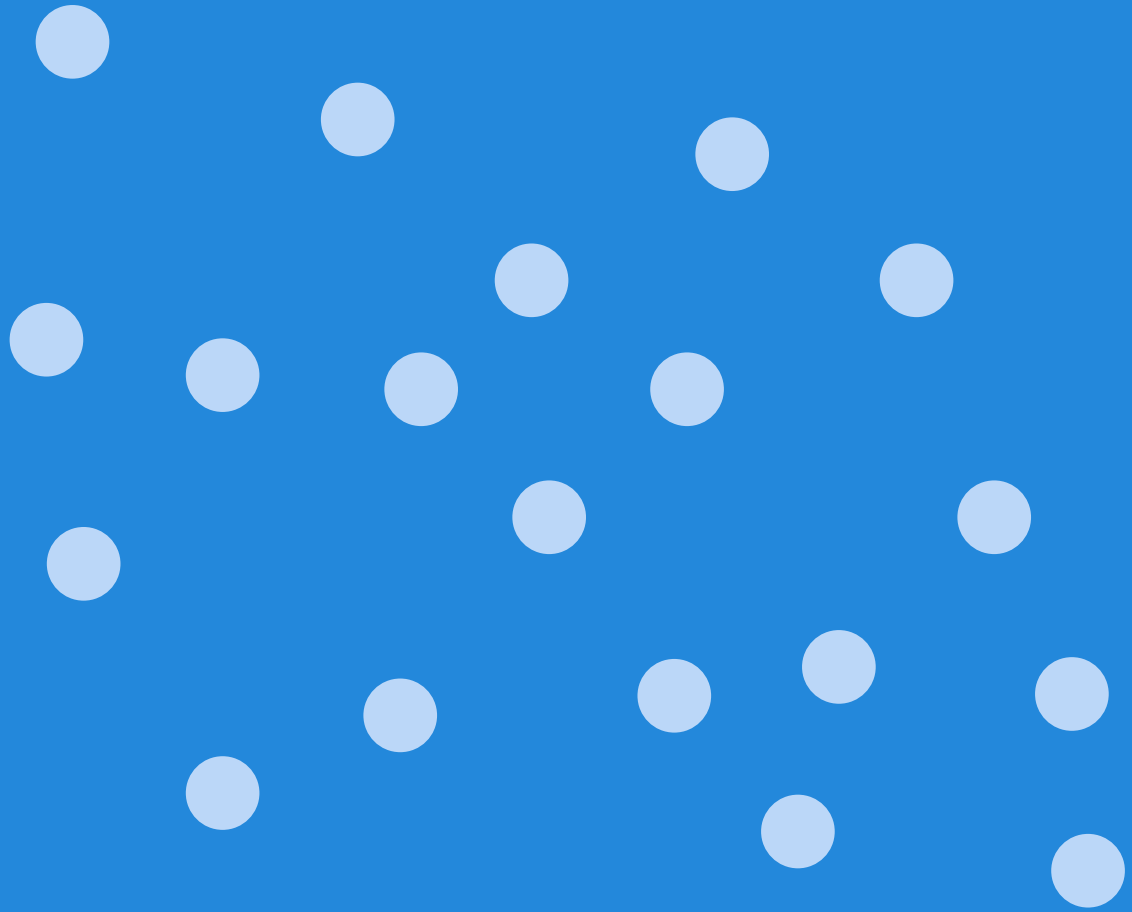
From a monolith to microservices + REST

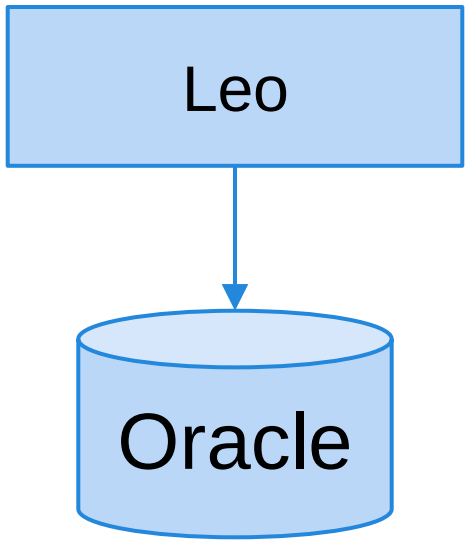
The evolution of LinkedIn's service architecture

by Steven Ihde and Karan Parikh (LinkedIn)



