Google Protocol Buffers

Me

Anders "Bosse" Carling

CTO & Co-Founder at Tummy Lab

- github.com/anderscarling
- twitter.com/lowe

Agenda

- The What
- The Why (& Not)
- The How
- The Live Hacking
- The Side Notes
- The End

The What

> "A language-neutral, platform-neutral, extensible way of serializing structured data for use in communications protocols, data storage, and more."

A Schema Based Data Serialization Format

- Statically Typed Message
- Backward/Forward compatible
- All fields must have a value
  - Missing fields will get empty value
- Binary and JSON Representations

A Code Generator

- Generate classes representing messages
- No more NSDictionary

A Library with really great platform support

- Google Supports Java, Python, C++, Go, Ruby, JavaNano, Objective-C, C#
- Apple has Swift support in development, but Objective-C implementation imports quite well.

Show me that Schema!

```
syntax = "proto3";

package LibraryAPI;
option objc_class_prefix = "LibraryAPI_";

message Book {
  int32 id = 1;
  string author = 2;
  string name = 3;
}
```

Generated Code

```
@interface LibraryAPI_Book : GPBMessage
@property(nonatomic, readwrite) int32_t id_p;
@property(nonatomic, readwrite, copy, null_resettable) NSString *author;
@property(nonatomic, readwrite, copy, null_resettable) NSString *name;
@end
```

```
@interface GPBMessage : NSObject<NSSecureCoding, NSCopying>
- (nullable instancetype)initWithData:(NSData *)data error:(NSError **)errorPtr;
- (nullable NSData *)data;
...
@end
```

Generated Interface...

```
open class LibraryAPI_Book : GPBMessage {
    open var id_p: Int32
    open var author: String!
    open var name: String!
}
```

```
open class GPBMessage : NSObject, NSSecureCoding, NSCopying {
    public init(data: Data) throws
    open func data() -> Data?
}
```

Parse me a Book!

```
let responseData = httpResponse.data
if let book = try? LibraryAPI_Book(data: responseData) {
    print("Book: \(book.author) - \(book.title)")
}
```

The Why (& Not)

```
BUT JSON!?
```

Schemas

- Provides up to date documentation of data format
- Great place to document quirks and data expectations
  - Comments in `.proto` are included in generated `.h`
- Provides mechanics and reason to think about backwards compatibility

Code Generation

- Less boilerplate
- Less mistakes
- Beautiful Data Parsing

Typed Data

- Richer set of types than JSON
  - Use the right type for your data
- If you only want static typing in one place, cross systems interfaces is a good place.

Compact data representation

- Field names are not sent
- Empty values are not sent

The Why Nots

- Public APIs
  - Not well known enough, and harder to debug.
  - Great news - Use prototbufs at server and serialize as JSON!

The How

```
Wherein I prepare you, and myself, for some live hacking.
```

Server APIs

- `GET /books` returns `Library.Responses.GetBooks`
- `POST /books` accepts one `Library.Book`

Protocol Buffer `library_api/book.proto`

```
syntax = "proto3"; package LibraryAPI;
option objc_class_prefix = "LibraryAPI_";

message Book {
        int32 id = 1;
        string author = 2;
        string title = 3;
}
```

Protocol Buffer `library_api/responses.proto`

```
syntax = "proto3"; package LibraryAPI.Responses;
option objc_class_prefix = "LibraryAPI_Responses_";

import "library_api/book.proto";

message GetBooks {
    repeated Book books = 1;
}
```

Client Data Structure `Book`

```
struct Book {
  let id: Int
  let author: String
  let title: String

  var label:String {
    return "\(author) - \(title)"
  }
}
```

Protobuf parsing with Alamofire

`DataRequest.responseProtobuf<T>`

```
extension DataRequest {
  @discardableResult func responseProtobuf<T: GPBMessage>(
        handler: @escaping (DataResponse<T>) -> Void
  ) -> Self {
    let rs = DataResponseSerializer<T> { req, resp, data, error in
      if let error = error { return .failure(error) }

      let result = Request.serializeResponseData(response: resp,
                                                 data: data, error: nil)
      guard case let .success(validData) = result else {
        return .failure(result.error!)
      }

      do    { return .success(try T(data: validData)) }
      catch { return .failure(error) }
    }
    return response(responseSerializer: rs, completionHandler: handler)
  }
}
```

