engineering.linkedin.com

Quick Deploy: a distributed systems approach to developer productivity

Adam Cataldo

7-9 minutes

The LinkedIn website is composed of a number of services, such as a service for Profile data, another service for Company data, and one more for Groups data. In production, each service is deployed separately and communicates with the other services via a well-defined API.

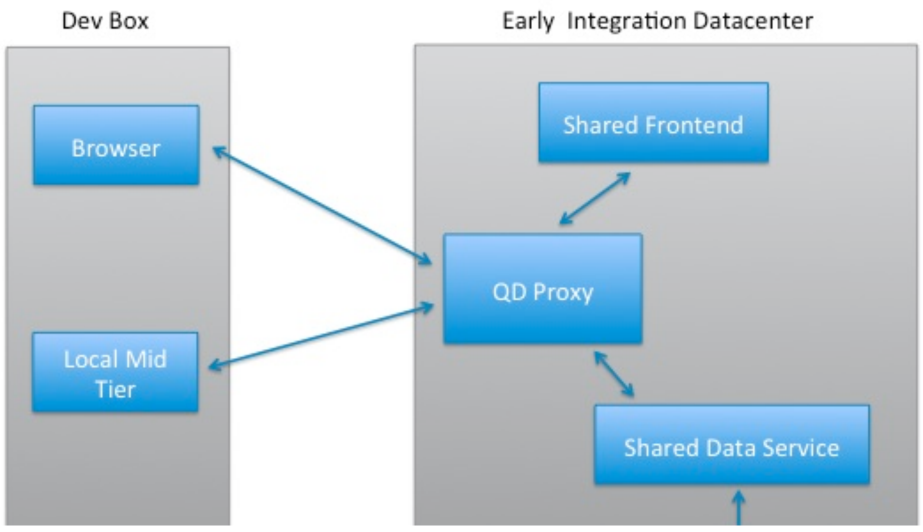
During development, LinkedIn engineers have historically deployed all the services on their local computer. LinkedIn has been growing fast and the number of services has grown along with it. This has gradually increased the time it takes to bring up the entire stack on a developer machine.

The ideal model for a developer is to deploy locally *only* the services he is actually modifying. Working with a few other engineers on a Hackday project, we showed how this could be done. We called this project "Quick Deploy", and in this blog post, I'll tell you how we used a distributed systems approach to make it work, some of the "gotchas" we ran into along the way, and the benefits we've seen as we roll this out to all of engineering.

Design

Most of the time, a developer is only modifying a handful of services. Our goal was that these should be the only services he deploys locally. Everything else should be deployed in an external shared environment. The main difficulty was how to correctly route requests: if there are many developers using this shared environment, and each developer has his own subset of services deployed locally, how do we figure out which services to go to for each request?

Our solution was to use a proxy and cookies. We setup a proxy in our early integration (EI) data center, where all of LinkedIn's services are deployed. These services are updated periodically and redeployed, so they are always running the latest code. All services - both those deployed in EI and those deployed on a developer's box - route all their requests through the proxy.



To use Quick Deploy, the developer workflow is as follows:

1. **Setup a routing profile**: a configuration specifies which services are deployed locally. Routing profiles are created using a simple web interface and are stored in a cookie in the developer's browser.
2. **Use the proxy server**: to test, the developer points his browser at the proxy server in the EI environment.
3. **Proxied routing**: the proxy will route all requests to EI or back to the developer's box based on the routing profile in the cookie. It will also ensure this routing info is passed along to all downstream services, so inter-service requests are routed correctly as well.

An example

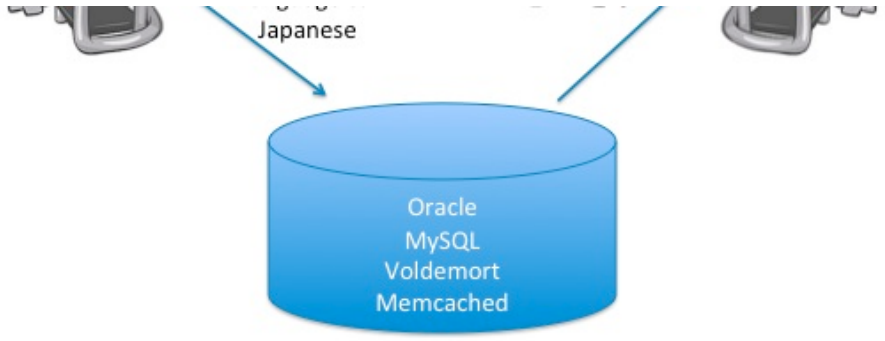
Suppose you're working on an app with a typical 3 tier architecture: a front-end that renders the UI (HTML), a mid-tier that handles business logic, and a data-service that reads and writes from the DB. You want to make changes to the mid-tier, so you deploy it locally and add it to your routing profile. Here's how requests will be routed when you test your service by clicking around the UI:

1. Your browser makes a request to the proxy server asking for the front-end.
2. The front-end isn't in your routing profile, so the proxy sends your request to the front-end deployed in EI.