Leo

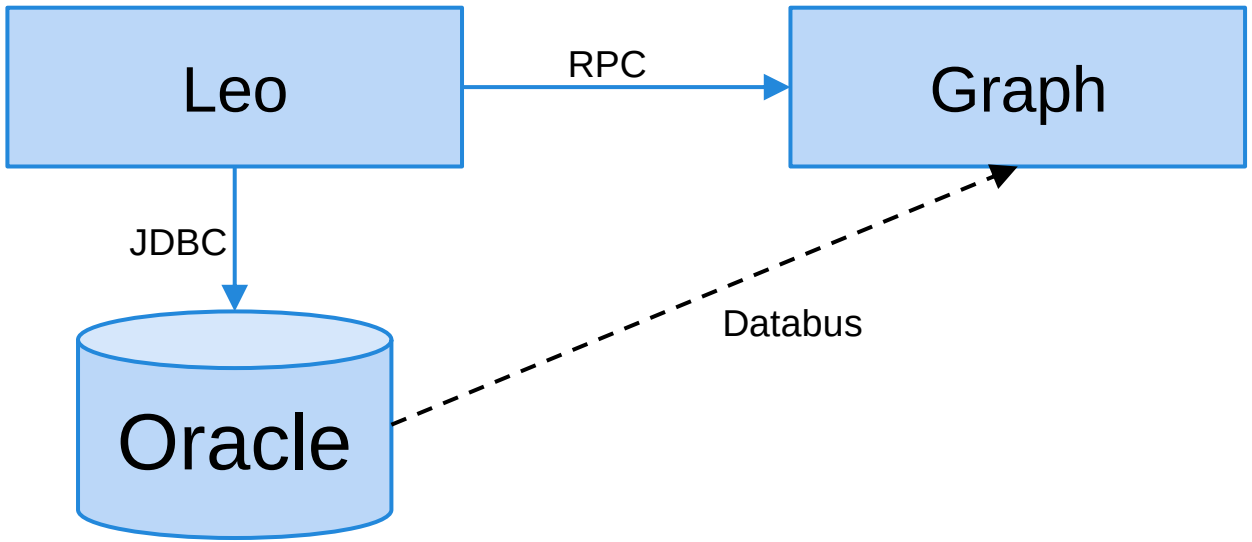
- Our original codebase
- Java, Servlets, JSP, JDBC



Remote Graph

- Graph: member-to-member connection graph
- **Complex graph traversal** problems not suited to SQL queries
- **RPC** was used to keep it separate from Leo
- Our first service

Remote Graph



Mid/Back Tier Services

- **“Back” tier** services encapsulate **data domains**
- **“Mid” tier** services provide **business logic**
- We applied the **service pattern** to many domains, e.g. member profiles, job postings, group postings

Front Tier Services

- **“Front” tier** services **aggregate data** from many domains
- **Transform the data** through templates to present to the client
- Should be **stateless** for scaling purposes

Service Explosion

- Over 100 services by 2010
- Most new development occurring in services, not Leo
- **Site release** every two weeks

Architectural Challenges

- Test failures
- Incompatibilities
- Complex orchestration
- Rollback difficult or impossible
- Complex dependencies between services

Microservices?

