Scalable architecture from the ground up

Oren Eini

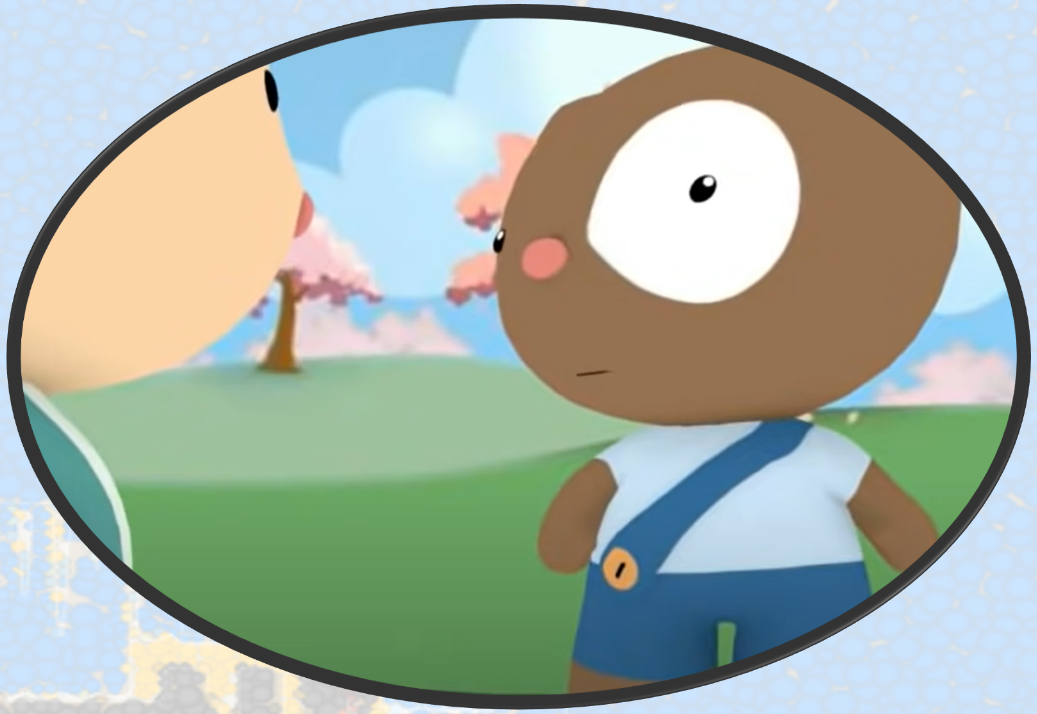
oren@ravendb.net

What is scalable?

- A service is said to be scalable if when we increase the resources in a system, it results in increased performance in a manner proportional to resources added.

