


The Old and New of Enterprise Application Development



Rich Dudley
Chief Evangelist, /n software
@rj_dudley
@nsoftware

About Me

- Over 20 years in enterprise development, manufacturing to logistics to financial services
- Wide variety of really cool projects
- Developer -> Principal Enterprise Architect
- Microsoft ASP Insider and AWS Community Builder



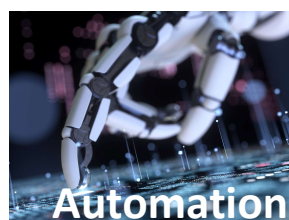
I am currently the Chief Evangelist for /n software, which I joined earlier this year. Prior to now, I was an enterprise developer for 20+ years, most of that time was at just two companies which valued innovation and technology. Over the years my interests and opportunities grew my career from a developer to a Principal Enterprise Architect to my current position. I am also currently a Microsoft ASP Insider and an AWS Community Builder.

Worked with major ERP systems, EDI integrations across hundreds of suppliers and distribution points, to cloud-based services-oriented architectures automating the mortgage process

The best part about being at the same companies for so long is you get a chance to see things come and go, and be part of a lot of changes. And that's what I want to talk about with you all today.

I am not speaking to you with the voice of my current role, but from someone who's been part of and led changes in enterprise applications, especially looking at the last few years.

Drivers of Change



Working in software development, changes are inevitable, whether it's changes in what our applications and services need to do, or changes in how we architect, build and host our applications and services. Some of the factors driving changes I'm going to discuss today include:

- Regulations (and penalties): we've all had to deal GDPR, CCPA, SOX, HIPAA, or something else and there is little doubt more regulations will bring more change in the future
- Information security, especially protecting critical information through authorization and encryption
- Analytics and data-driven approaches to running a business are growing in popularity. This isn't product development, but measuring productivity and the success of product lines and corporate divisions
- The ability to consume and analyze massive amounts of data in near real time is driving change. Big Data may not even be very big, it could be just cumbersome to work with using the tools and applications currently in our environment.
- Doing more with less—automating a lot of the day-to-day toil. This is both for the business users of our applications, as well as how we build and deploy what we create.
- March of technology: especially the cloud, containers, operating systems and language improvements

The list here is not an all-inclusive list of factors which drive change. One factor not on this list is the Covid pandemic. The pandemic had a massive effect on society, but on its own didn't directly affect application development that much; rather, the pandemic did remove the luxury of time to adapt to some changes which were already in motion.

An important thing to keep in mind is change is a good thing. I like to use the analogy that the bicycle didn't fly, the biplane didn't get us to the moon, and what got us to the moon won't get us to Mars. Change is how we take ourselves from where we are to where we want to be.

One thing about this talk, there are a lot of slides with text. I really wanted this to be "useful on Monday" without you having to take a lot of notes or screenshots. If you want the slides reach out to me, but don't worry about writing everything down.

Deployment Automation

- old: manual build and deployment
- new: automated build and deploy
- changes: improved source control, distributed build and deployment tooling, infrastructure-as-code, code signing



Automated build processes aren't meant just to reduce toil, they're also meant to improve the security and stability of our services catalog.

Automated processes aren't necessarily continuous, they may start with a manual trigger, but are automated all the way through a dev or test environment.

There is a lot of handwaving about "continuous", but the juice may not be worth the squeeze. An automated with a manual trigger is usually good enough, and on distributed teams helps keep developers from tripping over each other.

In many companies, developers also define their own infrastructure, which also gets deployed as part of these processes. This imparts the need for additional security guidelines developers will need to keep track of.

If you're not far in the adoption of automated build, some of the changes you'll see are...

Business Automation

- old: forms to data, data to forms
- new: forms to data, data to services
- old: monolith
- new: smaller monolith, supported by resource intensive or parallel activity services
- changes: choreographed or orchestrated events driven architecture



For years there has been a move toward “right-sizing” companies. This doesn’t mean just laying off staff, it’s more about finding ways to increase production without just throwing more people at a problem. This means automating a lot of manual tasks, especially tasks which can be done in parallel with other activities, whether manual or also automated. More transformational companies ask “before we hire a lot of people, how much of we need to do can we automate?”

People in these processes are then specialized to handle the proverbial “20%” or edge cases which are difficult to program for.

