Designing Messages

Patterns and Pitfalls

Hi I’m Richard

I help teams deliver software as a coach/consultant

My background is in designing and building distributed systems

I’ve been doing this for about a decade now

I’m @richardw1001 on Twitter

I write articles at http://richardwellum.com

Email me at richard@richardwellum.com

Why do we need Messages?

… to facilitate collaboration with minimal coupling

Types of Messages

Commands, Queries and Events

Queries

Queries are a request for information
    Named as a request

Sent from anywhere to single logical owner

Can be rejected – or owner can be offline/unavailable

Will contain information in a response

Highest coupling between sender and owner

Commands

Commands are a request to do something
  Named in the imperative

Sent from anywhere to single logical owner

Can be rejected

May include information in a response – but mainly about status

High coupling between sender and owner

Events

Events describe something that has happened
  Named in the past tense

Published from single owner to many subscribers

Cannot (semantically) be rejected by subscribers

Contain information describing what happened

Lowest coupling

Messaging Options

Transport type, Persistence, Transactions, Versioning

Request/response

Most common example is HTTP

Point to point communication

Request, containing data, receives a response, containing data

Great for queries

- The response contains the data you asked for

Great for commands

- You know from the response what happened

Publish/subscribe

Commonly seen in bus/brokers

Copies of messages are distributed to each subscriber

Can be used for events

- But messages only go to subscribers who already subscribed

Fire-and-forget

For example sending a command via MSMQ

Message is acknowledged but there is no response

Can be used for commands

- But only if senders don’t need to know the result of the command

Streams

For example AWS Kinesis, or Kafka

Events are persisted

Multiple consumers read the stream, with their own pointer to last read message

Great for events

- New consumers can replay the stream from the start to catch up

Persistence

Persistence in the delivery adds resilience

- Receiver can be offline - but this means there is no guarantee of a reply
- Receiver can be under load – reply may have high latency but the message is safe
- Transient errors can be retried – if the handler is idempotent or rolls back on failure

Read the small print around delivery guarantees

- At Least Once may mean messages are delivered multiple times
- At Most Once may mean messages are not delivered
- Is order of delivery guaranteed? Even if there are retries?

Messages as transaction boundaries

Treat each message as 1 transaction

Use internal commands as roll-back points

Be wary of chaining

Versioning

Treat messages as versioned

Use semantic versioning

Will you accept older (or multiple) versions of messages?

Will you issue older (or multiple) versions of events?

Patterns and Pitfalls

… learned mostly the hard way

Microservice Envy

No Messages (database level integration)

Messages as method calls

Messages as the boundary

Leaky messages

Messages made to be shared

Too many commands

Event driven flow

Subscriber agnostic publishers

Commands disguised as events

Chatty dialogues

Entity CRUD messages

Uninformative events (with back-queries)

Pointers to large (versioned!) payloads

Pointers to sensitive (versioned!) payloads

Command + status query for long-running job

Idempotent handlers

Retry policies

Concurrency assumptions (consider Actors)

Messages treated as infrastructure

Messages treated as part of domain

Heavy-handed conventions

No conventions

Lightweight conventions

Summing up…

Concepts > Technologies

If you model the domain badly, technologies won’t save you

If you model the domain well, you can swap out technologies later

Use the right type of message

Events for lowest coupling

Commands and Queries when needed

Make sure that the value added by commands and queries is more than the cost of the coupling

Use the right transport options

My defaults are HTTP for commands and queries, and streams for events

If this isn’t available, publish/subscribe for events is ok

Fire-and-forget commands are not ok!

Be deliberate about what you need from the infrastructure and honest about whether it fits

Design for change

Expect the system to evolve

Expect different parts of the system to evolve at different rates

This means versioning and rolling update matter!

Design for cases like adding new services, retiring services, or updating individual services with minimal disruption

Listen to feedback

As you progress, are things getting easier or harder?

What lessons can you learn as you go along?

It’s likely that the initial design of service boundaries will benefit from tweaking

I’m happy to answer questions

Come find me and say hi!

Hi I’m Richard

I help teams deliver software as a coach/consultant

My background is in designing and building distributed systems

I’ve been doing this for about a decade now

I’m @richardw1001 on Twitter

I write articles at http://richardwellum.com

Email me at richard@richardwellum.com