`GET /books` with `BookAPIClient.load`

```
func load(_ completion: @escaping ([Book]?) -> Void) {
  let req = Alamofire.request(BookAPIClient.url)
  req.responseProtobuf {
    (response: DataResponse<LibraryAPI_Responses_GetBooks>) in

    guard let msg = response.result.value else {
      completion(nil); return
    }

    completion(msg.booksArray
      .flatMap { $0 as? LibraryAPI_Book }
      .map { Book(id: Int($0.id_p),
                  author: $0.author,
                   title: $0.title) })
  }
}
```

Protobuf encoding with Alamofire

`ProtobufEncoding`

```
struct ProtobufEncoding: ParameterEncoding {
  enum EncError: Error { case failure }
  let message: GPBMessage
  var protoname { msg.descriptor()
                    .name.replacingOccurrences(of: "_", with: ".") }

  func encode(_ urlRequest: URLRequestConvertible, with _: Parameters?) throws -> URLRequest {
    var urlRequest = try urlRequest.asURLRequest()
    guard let data = message.data() else { throw EncError.failure }

    urlRequest.httpBody = data
    urlRequest.setValue(protoName,
                        forHTTPHeaderField: "X-Message-Class")
    urlRequest.setValue("application/x-protobuf",
                        forHTTPHeaderField: "Content-Type")
    return urlRequest
  }
}
```

`POST /books` with `BookAPIClient.post`

```
func post(_ book: Book) -> Void {
  let url = BookAPIClient.url
  let enc = ProtobufEncoding(msg: serialize(book))
  let req = Alamofire.request(url, method: .post, encoding: enc)

  req.response {
    if $0.error != nil {
      print("OH NOES! :(:( - \($0)")
    }
  }
}

private func serialize(_ book: Book) -> LibraryAPI_Book {
  let message = LibraryAPI_Book()
  message.id_p = Int32(book.id)
  message.author = book.author
  message.title = book.title
  return message
}
```

The Example App

```
Where I load up the simulator
```

The Live Hacking

Oh noez, the backend is updated!?

```
syntax = "proto3"; package LibraryAPI;
option objc_class_prefix = "LibraryAPI_";

import "google/protobuf/timestamp.proto";

message Author {
        int32 id = 1;
        string name = 4;
        google.protobuf.Timestamp birthday = 3;
}

message Book {
        int32 id = 1;
        reserved 2;
        Author author = 4;
        string title = 3;
}
```

The Side Notes

```
Just stop speaking already!?
```

Protobuf Versions

All examples today used protobuf v3, v2 exists and has a bit more features - like required fields.

There are good reasons for removing them though, like how required fields can never be removed without loosing backwards compitability.

gRPC

gRPC, a RPC API, library and framework from Google, uses Protocol Buffers as the Interface Definition Language.

```
syntax = "proto3";

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string reply_message = 1;
}

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply) {}
  rpc SayHelloAgain (HelloRequest) returns (HelloReply) {}
}
```

Challenges

To keep this meetup thing going, I hereby challenge..

â˝ď¸ **FootballAddicts** to speak about **YapDatabase** and why using it is better than just rolling it old school and build your own B-Tree's (or eh, using CoreData).

đ **MinaTjÄ¤nster** to speak about **ReactNative** and why we haven't put JavaScript in enough places already.

đŚ **A really awesome person** to speak about **Xamarin** and why .NET for mobile development seems quite awesome.

Please note the implied threat about public mockery unless these challenges are accepted.

The End

- github.com/anderscarling/library_api-ios
- github.com/anderscarling/library_api-backend
- github.com/anderscarling/library_api-protobuf
- http://developers.google.com/protocol-buffers

```
EOF
```