

O'REILLY®

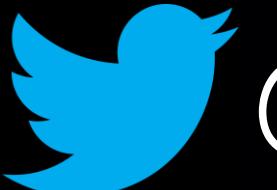
Velocity

CONFERENCE

BUILD RESILIENT SYSTEMS AT SCALE

Scalable Systems Design

velocityconf.com
#velocityconf

大胡子  @postwait



What does it mean to be scalable?

- A system is scalable when its design does not require change to service more or less

#velocityconf

O'REILLY®
Velocity

Do the right things...

- Use the right database.
- Use the right languages.
- Optimize and cache aggressively.
- Test stuff.
- Monitor stuff.

#velocityconf

O'REILLY®
Velocity

Identify scaling challenges

- Look at workloads or data sizes that
 - will not fit on a single machine
 - expect a growth larger than 26% CAGR

Identify scaling challenges

- Look at code bases that might not scale
 - rapid change,
 - poorly understood requirements or constraints

Isolation

- Isolate code, workloads, and data issues you have deemed “challenges.”
 - keep scalability challenges separate from non-challenges
 - keep different scalability challenges separated by constraints and requirements
 - availability, data safety, performance
 - focus on protocols first,
 - then focus on APIs,
 - then focus on implementations.

Microservices to the rescue?

- It would seem that microservices address this concern quite well:
 - Separate design decisions per service
 - Isolated by protocol (and API)

Microservices to the rescue? ... not so fast.

- The problem is the “micro” part (at least sometimes):
 - Small isn’t necessarily a good thing (or a bad thing)
 - It does not intrinsically align with “challenges”
 - Many challenges are highly specific (well suited to microservices)
 - Those things that aren’t challenges are not, together, micro.

Constraints?

- The world of possible constraints is big.
- What prevents you from just building your service as a monolith?
 - Apache or nginx
 - PHP or Java or Node.js or Go
 - MySQL or PostgreSQL
- Many of your constraints surface in answering that question?
- Rarely is the technology a valid constraint (it does happen).

Truths... Availability

- Some components require higher availability than others.
 - (login vs. search)
 - (inventory vs. purchasing)
 - (user targeting and suggestions vs. content)

Truths... Integrity

- Some components have different data integrity requirements
 - (auth/access control vs logging)
 - (canonical vs. non-canonical services)
 - (session state vs. persistent state)

Pitfalls... Data Storage

- Data stores are pretty tricky:
 - Each data source technology requires unique expertise
 - Development
 - Operations
 - Consider data model stability
 - Consider failover/high-availability
 - Consider backup and recovery
- Often micro services (to be isolated) require their own data stores.

Pitfalls... API Lifetime

- APIs have a much longer life than you think.
- Once you expose services via APIs over the network.
- It is often useful to have an API review board that spans teams.
 - It is really important to get APIs right.
 - You will still get them wrong and incomplete.
- Also track usage of API and usage by source or user.
 - When considering to deprecate APIs this information is vital.
- Be careful about “fixing bugs” as users often rely on them.

Pitfalls... Networks Add Complexity

- Networked services dramatically increase the likelihood of latent service interactions
 - Not mildly latent, but dysfunctionally latent (apparently byzantine)
- Circuit breakers can be used to protect service integration points...
 - No service should be depended upon without a well controlled timeout.
 - Timeouts should always have three tunables
 - Connect timeout (low, near zero)
 - Initiation timeout (low)
 - often the same as the completion timeout for web services
 - But, if SSL is on, this could be a SSL handshake timeout
 - Completion timeout (medium and often adaptive)

Service Isolation is not QoS Isolation

- Service isolation protects one service from another.
- Usually all services (or most) are involved in delivering the user's experience.
- Continuous deployment results in
 - Decreased risk of malfunction on deployment
 - Increased risk of overall service malfunction due to increased deployment frequency
- Isolation can be done via swim lanes

Swim lanes... at different levels

- Ultimately, swim lanes are designed to:
 - Prevent a single service outage from harming all customers
 - The abuse by one customer impacting all other customers
- It provides an sloppy, but effective, isolation of quality of services.

Swim lanes... Style #1: most isolated

- The entire architecture is duplicated and customers can live in only one swim lane
 - Separate!
 - services
 - storage
 - networking
 - They cannot be simply diverted to a separate swim lane; they must be migrated.
- This is obviously difficult for social sites, billing, and authentication.
- Often adds significant operational overhead costs.

Swim lanes... Style #2: service isolated

- Each services within the environment has multiple swim lanes.
 - Stateless services (no data store) can allow simply diversion of users to new lanes
 - This is often used to implement canaries and safe “develop in production” environments.
 - Beta or pre-release code is launched in a swim lane with no users;
 - Users are moved into the swim lane for testing;
 - Other swim lanes are upgraded;
 - Users are evacuated from the canarie lanes and the lanes destroyed.
- Allows for some services to have swim lanes and others to be operated traditionally.

O'REILLY®

Velocity

CONFERENCE

BUILD RESILIENT SYSTEMS AT SCALE



Thank You

Never build it bigger than you have to.

velocityconf.com

#velocityconf

O'REILLY®

Velocity

CONFERENCE

BUILD RESILIENT SYSTEMS AT SCALE

velocityconf.com

#velocityconf

#velocityconf

O'REILLY®
Velocity