  All Things Distributed – 2006

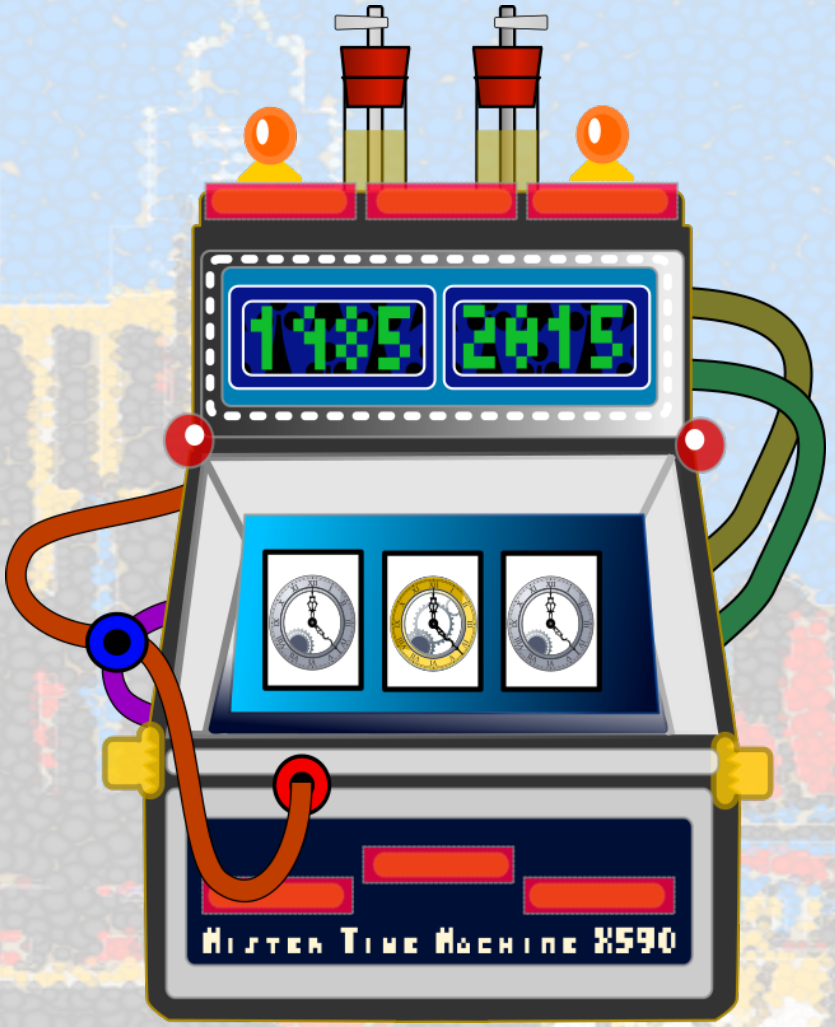
- https://www.allthingsdistributed.com/2006/03/a_word_on_scalability.html



How to build a scalable system?

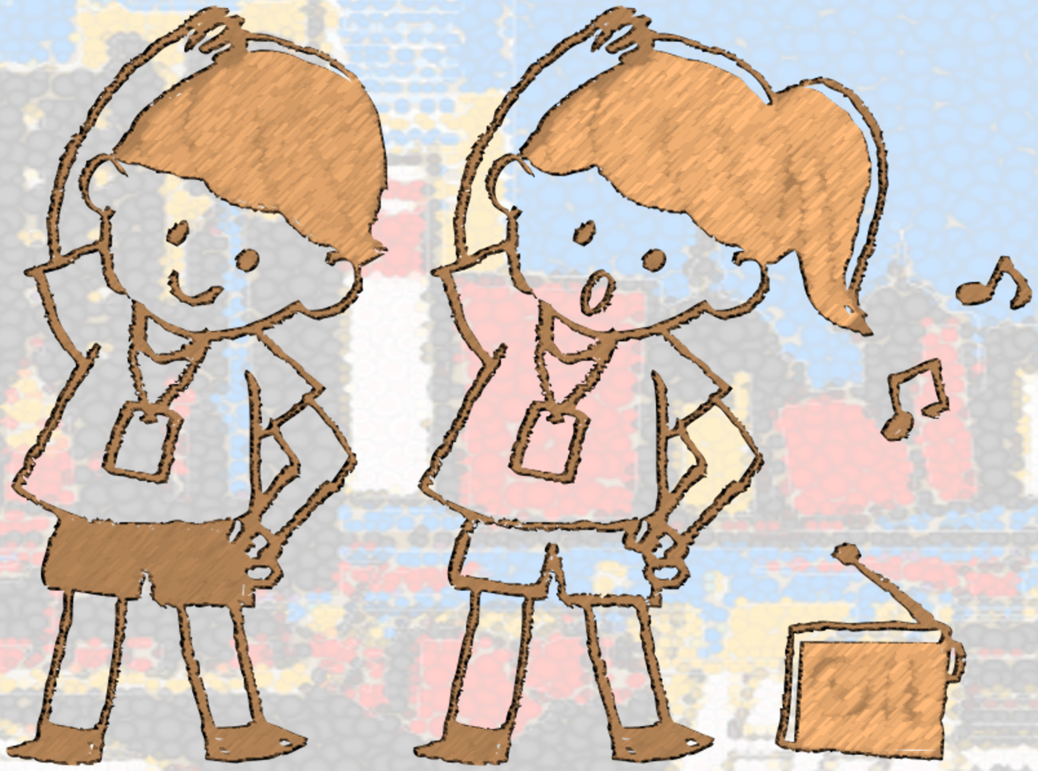
1. Invent time machine
2. Deploy service
3. Learn from your mistakes
4. Go to #2

5. TimeMachineUnavailableException…



Without a time machine?