- Services were fine grained
- But **monolithic build and release process** did not allow us to realize the benefits of microservice architecture

Solutions

- Continuous delivery
- Break apart the code base
- Devolution of control
- Strict backwards compatibility
- Better defined boundaries between tiers

Continuous Delivery

- Shared trunk
- Pre- and post-commit **automated testing**
- **Easy promotion** of builds to production environment

Decentralize Codebase

- **Separate**, independently buildable repositories
- Shared trunk within each repository
- **Versioned binary dependencies** between repositories

Devolution of Control

- Service owners **control release schedule**, release criteria
- Service owners are responsible for **backwards compatibility**
- Services must **release independently**

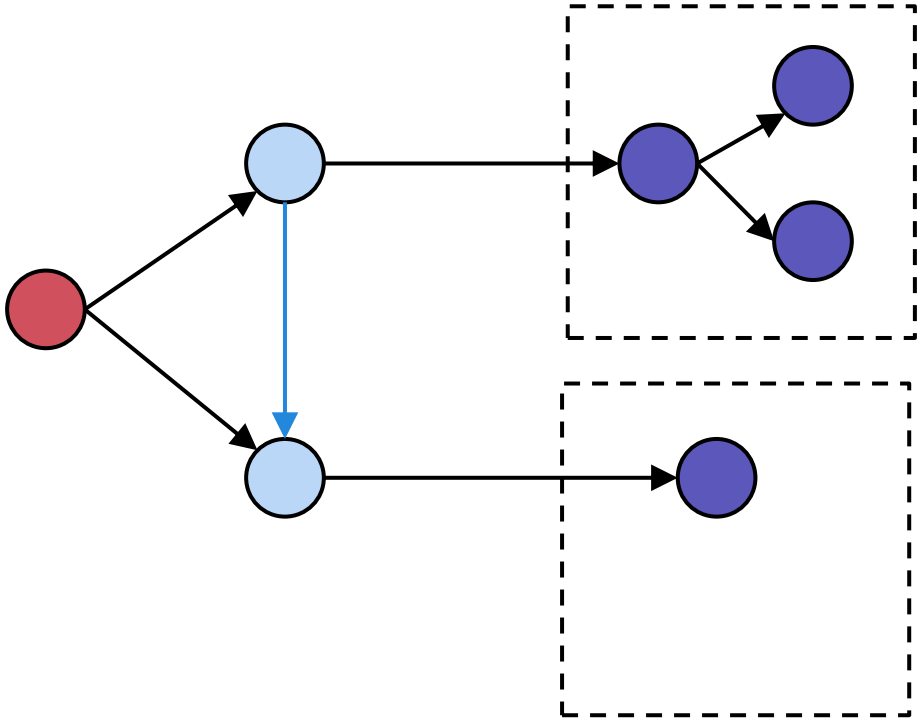
Backwards compatibility

- Insulates teams from each other at **runtime**
- Allows service owners to **deploy on their own schedule** without impacting clients

Boundaries Between Tiers

- **Limit aggregation** to the front tier
- **Limit crosstalk** in the back tier: **“superblocks”**

Boundaries Between Tiers



Java RPC

- Difficult to maintain backwards compatibility
- **Verb-centric APIs**
- **Use case specific APIs**
- Difficult to navigate the **proliferation of APIs**

Rest.li plus Deco equals Microservices at LinkedIn

What is Rest.li?

*“Rest.li is an **open source REST framework** for building robust, scalable RESTful architectures using **type-safe bindings and asynchronous, non-blocking I/O.**”*

Primarily **JSON over HTTP**.

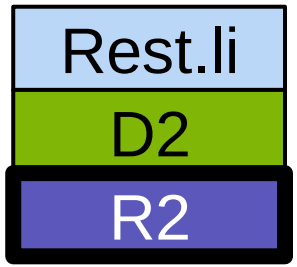
Why Rest.li?