The previous “forms to data, data to forms” usually revolved around monolithic systems like SAP or Peoplesoft. People take orders using a screen in the ERP, purchasing are run by clicking a button on an ERP screen, summaries and reports are reviewed by looking at an ERP screen. Resource intensive activities usually had to be run in overnight batches, or would slow down the entire system.

Forms to data, data to services still leaves room for the monolith, but some of the more resource intensive processes are farmed out to external services, as are activities which can be performed in parallel with the human workflow. One advantage of using distributed services instead of bolt-ons into the ERP is that they are much easier to modify, test, deploy

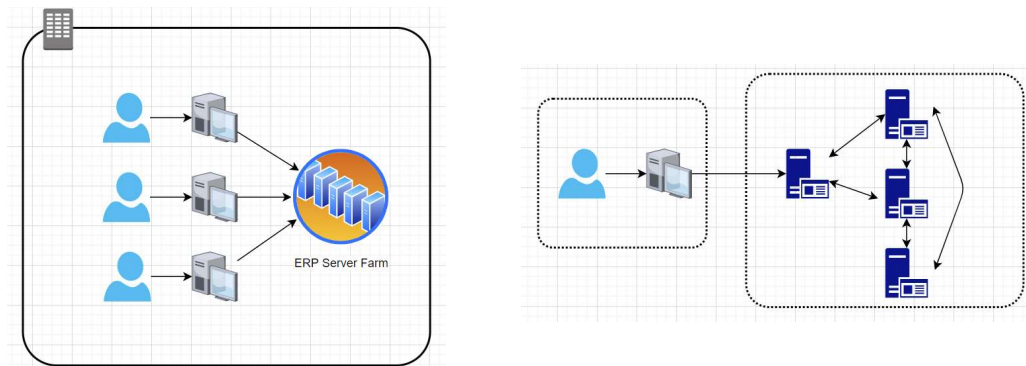
and swap out. This also means you can move compute to other places. And now the previously mentioned batch activities can be done in smaller batches in near real time so someone doesn't have to wait until tomorrow or Monday for their results.

These services can also open up new channels, since ERPs and other internal line of business applications are usually not meant for the public internet. By adding a front end, you can reuse these services to take sales orders online, schedule delivery, validate inventory and addresses and so on, in real time.

At a previous employer we did this to great success. While taking a mortgage application over the phone, after we received the subject property address, the processes to verify the address and estimate the home valuation were automatically initiated. We thought this would reduce the length of time on the phone, but what we found was the overall length of the process was shorter by 3-4 days because clients had better information earlier in the process and made better decisions. We then had these services at the ready when we decided to build a completely web-based origination process; this saved us years of development.

How does this affect what we all write? We'll probably move to having more orchestrated services in our environment, automating manual processes and easing resources.

Shift in Architectures



 **software**

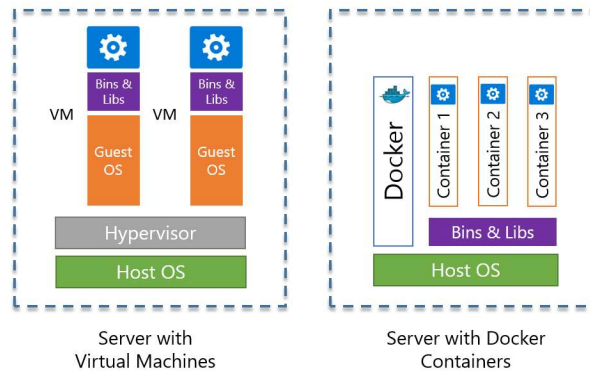
Your architectures are going to look something like this very small example.

On the left is a traditional ERP application, hosted on a server farm.

In the new architecture on the right, the main application is probably still going to be something like an ERP application with some additional services to add capabilities or reduce resource utilization (“shifting compute”). In a choreographed service catalog, these services will talk between themselves; in an orchestrated pattern, there will be a conductor service as well. This drives yet more changes to what we’ll write--every service in the inventory will have to have its own security and communication layers (this is a good time to see if your favorite component vendor has some toolkits to make this significantly easier).

VM vs Containers

- old: physical servers, virtual machines
- new: containers



<https://www.cybrary.it/blog/0p3n/docker-containers-security/>



Very likely these new services will be run in containers rather than on physical servers or virtual machines. I come across people who have deep knowledge of containers, others for which this is new. Hopefully my oversimplification helps those of you new to containers.

