# Moving Data

A guide for delivering meaningful data

## Getting Data from Here to There

Purpose: how to think about designing system interfaces

Discussion of data, contracts, and the organizations on either side of the wire

Consider how businesses need to change when designing interfaces

The focus is not to discuss the latest methods for code generation, tools nor languages. Instead the intent is to discuss the underlying principles that guide a given data payload's design in the first place. These principles will be applicable independently of the programming language or chosen data markup standard.

This manual is not a discussion of Integrated Development Environments (IDE’s), the programming language du jour nor any specific markup techniques. It is a discussion about how to approach a design problem that involves moving data, which is a broader topic than traditional ‘Extract, Transform and Load’ (ETL) problems. Data movement from my experience is about 2 way communications between two or more organizations or teams. This necessarily involves traditional ETL as well, including error handling, versioning, selection of the transport mechanism and timing (synchronous, asynchronous, batch). It also involves the tooling that developers need to work in this space, such as contract management tools, data dictionaries, and monitoring systems.

## Data 101

Serialization / Deserialization

Data “On the Wire”;, big/little endian, not the focus of this book, point to resources

Formats for demarking data elements – xml, json, binary, fixed width, comma, tab, space, …

Schemas – xsd, idl, custom

## Interface Types

Schema Embedded

Schemaless

* Order matters
* Order does not matter

|  |  |  |  |
| --- | --- | --- | --- |
| Data Payload Type | Order Matters | Payload Contains Schema | Element Identifiers |
| Flat Files, Delimited (CSV, TSV, Space, …) | Yes | Yes or No | If Yes: column headings can be in the first row  If No: Elements identified by position. External schema maps positions to named parameters. |
| Query string parameters | No | Yes | Expressed as Name/Value pairs. Parameter name is present in payload. Values are typically simple data types. |
| XML, JSON | No | Yes | Parameter name is present in payload. Can contain complex or nested objects. |
| Protocol Buffers | No | No | Expressed as Id/Length/Value. An external schema maps Id’s to named parameters. Can contain complex or nested objects. |

## Versioning

Side by Side Endpoints Vs Multi-version endpoints

Multi-version Requests

Multi-Version Responses

## Call Types

Synchronous: HTTP, TCP, etc

Async: Pub/Sub

Batch: Flat files, Database Tables

Push vs Pull paradigms, when applicable. E.g. for file transfer (log collection)

Push – ideal for many authors that can change rapidly, pushing to a known destination that does not change rapidly

Pull – suited for problem spaces with few or slowly changing authors; requires that pulling system be aware of every author; destination of data is abstracted from the authors

## Data Payload Types

Schema-less Payload vs. Embedded Schema (mostly), order matters, order does not matter

Plain Text, Binary, Compressed/Uncompressed, Streams

## Data Identification

ID’s, Namespaces, Unique Id generators, idempotence, round-tripped id’s (echoed)

Tell me again what you sent? (does each txn have a unique id) [When requesting to create a new order, you don't know the Order number, so how do you debug the request? what is its ID? GUIDs are the accepted means for giving each transaction a unique ID that can be used for locating it in the logs, sharing with the API owner, etc. This value is often sent in a property like RequestId, CorrelationId, TraceId or the like.

If the call is successful then the API owner will supply you with an Order Id from their system that you can use from that point onwards to track related transactions.

Note: this is different than another common paradigm in order systems - external reference number. There is a business need for the caller to add their own Purchase Order number, internal tracking number, accounting department code, cost center, etc. The differnce is in cardinality - the RequestId is unique for every new order Add request that you send. However many orders might contain the same cost center or blanket PO number. The cardinality of RequestId to Request is 1:1, the cardinalty of external reference number to Request is 1:Many, thus these server different purposes and you need both.

## Experiences

Water flows around rock – an exercise in keep it simple

## Error Handling

All-or-Nothing vs Partial failures

Idempotency

# Understanding Data

## Enterprise Data

Customer hierarchies, views of data by geo, line of business, legal entity, country,…

## Accounting Data

Transactions, journals, ledgers, billing documents, cash application, revenue recognition, audit and compliance

## Schemas

[Define schemas, provide examples] A schema provides the blueprint or skeleton of a data payload. It defines property names and data types (string, integer, array, etc). Schemas also can define data hierarchies, for example an Order object can contain a list (array) of OrderItems. [provide example]

Schemas help you deserialize data. They sometimes help you understand the data. Good examples of when they help you understand data are order action codes, e.g. action = create, update, delete. There is no ambiguity what is intended.

## Semantics

[this is the tricky part, it is what the data means] There is no good way to convey meaning in as precise a manner as a schema, i.e. there is not a system that can be interpreted by both a compiler and by humans. Contrast this with schemas, which can be ingested by code and used to programatically deserialize data, all without the developer having to hand type data property names.

## Schema Does Not Always Inform Semantics

[provide examples of permutations on the same schema that have different business meanings. you write different code depending on which values are populated or not]

## Sample Data Payloads

Sample data payloads (SDPs) are canonical examples of payloads that illustrate a specfic business case. These are an essential complement to schemas for understanding data in most systems that i have seen. These are populated data payloads, each with representative data values that demonstrate a given business scenario.

