How do you handle change in an application?

- eg if a new field needs to be added to a data model?
  - relational db with schema-on-write -- need to alter the schema
  - document db with schema-on-read -- can handle on reads

Migrations

- server side -- rolling upgrade?
  - can deploy without downtime
  - 'backend'?
- client side -- at the mercy of the user
  - 'frontend'?
- ideally need bacward and forward compat
  - backward is easy enough: as the developer you know what came already
    - newer code can read data written by older code
  - but forward looking?
    - older code can read data written by newer code

The question becomes: how do you encode data?
Several format options here

- JSON
- XML
- Protocol buffers
- Thrift
- Avro

how do these formats

- handle schema changes?
- support systems with old and new data and code need to coexist?

how are these formats used for

- data storage
- communication
  - REST
  - RPC
  - message-passing systems

Formats for encoding data

Programs usually work with data in (at least) two reresentations:
1. in memory: data is kept as objects, structs, lists, etc. optimized for efficient access and manipulation by CPU, typically using pointers
2. when you want to write data to a file or send it over the network, you nede to encode it as some kind of self-contained sequence of bytes

eg: a JSON document

a pointer would not make sense to any other process -- so this representaton needs to be self-contained and is typically quite different from an in-memory representation

We need some kind of translation between the two representations.

- from in-memory to byte sequence = encoding or serialization
- from byte sequence to in-memory = decoding or deserialization or parsing

This is a very common problem so there are many solutions available.

Language specific formats

java = Serializable
python = pickle
etc
however the trouble with these is:

- the encoding is tied to a particular programming language, and reading the data in another language is difficult
- to restore data in the same object types, the decoding process needs to the able to instantiate arbitrary classes, which can be a source of security problems
  - if an attacker can get your appllication to decode an arbitrary bute sequence, they can instantiate arbitrary classes, which may allow them to execute arbitrary code remotely
- versioning data is usually an afterthought with these libraries -- the priority is quick and easy encoding in the programming langauge
- efficiency is also an afterthought

Standardizing encodings: JSON, XML, Binary variants

JSON, XML, CSV are obvious contenders for a standardized encoding format
But there are issues

- ambiguity around encoding numbers
- no support for binary strings in JSON and XML
  - sequences of bytes without a character encoding
- clients that consume XML/JSON need to hardcode the encoding/decoding logic to infer the schema present
- Csv on the other hand has no schema, so the application has to interpret every row and column

That being said, as long as people agree on a format, these are good for purpose as data interchange formats -- i.e. for sending data from one organization to another

Binary encoding for JSON -- MessagePack -- reduces the size of the JSON blob but is still bloated relative to other options

Binary encodings

Thrift and Protocol Buffers

Apache Thrift (oroginally facebook) and Protocol Buffers (from Google) are binary encoding libraries that are based on the same principle

Both require a schema defn, which can be specified. Each comes with a code generation tool that takes a schema defn and produces classes that implement the schema in various programming languages.

An application can then call this generated code to encode or decode records of the schema.

These protocols have their own way of maintaining forward and backward compat -- using field tags

- old code that encouters a field tag i doenst know about just ignores the field
  - a datatype annotation allows the parser to know how many bytes to skip
- same applies for removing a field
  - can only remove optional fields
- however, changing a datatype for a field may be tricky
  - lose precision?

Avro

Apache Avro is another binary encoding format, that is different from Thrift and Protobuf
See book for more details on all these

Several nice properties of binary encodings

- more compact that 'binary JSON' variants
- schema is valuable form of documentation, and the canonical schema is guarateneed to be up to date since it is used for decoding
- for users of statically typed languages, the ability to generate code from the schema is useful since it enables compile-time type checking

Models of Dataflow

At the beginning, we said that whenever you want to send some data to a process with which you dont share memory -- for example, when you want to send data over the network or write to a file -- you need to encode the data as a sequence of bytes. We then discussed different ways of doing this and concerns around fwd/backward compat.

Compatibility is a relationship betwen one process that encodes the data, and another

process that decodes the data.

But this is fairly abstract. There are many ways data can flow from one process to another. For example:
- via a database -- dataflow through databases
- via service calls -- dataflow through services
    - via async message passing

Dataflow through databases
In a database
- the process that writes to the db encodes the data
- the process that reads from the db decodes the data

Typical flow is
- one process writes encoded data
- another process reads it sometime in the future

There may be a single process accessing the db, in which case storing something in the db is simply sending a message to your future self.
- so backward compat is clearly necessary -- you need to be able to read what you wrote in the past!

