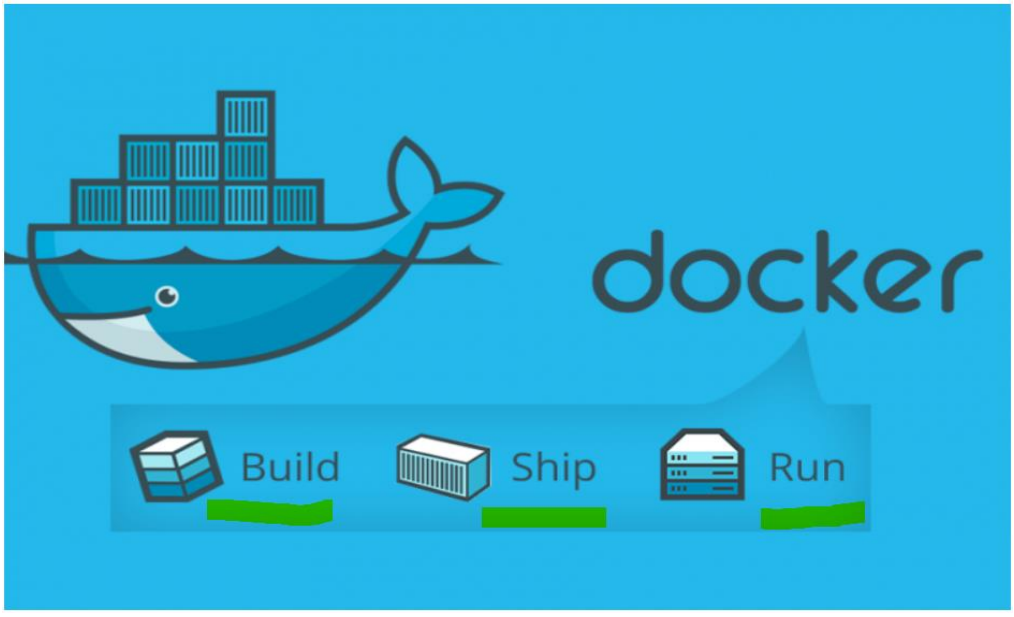
DevOps

Development and Operations

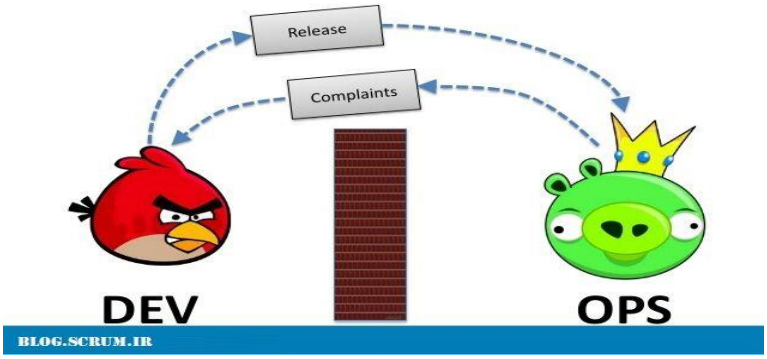


فهرست

DevOpsچیست؟ و کاربرد آن کجاست؟

شاید IT یکی از بزرگترین صنایعی باشد که هر روز در آن واژگان جدیدی به دایره لغات ما افزوده می‌شود، یکی از این لغات جدید DevOps است که از سال 2009 شروع به ظهور کرده و از ۲۰۱۴ بسیار مورد استقبال قرار گرفته است و اگر در لیست مشاغل خارجی بدنبال آن باشید، می‌بینید که شرکت‌ها بشدت دنبال افراد متخصص در این حوزه می‌گردند.

روزگاری در شرکت‌ها توسعه نرم افزار دو تیم وجود داشتند که با یکدیگر دوست نبودند، یکی از آنها Dev یا تیم توسعه و آن دیگری Ops یا تیم عملیات بود. شاید به ظاهر در یک واحد تحت فرمان مدیریتی یکسان بر روی پروژه(های) مشترک کار می کردند ولی اهداف آنها کاملا متضاد بود. هدف تیم توسعه ساخت ویژگی های جدید و تغییرات زیاد بر روی محصول بود ولی تیم عملیات بدنبال پایداری و ثابت نگه داشتن وضعیت سرویس های موجود بود.



برای همین مابین این دو تیم یک دیوار نامرئی (و گاها در تجربه ما در ایران دیوارهای مرئی) به وجود می‌آمد. مفهوم DevOps بدنبال این است که با از بین بردن دیوار مابین (مرئی یا نامرئی) تیم‌ها، و افزایش تعامل نفرات، موجب افزایش سرعت تحویل ارزش به مشتری شود. پس خیلی ساده، DevOps فرآیندی است برای تحویل سریع ارزش به مشتری و از بین بردن هر نوع مشکل که باعث کندی در فرآیند تحویل ارزش شود.