- Learn from other people's experiences
  - Others' pain, your gain
- Your scenario is *different*
- Things change over time
  - Twitter features:
    #hash, @mentions were created by users,
    then adopted by Twitter

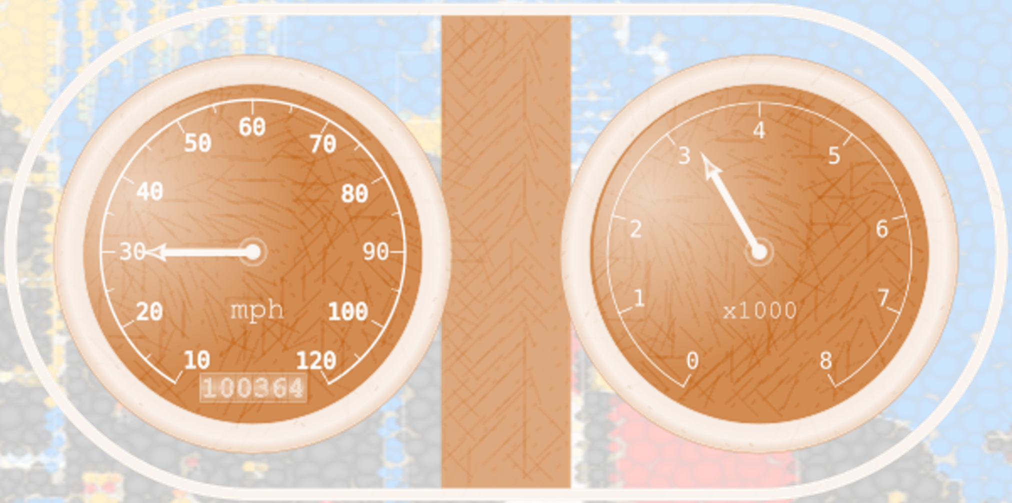


Architecture styles

- Domain Driven Design - DDD
- Command Query Responsibility Separation – CQRS
- Buzzword of the day – BOTD
- Beware of complexity

#1 rule: What is your SLA?

- Page should load in < 200ms for 99.99% of users
- **Metrics**
- *Ongoing* feedback



#2 rule: Don’t do what you can’t guarantee to meet the SLA

- Don’t write checks that you can’t cash
- How do you implement features, then?

Being reckless with promises…

- Doing work in the critical path will kill your system

- Render a page
- Process a request

- You have ~200ms time budget
- What is involved?

- Expensive queries
- Dynamic content
- 3rd party services

- Can you track across all of those?

Paying promised with a payment plan

What is cheap?

- Get data by key
- Query by simple index
- Local queries
- Bounded amount of data / work
- Put in queue

What is expensive?

- Complex queries
- 3rd party remote calls
- Processing *lots* of data

My father told me:

Buy low, sell high, be rich

Don’t do the expensive things (right now)

- Cheap: Put in queue
  - Trivial to scale
  - Available everywhere
- Separate processing of items in queue
- Online operations operate over prepared data

- CQRS, but for speed / latency

- UX concept, we don’t do things *right now.*
- Accept & process in backend
- Frontend does little
- Prepared in advanced

Why not auto scale?

- When web server hits 75% CPU utilization > 15 seconds
  - Spin new node
  - Rebalance traffic
  - See latencies go down
- When web server hits < 25% CPU utilization > 30 seconds
  - Rebalance traffic
  - Shutdown node
- Basically available everywhere

Great idea, if cost is in serving requests

- Typical limiting factor is backend operations
- Complex queries at backend, you'll need to scale your database
  - Or add (non trivial) caching
- 3rd parties operations, you are going to *wait* on those
- Other side is a human, there is a timer (their patience)
  - Major impact on revenue, even for 100ms additional latency

- Can you do this across the board?
  - Including databases, backends, 3rd parties?

