrei@simionescu.eu
https://simionescu.eu/
@smnscu
https://github.com/grpc-ecosystem/awesome-grpc
https://github.com/alecthomas/go_serialization_benchmarks
36