- **Polyglot** (frontend) ecosystem - Java, Scala, Python, Node.js, Objective-C
- Uniform service interfaces (**REST**)

The Rest.li stack

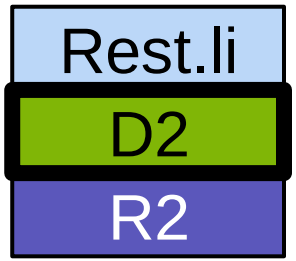
Request Response (R2)

- **REST abstraction** that can send messages over any application layer protocol (HTTP, PRPC (old custom LinkedIn protocol))
- Client - fully asynchronous **Netty**
- Server - **Jetty**, Netty (experimental)



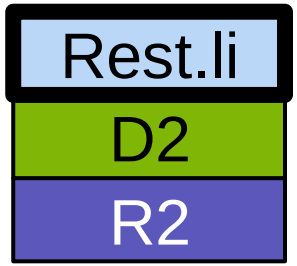
Dynamic Discovery (D2)

- **Apache ZooKeeper**
- Dynamic **server discovery**
- Client side **software load balancing**
- **D2 service**



Rest.li

- Data using PDSCs (**P**egasus **D**ata **Sc**hemas)
- **RESTful API** that developers use to build services
- CRUD + **finders** + **actions**
- **API and data** backwards compatibility checking



830 Rest.li resources.

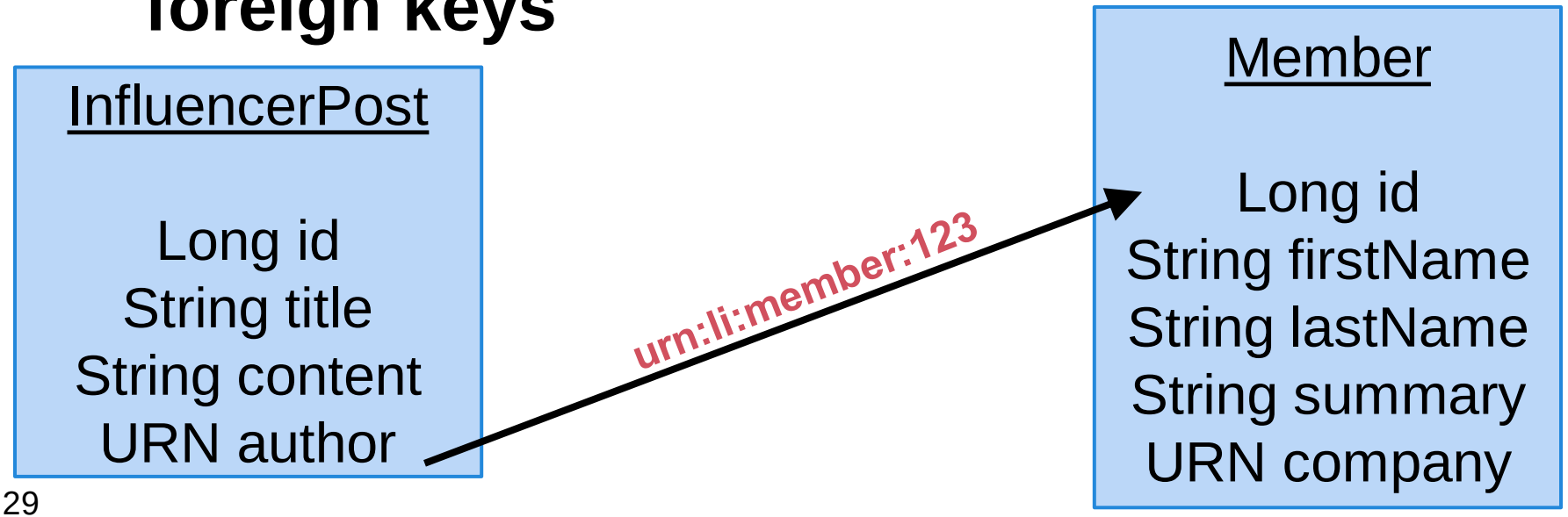
90 billion Rest.li calls/day across multiple datacenters.

65% service-to-service calls.

What is deco?