But in general, you may have different processes accessing a db at the same time. These processes may be different applications or services, or several instances of the same service.
And it is likely that some processes accessing the db will be running new code (if the application has been upgraded somewhere) and some older code (perhaps because said instance is behind in a rolling upgrade)
- so a value in db may be written by a newer version of the code but needs to be read by an older version of the code
- so forward compat is also necessary

One anti pattern is here:
- new code adds a new field to schema
- old code that reads the new schema simply skips the new field it is not aware of
- old code then writes new data sans the new field to the db
- now the new field is lost
-- this is not a hard problem to solve but you need to be aware of it!

Dataflow through services: REST and RPC

When processes need to communicated over a network, there are different ways to arrange the communication.

The most common is to have two roles: *clients* and *servers*

- A server exposes an API over the network
- A client can connect to the server to make requests to that API
- the API exposed by the server is known as a *service*

Typical flow is

- one process sends a request over a network
- the same process epxects a response as quickly as possible

A web browser is not the only type of client!

- a native app running on a mobile device or desktop can make network requests to a server
- a client side js application runnign inside a web browser can use XMLHttpRequest to become an HTTP client -- this is known as *Ajax*
  - in this case the server's response is typically not HTML for dispalying to a human, but rather data in an encoding that is convenient for further processing by the client side application code (such as JSON)
- HTTP can be used as the transport protocol
  - it is a protocol for fetching/setting data?
- but the API implemented on top is application specific, so the client and server need to agree on the details of that API
- moreover, a server can itself be a client to another service
  - for eg a web app server is a client to a database
- This approach is used to decompose a large application into smaller services by area of functionality such that one service makes a request to another when it requires some functionality or data from that other service
  - this is considered a *service oriented architecture* or a *microservices architecture*

In some ways services are similar to databases

- they typially allow clients to submit and query data

However

- dbs allow arbitary queries using a query language
- services expose an application-specific API that only allows inputs and outputs that are predetermined by the business logic of the service
  - this restriction provides a degree of encapsulation -- services can impose fine grained restrictions on what clients can and cannot do

A key design goal of microservices architectures is to make the application easier to change and maintain by making services independently deployable and evolvable

- each service is typically owned by one team that can release new versions of the service without relying on/coordinating with other teams

- In otehr words, we should expect old and new versions of servers and clients to be running at the same time -- so the data encoding used by servers and clients must be compatible across verions of the service API

Web services

When HTTP is used as the underlying protocol for talking to the service, it is called a *web service*

There are two popular approaches to web services: REST and SOAP. They are almost diametrically opposed in terms of philosophy

REST is not a protocal but a design philosophy

- emphasizes simple data formats, using URLS for identifying resources, and usting HTTP features for cache control, authentication, and content type negotiation

SOAP is an XML based protocol for making network API requests

- A SOAP API is described using an XML based language called the WSDL

RPCs

- an older paradigm for making API requests over a network
- an idea that tries to make a request to a remote netwrok service look the same as calling a function or method in a programming language, within the same process -- ie make a remote service look like a local object in the programming language
- this is a flawed idea for many reasons
  - network calls can fail unlike local function calls
  - they may not return anything because of network loss and unless they are idempotent repeated calls may yield bad results
  - packaging all the data needed for an RPC into a byte sequence can be gnarly when there is enough data (unlike when the refefnce to the object is within the same process, when you just use pointers in local memory)
  - the RPC may have to translate data types between programming languages since the client and server may ise different languages

Data encoding

- unlike with data flowing through databases, we can make a simplifying assumption for data flowing through services: servers get upgraded before clients
  - why?
- So you need
  - backward compat for requests
  - forward compat for responses

- RESTful APIs typially used JSON (without a formally specified schema) for responses, and JSON or URI encoded request parameters for requests
    - adding optional request params and adding new fields to responses are usually considered changes that maintain compatibility

Message passing

- somewhere between databases and RPC
- similar to RPC ins that a client's request (a *message)* is delivered to another process with low latency
- similar to dbs in that the message is not sent via a direct network connection, but goes via an intermediary called a *message broker/message queue*
- asynchronous

Advantages

- queue is a buffer if the recipient is unavailable or overloaded
- queue can redeliver messages to a process that has crashed
- it avoids the sender needing to know the IP address of port number of the recipient
    - useful in cloud deployments where VMs come and go
- allows one message to be sent to many recipients
- logically decouples sender from recipiet
    - sender doenst care who consumes the messages it publishes
    - and it does not expect a response to its message

Typical flow

- a process sends a message to a *named queue* or *topic*
- the message brokert ensures that the message is delivered to one or more *consumers or subscribers* to that queue or topic
- one way dataflow
    - but you can set up a reply queue that the producer subscribes to

Apache Kafka, TIBCO