Virtual machines were entire operating systems in a virtual environment, plus the required libraries, plus the application. The application was usually only a tiny fraction of the overall size of the VM. The OS in the VM handled all the processing. As far as the application knew, it was running on a physical machine.

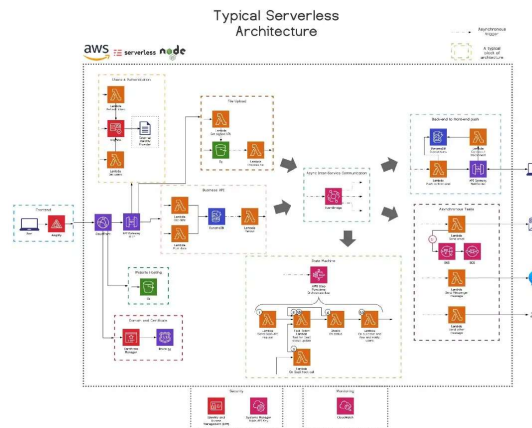
Containers take a very different tack. The only stuff in the container is the application, its dedicated libraries and anything special not found on the host OS. Instead of a hypervisor, an orchestration system like Docker or Kubernetes manages the containers. Containers are still isolated from one another, and are essentially their own little subnet of a physical machine. The upside to this is containers are very compact, easy to spin up and terminate, and you can run a lot of them on lower powered machines. This makes it really easy to test services on a local dev machine in a nearly identical environment to prod, and before deploying to a test environment. The definition of a container is defined in a text file, and build processes can install all the essential libraries, meaning the runtime environment can now be placed into source control and revisions easily tracked.

With VMs you could mix and match host and virtualized OSs. With containers, you can't—the host and container have to match. This is a lot of what's driving the move to Linux. Microsoft put a lot of effort into containerizing Windows, but as a GUI first OS it just didn't work well. But, thanks to the Windows Subsystem for Linux, you can host Linux containers natively on a Windows machine, same as if you were using a Linux or macOS.

Containerization is contrasted with serverless, and the compare/contrast is an entire session of its own. There really isn't a right or wrong, just what's best for us.

Serverless Architecture

- old: physical servers, virtual machines
- new: serverless



<https://medium.com/serverless-transformation/what-a-typical-100-serverless-architecture-looks-like-in-aws-40f252cd0ecb>

in software

Since I just mentioned it, another architecture pattern replacing servers and VMs is Serverless. Serverless is heavily promoted by AWS, but Azure, Google and other cloud providers also have serverless offerings.

The author of the image and linked article calls this a “typical serverless architecture”. In my experience there isn’t really a “typical” serverless because the uses are so varied, but this is a good and thorough example.

In Serverless architecture, we don’t have to manage the execution environment, we’re only responsible for creating the infrastructure whatever code is necessary for business logic. All the communication, security, etc. are managed by the cloud provider. The cloud provider has lots and lots of small specialized services which we piece together into a single application. These services can be API gateways, databases, caches, functions, and so on.

As I mentioned a moment ago, stand-alone services all need their own authentication and communication code, in addition to the service wrapper. In some cases, the code to run the services can be more than the code for the business logic. And that’s one of the factors which makes Serverless a compelling option.

Instead of writing code for multiple stand-alone services, we use infrastructure as code

(usually something like Terraform or Pulumi) to define which managed services we want to use, and then we're just responsible for coding business logic in a Lambda Function, or a Step Function, or Event Bridge rules, or whatever we're using.

One of the criticisms of Serverless is vendor lock in—the code for AWS Lambda Function won't run in an Azure function. There is sometime difficulty in adapting to something so different, and managed services are a bit of a black box with lots and lots of settings to get right. Looking at a diagram like this, Serverless seems to have increased complexity because code paths and classes are replaced by managed services and connections, so the dirty laundry is more visible. If you diagrammed the classes and code paths in some of your services, it would probably look pretty hairy too.

A confusion or complaint to dispel about serverless is yes, somewhere in the stack, there are a bunch of physical machines. However, the hardware and cloud fabric OS are designed in such a way that the physical machines operate as one large machine, and the OS can spread resources out and provide redundancy all invisibly to us.