At this point it is reasonable to argue that a well designed API should not need more than its schema to be understood by developers and eventual business consumers of the data on the other side. If you can always enforce that there are no permutations of data that convey a different scenario than the properties can define by themselves then SDPs are not necessary. [do any behaviors require the interaction of 2 or more elements in the schema, i.e. are there interdependencies across elements] Many APIs may start off with each schema element being independently defineable, but as time goes by and subtle changes occur in the business, dependencies across elements creep in. Teams add an optional property to the schema. Older data won't have it, but newer data will. Older data may be given a default value or left null for the new property. This is a typical "non-breaking" change that occurs all the time in business systems, and is a valid scenario. You do not want to ship a full new API version for the addition of a single field or two. However as a few of these additions get accumulated, their interactions can matter. Under which circumstances are they populated? What business scenario required them to be added?

Perhaps you added an extended tax amount to support orders in the new country that you now support. Local orders will not contain that code, but orders for country XYZ will require that value. You are not going to stand up a dedicated Order endpoint for just that one country, so you add it to the existing one. Business logic does the enforcement:

If (order.Country == "XYZ")

{

order.ExtendedTaxAmount =

TaxModule.ComputeVatTax(order.TotalAmount);

}

While perhaps a trivial example, it illustrates the key point: a developer that consumes data from this API must write different logic based on the value of given property, order.Country. The schema will only say that this property is a datatype string, and that it contains a valid country code. It is silent on the use of this property for tax computations. Utterly silent. Not a peep. In my experience this is the rule, not the exception. Any mature business API will have 'special case' logic to handle important scenarios, and the intent (the semantic meaning) is not easily captured in a schema.

In the next release you might refactor the schema to have a required order property such as TaxType. All orders will have a TaxType and it will be obvious from the schema to any developer that they must have logic to deal with the different values in the TaxType range of values. In fact this would be a good property to have added to your public API in addition to the ExtendedTaxAmount. The difference is that the tax value is optional, TaxType is required. Customers using the existing API will not send it. We cannot require customers to change code for a non-breaking change, so we must either write code that populates an intelligent default (if possible), or wait until the next release of the API.

In the event that you can populate intelligent default values then you can add it to your private schema and enforce that all orders contain this value prior to saving them. Later you can add it to your public schema in its next major release (when breaking changes are allowed). This is another good reason to keep private schemas separate from public schemas - it is essential for business agility that they can change at different velocities.

## Public Vs Private Schemas

[explain what they are, why to keep separate]

## Cookbook

Querystring params – risks of logging entire querystring; same architecture paradigm as pass through payload; can wind up with downstream systems depending on parameters you don’t know exist;

Co-authored Querystrings – on callbacks that require the customer to form the URL and that also allow for customer-defined parameters (e.g. conversion tracking) there is risk of parameter collision. &CID = Customer ID, Client ID, Contract ID, Campaign ID, Click ID, … Options to solve: encrypt your params in a blob that must be round-tripped; define ‘system’ params that the customer cannot use; require ordering – list my &CID first (or last) and parse in that order (hokey, but we once resolved a live site this way)

# Managing Data

## Towards Organizational Intelligence

Respositories Vs. Tribal Knowledge

## Building and Maintaining Data Repositories

Repositories are required to help an organization scale. Repositories take essential facts out of the realm of ‘tribal knowledge’ and put them into a discoverable place. Tribal knowledge will always exist in some form, however the key elements required to build new interfaces and debug existing data in production should be discoverable. This will aid both new hires and the DRI in the middle of the night.

Capturing data into repositories should be done as a by-product of the developer’s normal job. This should not require duplicating data into a form or some other system that can go stale. The data contracts that developers must write to make their code work should also contain the data necessary to populate the repository. Code check-ins should [allow for another process to catpure them and reflect that into the repository].

# Appendix

## >>Flatbuffers

I don’t get it either.  Seems like the definition is that the in-memory format is the same as the wire format.  So, that burdens the app with overheads when it works with the data?  Sometimes that tradeoff might make sense, but that would not be generally true.  I would be surprised if it was even commonly true.

What am I overlooking here?

Some previous discussions attached.

More useful comparison from Cap’n’Proto guy here: <https://kentonv.github.io/capnproto/news/2014-06-17-capnproto-flatbuffers-sbe.html>

As I said I don’t really get this. Are you supposed to move your data between internal structures and the flat buffer whenever you send/receive data? Doesn’t that de facto amount to manual de/serialization?

Here is a "newly" announced Google project:

<http://google.github.io/flatbuffers/md__white_paper.html>

This is a somewhat interesting debate between current and former protobuf developers: ​

<https://news.ycombinator.com/item?id=7902445>

## .Net Core Article

<http://blogs.msdn.com/b/dotnet/archive/2014/12/04/introducing-net-core.aspx>

Unfortunately, we’ve also learned that even compatible changes can break applications. Let me provide a few examples:

* **Adding an interface** to an existing type can break applications because it might interfere with how the type is being serialized.
* **Adding an overload** to a method that previously didn’t had any overloads can break reflection consumers that never handled finding more than one method.
* **Renaming an internal type** can break applications if the type name was surfaced via a ToString() method.

## Semantic Versioning

<http://semver.org/>

* Web-scale systems must provide self-defining (and versioned) objects. In the case of SDS, self-defining disk formats with ability to encode and serialize structured data in efficient yet extensible formats, like protobuf, Avro, et al. This way, upgrades of disk data can be done lazily. Web-scale cannot assume a one-shot data upgrade, given the scale.
* Web-scale systems should have self-describing (and version-aware) services such that different parts of the distributed system can communicate at different versions, without expecting a one-shot upgrade for all components.

<http://www.nutanix.com/blog/2014/03/11/understanding-web-scale-properties/>