این مفهوم چرا مهم شد؟

با جدی شدن بحث Cloud و حرکت تیم‌ها به سمت توسعه نرم افزار چابک (اینکه در این روش سرویس‌ها به سمت زنده بودن و تعامل همیشگی با مشتریان و تغییر بر اساس نظرات آنها پیش رفت)، دائما نیاز بر این داشتیم که نسخه‌های جدید محصول در دسترس مشتریان قرار بگیرد. ارتباط ضعیف مابین تیم‌های تضمین کیفیت، عملیات و تیم توسعه، باعث می‌شد فرآیند تست، انتشار و تحویل زمان بر باشد و هر بار هر مشکلی مشاهده می‌شد این تیم‌ها همدیگر را سرزنش و محکوم می‌کردند.

در مفهوم DevOps ما سعی می‌کنیم این تیم‌ها به هم نزدیک‌تر شوند و با تعامل و همکاری بهتر و البته اتوماتیک کردن بسیاری از روال‌های تکراری، تحویل ارزش به مشتری دچار مشکل یا کندی نشود.

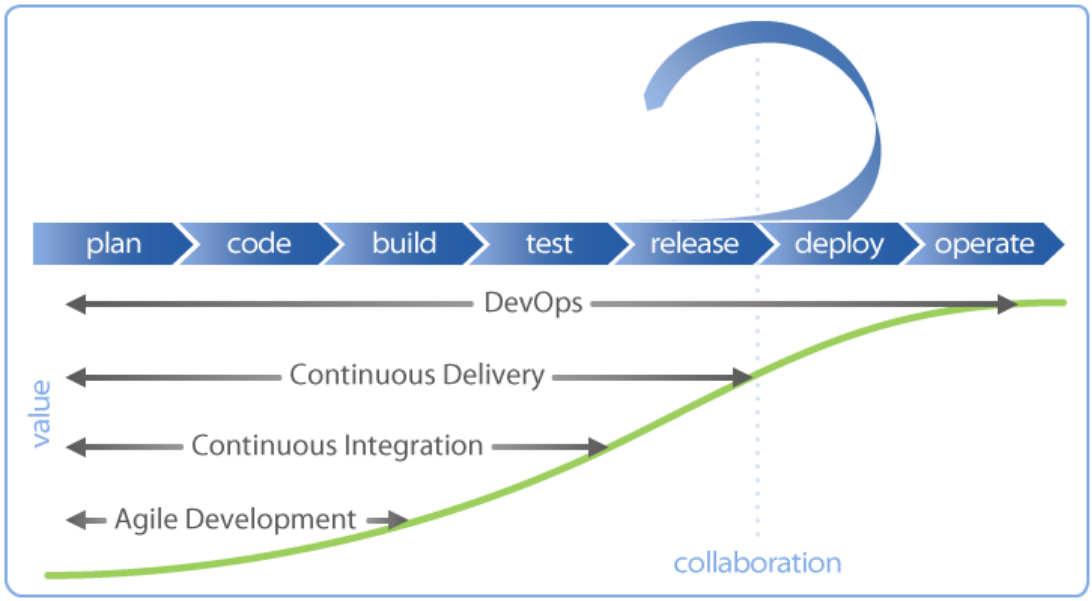
کج فهمی‌ها در موردDevOps

با توجه به جدید بودن این مفهوم، کج فهمی‌های زیادی در این مورد وجود دارد، و البته این فقط در ایران نیست و برخی خارجی‌ها نیز در این مورد درک درستی ندارند.

DevOps فقط Continuous Delivery نیست

خیلی از دوستان فکر می‌کنند DevOps همان Continuous Delivery است. یعنی اینکه ما در ابزار TFS یا Jenkins یک CI راه بیاندازیم و عملیات Deployment را اتوماتیک کنیم، پس ما DevOps هستم. حتی در بعضی از جاها با عنوان مهندس DevOps آگهی استخدام می‌زنند که در شرح شغل فقط دنبال کسی هستند که ابزار CI را پیکربندی کنند.

اتوماتیک کردن روال تحویل یا انتشار محصول به سرورهای تست یا سرورهای تحت بار مشتری، فقط بخشی از چارچوب کلی DevOps است.



DevOpsیک تیم نیست

بعضی نفرات فکر می‌کنند که DevOps یعنی یک تیم متشکل از برنامه نویسان و بچه‌های عملیات. خود ساخت این تیم مفهومDevOps نیست ولی شاید یک روش برای رسیدن به این فرآیند باشد و شاید در بسیاری از سازمان این روش نارکارآمد باشد.

چارچوب CALMS

CALMS یک چارچوب راهنما برای رسیدن به فرآیند Devops است:

۱-**C**ulture

همانطور که گفته شد، DevOps بیشتر یک مفهوم فرهنگی است، یعنی دقیقا چیز خاصی نیست که آن را پیاده سازی کنید. نیاز داریم تا دیوار بین افراد و تیم ها شکسته شود تا آنها تعامل خوبی با هم داشته باشند و اهداف متضاد آنها تبدیل به اهداف مشترک شود.