With a queue, you have *time*

- Put in queue, the backend will process
- Frontend registers acceptance, show in the UI
  - Operation done for user (even if not completed) – show as such immediately.
  - Operation accepted (will update when completed)

- Process message in queue
- If queue drain rate not enough
  - Add more workers
  - Same as auto scale?
- Temporary spikes are smoothed
- Log all queue messages long term?
  - Note: Privacy!

With a queue, you got *options*

- Don’t process one message at a time, use *batches*
  - Grab a bunch of messages
  - Load all related data upfront
  - Reduce per message costs
- Don’t use *a* queue, use many
  - Buying from Brazil?
    - ***/Q/Purchases/Brazil*** - Got a worker just for that
  - Similar behavior and patterns
- Don’t need to scale out as often
  - Spikes are smoothed
- Message processing code is simpler (no UI concerns, pure computation)
- No expected shared state to deal with (session, cache, etc)

To queue, or to fail, there is no try

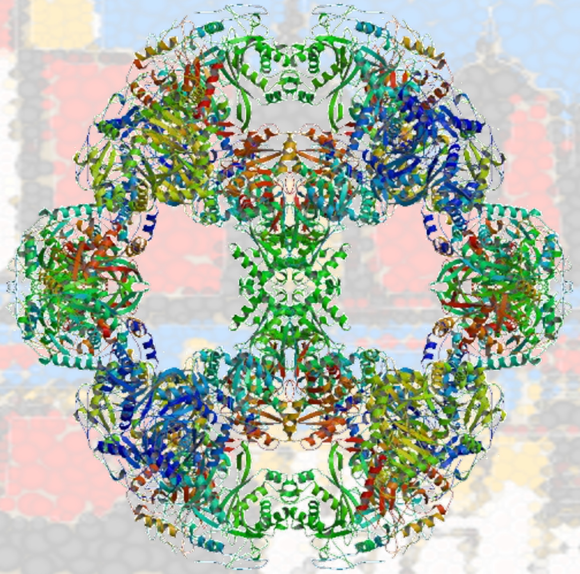
- Retries are simple
- Debugging is trivial (you get the input ☺).
- Operations are simple
  - Need to upgrade
  - Spin a new worker with new code
  - Shut down old worker
  - Zero downtime
- Queue are operationally simple to operate

- Metrics:
  - Time in queue
  - Drain rate
  - Number of messages
- What is going on right now?
  - Peek to the queue…

What about the front end?

- Pull prepared data
- Separated read mostly portion
  - Some *cheap* operations can be done inline.
- Mostly concerned with presentation
- Actions will typically put in queue
- **UX challenge!**

- Explicitly deny complex operations inline
  - Reports
  - Dashboards
- Do them in the background



Architecture superpower: Let’s wait…

- Most operations are trivial CRUD
  - Can do synchronous operation from frontend all the way to the end
- The key aspects and common operations? Complex
  - Don’t try to do them directly
- It’s okay to make the user wait
  - Not on an active request
- We got your request, we’ll process it
- What is reasonable?
  - If < 10 seconds, users won’t care
  - Okay to slip timeline if they aren’t actively waiting for us