Multiple Operating Systems

- old: Windows desktop everywhere
- new: Windows, Mac, Linux, iOS, Android
- changes: increased browser-based or cross-platform desktop applications



It was not that long ago my development team reassured ourselves we would not see a time where Windows wasn't the only OS we needed to develop for. Well, that turned out to be very optimistic.

Today's desktop applications have to work on both Windows and Mac, and we may need a mobile app also (which is a whole new UI ballgame being touch based rather than mouse and keyboard based). Linux probably won't be a major factor in desktops, but I've been wrong before.

Servers however are increasingly moving toward Linux, especially with cloud and containerization. I'm not sure whether Windows or Linux is more prevalent today, but I'd guess in the next 5 years Linux takes a large lead in server deployments. It's difficult to measure this accurately with elastic scaling and containers, so this may be true today.

How does this change the applications we build?

You can't totally future-proof your technology choices, but you can reduce the risk of OS changes by aligning with companies which emphasize cross-platform capabilities. It's very likely those companies will continue to do so in the future.

New Authentication Methods

- old: username + password
- new: OAuth, OIDC, OTP, Single Sign-On (SSO)
- changes: zero trust, migration from LDAP/Kerberos to OAuth, authentication no longer inherent or invisible, federation with partners, multi-factor authentication



As a follow up to the previous slide, authentication and authorization are changing.

An all Windows network with Active Directory sure was easy street when it came to securing applications, just add someone to a group and everything just worked. That environment is quickly going away. Part of what's driving this is Windows, Mac, Linux and mobile all have different capabilities in authentication and authorization, so the increasing heterogeneity in OS is driving technologies like OAuth and SSO. Also, AuthN/AuthZ often have to be federated with external partners like Salesforce, which have their own AuthN/AuthZ systems. Additionally, the bad guys have better computing power, so if they intercept network packets they can usually crack passwords and keys fairly quickly. The time-based components of OAuth make this a quantum-level calculation, and by that time the keys have rotated.

OAuth and OIDC are used to allow an app or a website to connect to another on your behalf. If you've connected the mail app on your phone to a Gmail account, you've seen OAuth in action. If you've used your Google account to log into eBay, that was OIDC. The mail app or partner website no longer stores your credentials like it used to in POP/SMTP/IMAP days. Now, you log into Gmail directly, and the mail app caches a long-lived token. Microsoft has deprecated the old username+password for many of its services, those are now OAuth based.

OAuth doesn't apply just to user AuthN; rewinding a couple of slides, OAuth is also used for service-to-service AuthN.

As companies use more SaaS platforms, there is a greater adoption of OIDC and other SSO technologies. This way, a user's login to a corporate security portal can be used to log in to the SaaS platform. Users only need to log into a portal once, and security can be centrally managed by the security team. It's win-win, so very popular.

OAuth and OIDC still largely use username and password, but we have less and less trust that a login is actually the user. To increase our trust, we add an additional factor to the login process. Multi-factor authentication, especially one-time passwords, is becoming more and more prevalent on both the public internet and inside corporate environments. One-time passwords are those six digit codes which are accessed in an authenticator app, or are texted or emailed to you so you can authenticate.

Changes we'll see include supporting some or all of this, depending on what you're building. Zero trust essentially is assuming we can't trust anything—the credentials aren't being used by the correct user, the communication channels aren't secure, etc., and then adding additional factors to increase trust...

Auth Dictionary

- OAuth = An open standard designed to allow an application (e.g., mail app) access to a user's information (e.g., Gmail). The login used is that of the information holder (e.g., Google)
- OIDC = Open ID Connect, uses OAuth and JWT, allows one website to use a login from a different website (e.g., Google login to eBay)
- SAML = XML-based language for security assertions
- JWT = JSON-based standard for holding claim tokens, and supporting encryption/signing
- OTP = one-time password, the 6-digit number from an auth app or SMS/email in addition to a password or cookie