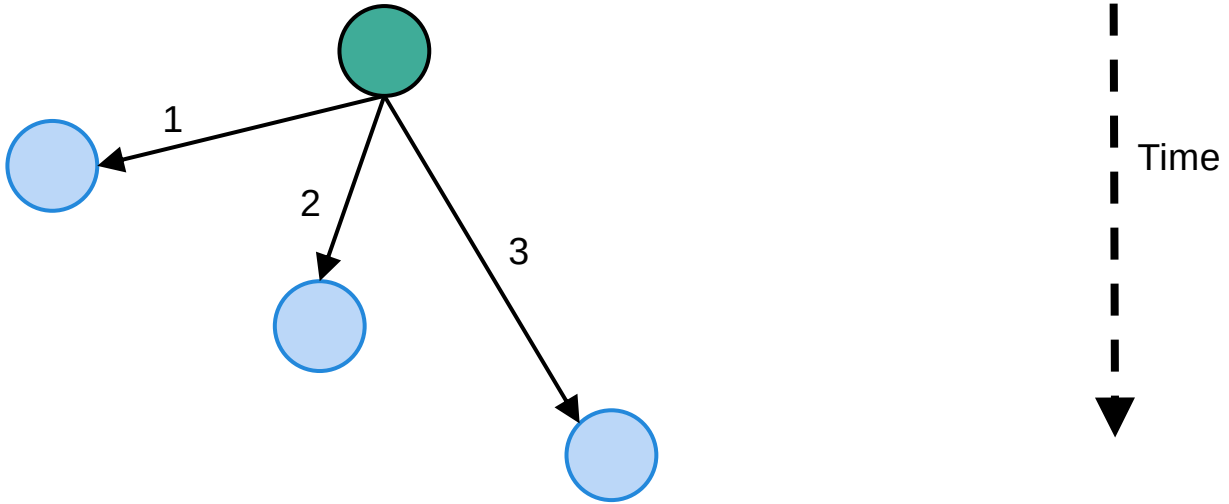
Aside: Normalized Domain Models

- **Links** over inclusion (denormalization)
- **URNs are fully qualified foreign keys**