3. The front-end needs info from the mid-tier, so it sends a request to the proxy.
4. Your routing profile is in this request as well, so the proxy knows to route this request to the mid-tier deployed on your developer box.
5. Your mid-tier needs data from the data-service, so it sends another request to the proxy.
6. The proxy routes this request to the data-service in EI.
7. Now, we wrap up: the data-service responds back to the mid-tier, which responds back to the front-end, which finally sends HTML back to your browser.

Problem 1: shared data

As we started rolling out Quick Deploy to the engineering staff, we ran into a few technical issues. One issue was that engineers were sharing the same database. We had developed a set of test accounts (Alfred, Bruce, Carole, Diane, etc) that all engineers used when testing their code. While getting ready for the Japanese language launch of the site, we often ran into issues when someone would change Alfred's language to Japanese. Any other developer using Alfred would get a big surprise.

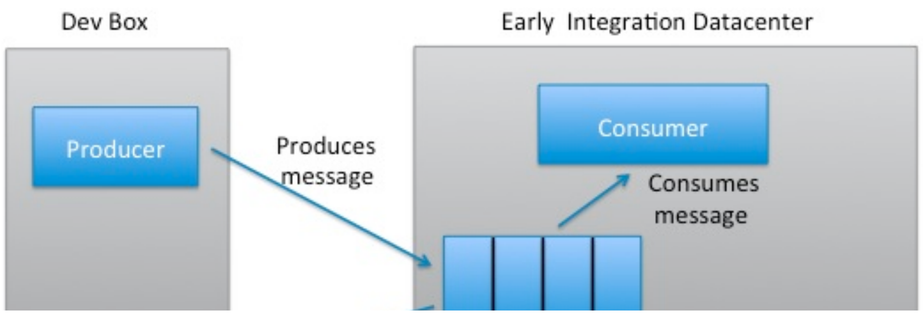


The solution to this was simple: we needed to create more test accounts. This way each engineer could have his own set of users to test with.

Problem 2: asynchronous communication

We have a few services that communicate asynchronously, such as services that use Quartz to perform scheduled tasks and services that use ActiveMQ to send messages. This communication is "asynchronous" in the sense that it happens outside of a user-driven request/response cycle, which means there is no cookie with routing profile information available. This can lead to unpredictable behavior as services deployed locally contend with those deployed in EI.

For example, with ActiveMQ, at most one consumer can consume a message on the queue. If a consumer is deployed both locally and in EI, there is no way to tell which one will actually consume any given message:



Consumer
Never sees message
Queue

The solution for this problem is to deploy locally both the asynchronous technology (e.g. ActiveMQ, Quartz) as well as the "cluster" of services that your changes affect. This removes contention between services in EI and services deployed locally. For example, if you're making changes that affect a particular set of ActiveMQ messages, the workaround is to deploy locally both the relevant "cluster" of consumers and producers for that set of messages and your own instance of ActiveMQ. The services deployed locally only consume from the local queue and the services in EI only consume from the EI queue, so there is no contention.

The Benefits

As we iron out the bugs, we are rolling out Quick Deploy to more and more people, and expect all of engineering to be using it by the end of the year. What started as a Hackday project is now yielding huge benefits for the entire company, including:

- **Faster iteration**: engineers deploy just the services they need, while all other services are centrally managed and kept up to date. This saves lots of time on every single project.
- **Big data early**: before Quick Deploy, due to the limited resources on a developer box, the data set loaded in the

local DB for testing was very small. We could only use a full data set (ie, on the order of 135+ million members) when the code reached a staging environment shortly before release. With Quick Deploy, services in EI are deployed across a large number of machines, so we are able to use much larger data sets early on, catching performance and scaling issues from day one.

- **Continuous integration**: in the past, features would be developed and tested on a developer box and only integrated with other projects shortly before release. As we migrate all of engineering to use Quick Deploy and trunk development, all projects will be tested together through out the entire development cycle.

Acknowledgements

The original hackers who prototyped Quick Deploy were Adam Cataldo, Prachi Gupta, Thomas Feng, and Angelika Clayton. It won the "Most Likely to Make Engineers Happy" Hackday award. Brent Vincent has been managing the effort to roll out Quick Deploy to all of engineering.

Swee Lim, Omer Shapira, Brian Geffon, Rob Loh, Lisa Manresa, Andy Chen, Hans van de Bruggen, Shruthi Dathathri, Steven Ihde, David Hoa, and Priyanka Salvi have helped build a scalable, fault tolerant proxy, along with related tooling.

Yu-En Tsai, James Cope, Sudhakar Krishnamurthy, Peter Poon, Lifeng Sang, John Kristian, James McPhail, Dmitriy Yefremov, Daniel Wong, Bryan Hsueh, Qunzeng Liu, and

Jim Dumont have helped the engineering teams work through service-specific changes needed to take advantage of this technology.