OAuth is maintained by IETF

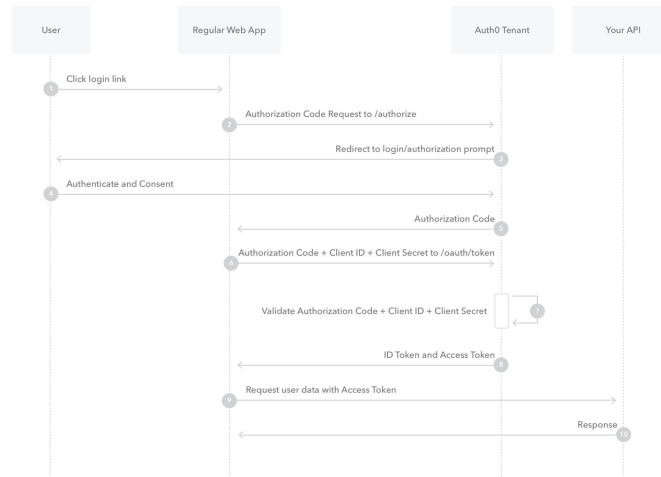
OIDC is maintained by OpenID Foundation. The original idea of OpenID was to be a single sign-on system where public websites wouldn't have to roll their own login systems, they would just trust a few of the big players, referred to as identity providers. There are many OpenID identity providers, including Facebook, Twitter, Google and Amazon. Since this OpenID is a standard, you only need to implement OpenID once in your website to support all these identity providers.

SAML is maintained by OASIS

JWT is maintained by IETF (proposed status since 2010)

OTP is an IETF standard

OAuth Process



<https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow>



If you've never seen what the OAuth workflow looks like for a web application accessing an API, this is it. This is directly from Auth0's documentation, and you can see where the tokens are requested and used. The User column includes the UI app (browser, mobile, desktop), the Regular Web App is the back-end service.

This diagram also applies to the service-to-service scenario, with a couple changes—there is no User, and Regular Web App is any type of service.

This diagram has been a good reference for me over the past few years when discussions of how we secure services.

Work From Anywhere

- old: private network and work from office
- new: public internet and distributed workforce
- changes: browser-based apps, more APIs, the need to secure apps/APIs/communication channels



For a variety of reasons, fewer and fewer people are working in the office. This is one of the most significant changes the pandemic forced, but was also a growing trend before the pandemic. I've been a work-from-home remote for over 10 years, my previous employer had about 25% of the tech staff and about 10% of the business staff as remotes. We had a good head start when the pandemic forced 100% work from home.

Internal applications and resources now have to be more accessible from anywhere, whether that's home, coffee shop, a plane, wherever. In the past, you'd use a VPN and essentially be on the network, business as usual, but VPNs are being phased out, largely because they are expensive and can be slow and require a lot of care and feeding. VPNs also prevented users from accessing local resources, especially printers, and a distributed workforce often needs local device access.

Now, with web-based applications such as Microsoft 365, Google Apps and Salesforce users have the expectation that they can just open a browser or a mobile app, log in, and start working. This changes the entire security paradigm of the apps we write, and reinforces the need for a good service inventory since browsers are insecure and resource limited (you can no longer count on local processing power).

This isn't to say every app can or should be browser-based, but you'll likely be asked to add

a web experience where you can. Whether desktop app or web app, we can no longer act as if the connection is protected by our corporate network boundary. The security of data in flight becomes a principal concern.

Data Security

- old: open access to data
- new: end to end encryption and limited access
- changes: encryption/decryption, AuthN/AuthZ more important



Speaking of data security... It wasn't that long ago almost anyone could just dump a company's data into Excel and do some analysis. As more and more data was accumulated, and especially more and more private data, that turned out to be a very bad practice. People had the ability to access data they really shouldn't have had access to, and weren't securing it well. The inevitable data releases happened and continue to happen.

This has rightfully led to a greater effort to secure data—including where data is stored, how you can access it, what is available, what is encrypted.

Complicating this is work from anywhere, where data may now travel over the public internet instead of just inside a corporate network.

Regulations such as GDPR, CCPA, HIPAA are focused on categorizing what is private data and how private data should be handled. Regulators have lost patience and penalties are severe these days.

A big take-away for encryption is to not create your own encryption. This is one of those cases where you absolutely want to buy something and use it.

Observability



Some of the changes we've talked about so far—especially work from anywhere, distributed services and developers responsible for their own infrastructure--make information security teams more nervous because it's far more difficult to know what is going on. As a result, observability is an additional trend growing in importance.

Observability can be described as “who, did what, where and when, and all that information from all users and applications is streamed in near real time to an analysis platform”. These observability platforms quickly analyze data and look for operational and security issues.