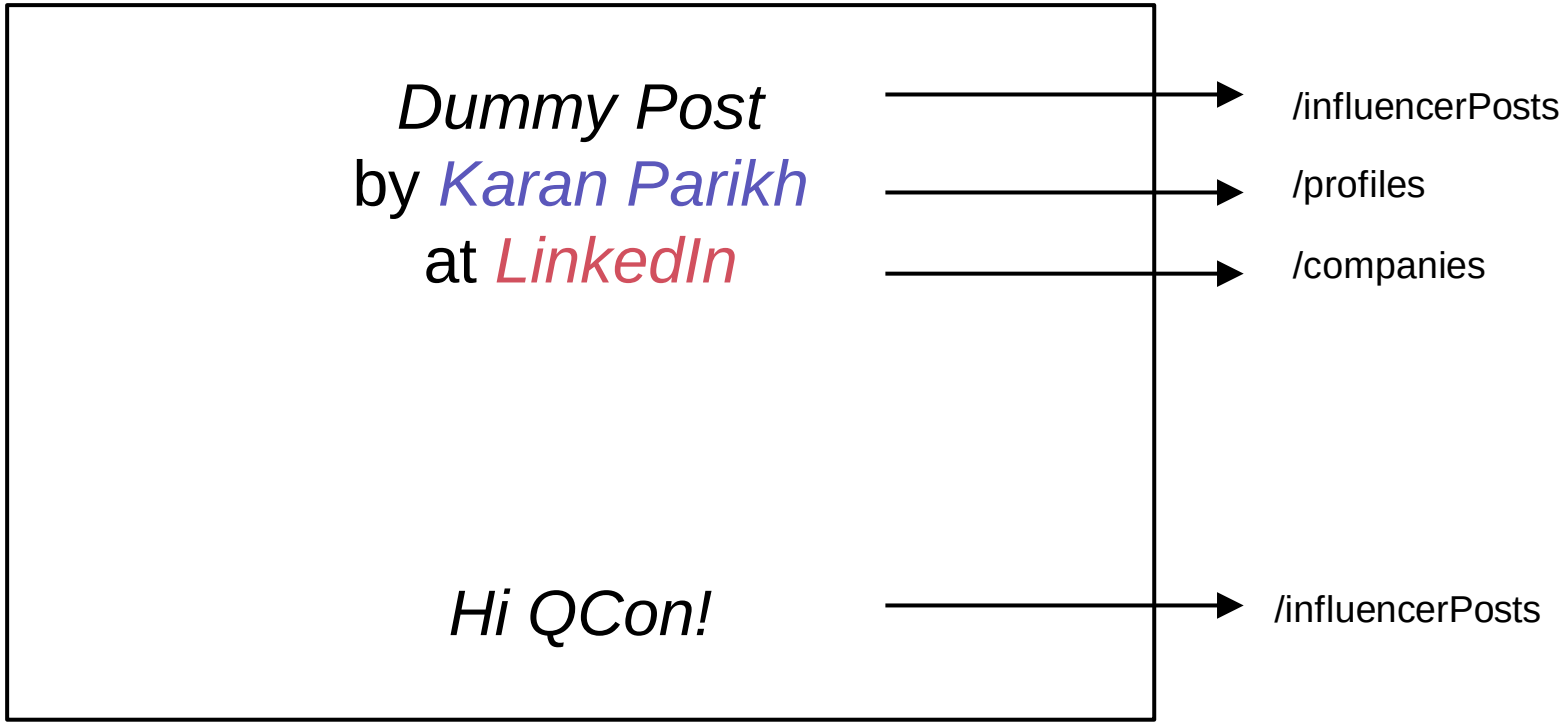


What is Deco?

- **URN resolution library**
- **What** data you want, not **how** you want it



Deco Example: Influencer Post



Deco Example: Influencer Post

deco://influencerPosts/123?**projection**=(title, content, **author~**(firstName, lastName, **company~**(companyName)))

Three services.
One client call.
Deco.

Rest.li plus Deco equals Microservices at LinkedIn

How Rest.li enables Microservices

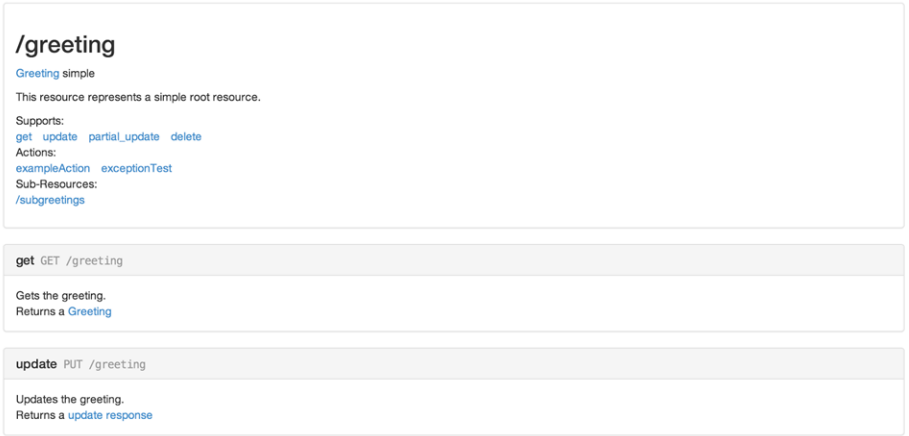
- Rest.li + D2 facilitate **domain specific** services
- Services can **easily configure** clients via D2
- D2 helps us **scale** the architecture