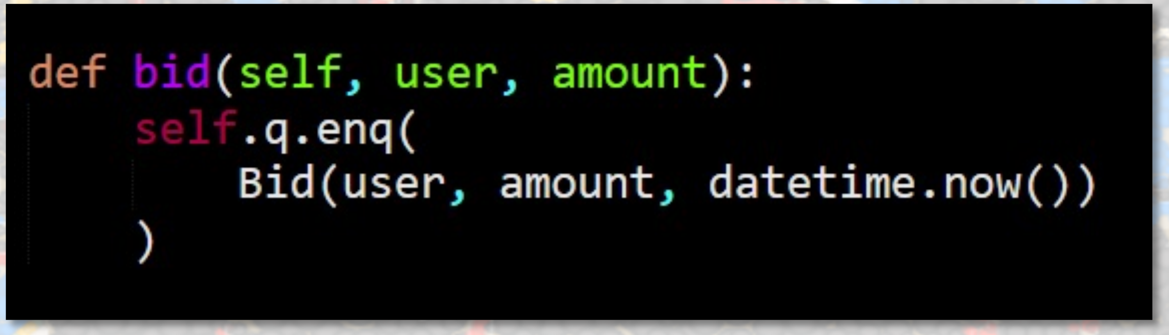
Architecture super power: Saying sorry

- I’m selling TVs
- Black Friday sale
- Lots of users want to buy
- What to do?

- Accept all orders
- Mark them as accepted
- Process them
- If run out of inventory?
    - Send “sorry, do you want to buy this other model”?
    - Lost the deal, no money down
- Really powerful technique
- **Dramatically** *simplify* apporach

Architecture superpower: Accepted vs. Confirmed

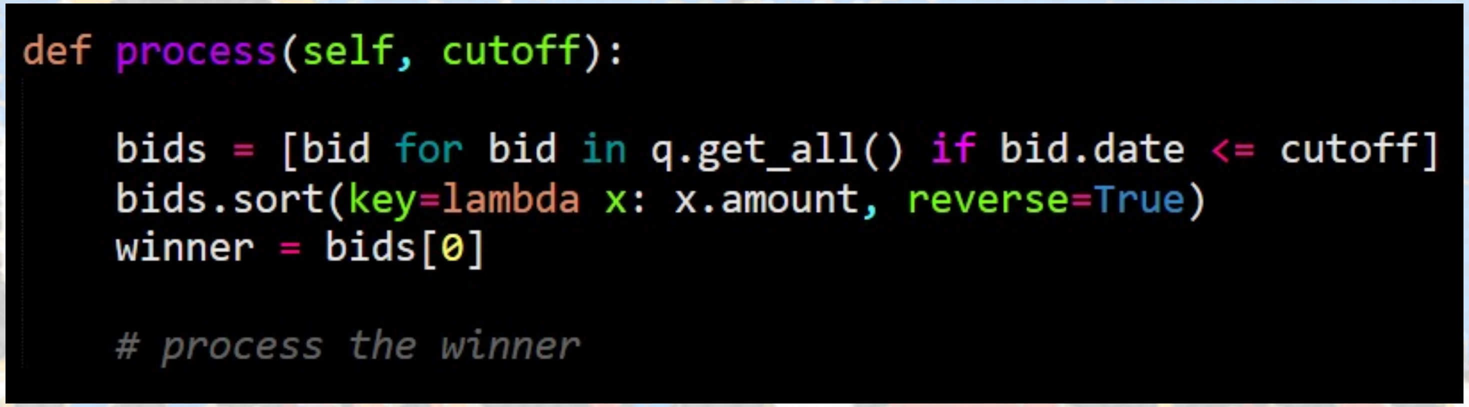
- Selling the Mona Lisa
    - One item
- Auction style
- High concurrency
- Only single winner



```
def bid(self, user, amount):
    self.q.enq(
        Bid(user, amount, datetime.now())
    )
```

Processing the sale…

- No need to deal with concurrency
- Audit log if contested
- Complex business rules?
  - No problem!



```python
def process(self, cutoff):

    bids = [bid for bid in q.get_all() if bid.date <= cutoff]
    bids.sort(key=lambda x: x.amount, reverse=True)
    winner = bids[0]

    # process the winner
```

Architecture superpower: Being stupid

- It’s easy to scale stupidly simple architecture
- Very possible to create sophisticated solutions on simple concepts
- **Resist** adding complexity
  - Compounding costs
- Background processing can do amazing things.
- Copy & paste solution for most issues (backend chew the solution the frontend)
- Frontend is simple
- Infrastructure handles complexity

Questions?