Observability

- old: exception alerts and logging
- new: observability, monitoring and auditability
- changes: interface with streaming or ESB platform, send all application activities to a different system, let InfoSec figure it out
- Pro tip: opentelemetry.io



With logging and notifications, we write code to produce messages when certain actions are taken, or when something goes wrong. This means developers at code time are making decisions about normal and not normal activities. Alerts are usually sent to a small group, logs periodically need to be cleaned up and are usually looked at only when something goes wrong. Logging and alerting can miss application issues until they reach a threshold of errors.

Monitoring/observability changes where and when the decisions about “normal or not” occur. An application just sends everything to a different system. Analysis rules are applied there, patterns of activity are watched by specialized analysts, and anomalous activities can caught before they become a problem. This platform sends alerts and provides a basis for auditing in case of regulator questions.

A pro tip here is to keep an eye on Open Telemetry. This is an industry standards organization, backed by the major players in observability/monitoring. Seeing what that group is up to is a good way to watch the industry, and you’ll likely need to interface with their participants in the future.

Governance

- old: confusion about data, tech arguments
- new: data governance, technology governance
- changes: be a positive force for change, take a seat at the table for governance committees



There was a short lived trend as part of the “startup culture” big enterprises wanted to nurture where teams were free to make their own technology choices. This sometimes went poorly, ended up with a hodgepodge of resume driven development and incompatible systems, not to mention data issues.

At previous companies, before we implemented some level of data governance, we wasted way too much time tracing the ontology of data elements. Implementing data standards is a way to spend more time developing and less time troubleshooting.

Data governance means:

- Data has the same meaning throughout the company (do we have clients or customers)
- Data has the same metadata throughout the company (type, length, precision – especially difficult with GUID/UUID)
- Data has the same security implementation throughout the company (employee tax numbers are always encrypted and only an HR manager can see them for only specific reasons)
- All tech uses the same schemas for APIs and streaming services

Growing use of industry standards, FinDEx, Swift. If you remember EDI, there was one standard for the 850 Purchase Order, but every vendor had their own implementation.

Same for all the other documents. EDI tried to cover all industries, and being all things to everyone was part of the problem. The emerging industry standards don't allow for that, they are industry specific, designed by participants and everyone uses the same implementation. They are also designed from the start to be much more friendly to APIs and other modern technologies than EDI is.

More and more companies are also establishing technology standards teams. Some of what these teams standardize on are:

- Architectures (serverless or containers)
- The languages and platforms we use
- To what extent do we/can we embrace open source vs purchased components
- Ensuring software supply chain security
- How do we review products we think we need to buy, and how do we make the purchases

The guidelines which come from standards teams are inevitably going to bring some changes to an enterprise.

In my experience, some developers respond poorly to standards, with the mindset of "you can't tell me what to do", especially if their preferred language or platform is not in the standard. I say this as someone who has authored industry and corporate standards, we're not trying at all to diminish you and your skills as a developer, we're trying to prevent colossal mishaps. We need you and your skills to be awesome, and you can be a very positive force for change. Positivity brings results and rewards.

Generational Changes

- old: Boomers, Gen X
- new: Millennials, Zoomers
- changes: better application design, be empathetic and treat people well



One final non-tech, non-business driver of change is who is in the workforce, and their expectations. I'm Gen X, and recall that most of my last 20 years was spent at just two companies. If I wanted to it could have been just one company this entire time. Younger generations have a much more "employment-at-will, to the highest bidder attitude", with fewer qualms about bouncing from place to place over a short period of time.

Older generations are becoming more used to not being in the office as much, which when combined with a tech-savvy younger generation, the tolerance for what users will endure is much less than it used to be. Clunky, slow applications with poor UX will frustrate users and drive them away. Internal line of business applications have some wiggle room, but SaaS applications and commerce sites and apps do not. UI/UX will become more of a feature than it already is. This is a progression from "slash menus" to "swipe right".

With such a diversity of coworkers and teammates, not merely in age but everything else, be empathetic and treat people well.

Summary

- There will always be reasons for changes to our applications, including
 - Automation of business and development processes
 - Increased regulation
 - Security concerns
 - Flexible work locations
 - Generational changes
 - Corporate and Industry standards
- The major changes to our applications at this time include:
 - Cross-platform, lighter UI
 - Reusable back-end services to add capabilities and shift resource utilization
 - Inter-application communication
 - Reduced trust, increased user, application and data security
 - Observability





Thanks everyone for listening, I hope this was interesting!