How Deco enables Microservices

- Deals with **service explosion**
- **Abstracts away** services from clients

Challenges

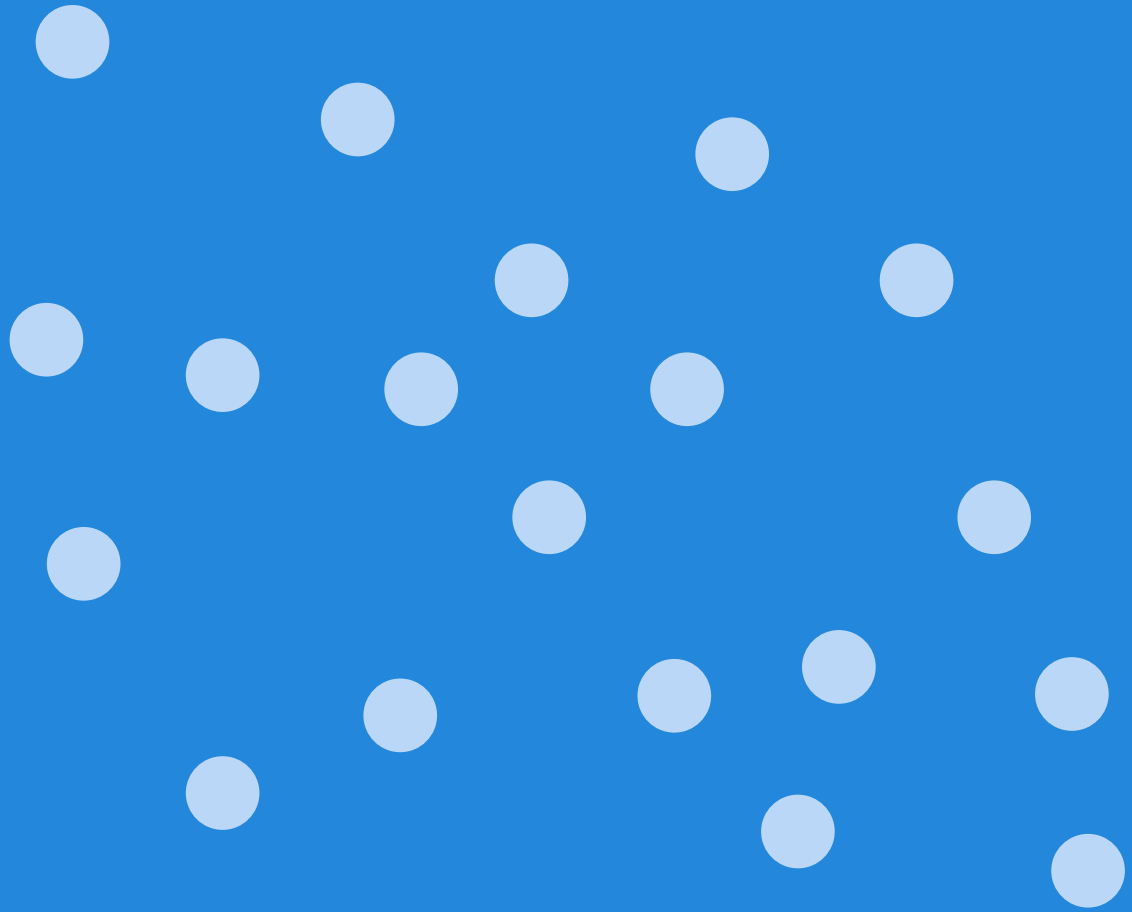
- Coordinating a **massive** engineering effort. (**LiX** to the rescue!)
- Ensuring **uniform** RESTful interfaces
- Performance



Rest.li API Hub

Wins

- All languages talk to the **same service**
- Developer **productivity**
- **Reduction** of hardware load balancers
- Ability to **expose APIs directly** to third parties



LinkedIn  Microservices

Questions?

References and links

- Rest.li: http://rest.li/
- Rest.li API Hub: https://github.com/linkedin/rest.li-api-hub
- Rest.li user guide: https://github.com/linkedin/rest.li/wiki/Rest.li-User-Guide
- Modeling resources with Rest.li: https://github.com/linkedin/rest.li/wiki/Modeling-Resources-with-Rest.li
- LinkedIn engineering blog posts about Rest.li: http://engineering.linkedin.com/architecture/restli-restful-service-architecture-scale http://engineering.linkedin.com/restli/linkedins-restli-moment
- LinkedIn's GitHub projects http://linkedin.github.io/