۲-**A**utomation

در اینجا دقیقا مفاهیم Continuous Integration – Continuous Deployment – Continuous Delivery مطرح می شود، امکان ندارد شما ادعا کنید ما فرآیندهای DevOps را داریم ولی از ابزارهای مثلا CI استفاده نمی‌کنیم و همه کارها را دستی انجام می شود. فرآیندهای دستی کند هستند و امکان خطای انسانی در آنها زیاد است. برای همین تا آنجایی که امکان دارد باید تمام فرآیند تحویل محصول (از کامپیوتر برنامه نویس‌ها تا مشتری واقعی) اتوماتیک شده باشند.

۳-**L**ean

تکیه بر اصول اصلی تولید ناب نرم افزار که در اینجا کاملا به آن اشاره شده است. یکی از اصول اصلی این تفکر، از بین بردن تمامی فرآیندها و کارهای زاید است. یعنی هر ویژگی، فرآیند، فعالیتی که تولید ارزش نمی‌کنند باید حذف شوند. ناب بر ارزش محور بودن فعالیت‌ها و کاهش هر نوع فعالیت غیر ارزشمندی تاکید دارد.

برای مثال، کوچک بودن تیم‌های توسعه، توسعه ویژگی‌هایی که مشتری واقعا نیاز دارد، کم کردن دوباره کاری، کم کردن Task Switch و. ...

۴-**M**easurement

تا زمانی که ندانیم کجا هستیم، نخواهیم دانست که کجا می‌خواهیم برویم.

برای ایجاد یک فرآیند خوب و منظم، نیاز به شفافیت در کلیه سطوح داریم، برای ایجاد شفافیت و تصمیم‌گیری بهتر، نیاز داریم تا بتوانیم وضعیت موجود را ارزیابی کنیم. معمولا در هر سطح نرم افزار برای نوع سرویسی به چنین مانیتورینگ‌هایی نیاز است:

- Infrastructure Monitoring
- Log Management

- Application and Performance Management

اما فقط چنین اندازه گیری‌هایی برای حداکثری کردن ارزش کافی نیست، گاها نیاز است نرخ تبدیل مشتریان، میزان استفاده از هر ویژگی ، تعداد باگ‌های هر نسخه، سرعت میانگین تحویل هر نسخه و هر متر و معیاری که در حداکثری کردن ارزش به ما کمک می کنند را بدانیم.

Sharing-۵

این مفهوم در مورد اشتراک گزاری درس‌های یادگرفته است. ما از اندازه گیری‌ها و مانتیتورینگ چه درسی گرفتیم؟ پیشتر وجود دیوار مابین اعضای تیم باعث می‌شد این درس‌ها بین اعضای تیم پخش نشوند ، اشتباهات مکررا تکرار می‌شد و کارها صرفا با غر زدن پیش می‌رفت.

اینکه درس بگیریم که دیگر اشتباهات را تکرار نکنیم، یا اقدامات پیشگیرانه انجام دهیم و ... .

کاربرد DevOps کجاست؟

اگر شما یک سرویس یا محصولی تولید می‌کنید که دائم بر اساس نظرات مشتری یا بازخورد بازار تغییر می‌کند و ویژگی های جدید به آن اضافه می‌شود و فکر می‌کنید مزیت رقابتی شما ارائه سرویس خوب به مشتری است، پس احتمالا باید بدنبال این مفهوم باشید. اما معمولا اگر در سازمان‌هایی هستید که سرویس‌هایی با تکنولوژی‌های خیلی قدیمی وجود دارند و اصولا همه چیز دستی انجام می‌شود و روال‌های سازمانی اجازه اتوماتیک شدن به شما را نمی‌دهند، شاید استفاده از این مفهوم کار بسیار سختی باشد.

چگونه در کمتر از شش ماه به یک مهندس دواپس (DevOps) تبدیل شویم؟ (بخش اول)

https://arcademy.ir/article/375

آیا شما یک دولوپر هستید و می‌خواهید که مسیر شغلی خودتان را به سمت یک مدل دواپس‌تر تغییر بدهید؟

آیا شما به عنوان یک فرد عملیاتی آموزش دیده‌اید و می‌خواهید طعمی از دنیای دواپس را تجربه کنید؟

یا این که هیچ کدام از این‌ها نیستید. مدت زمانی با تکنولوژی کار کرده‌اید و الان به دنبال تغییر مسیر شغلی خودتان هستید و هیچ ایده‌ای ندارید که از کجا باید شروع کنید؟

اگر این طور است به مطالعه‌ای این مطلب بپردازید، چرا که ما به شما خواهیم گفت که چگونه در شش ماه به یک مهندس دواپس در سطح متوسط تبدیل بشوید.

این دیگر چیست؟

دواپس یک روش ارائه‌ی نرم‌افزار با مسئولیت مشترک است.

این یعنی که به طور مرسوم، دولوپرها (افرادی که نرم‌افزار را تولید می‌کنند) این حس را داشتند که با عملیاتی‌ها (افرادی که نرم‌افزار را اجرا می‌کنند) تفاوت بسیاری دارند.

به عنوان نمونه، من به عنوان یک دولوپر می‌خواهم که در سریع‌ترین زمان ممکن هر چقدر که می‌توانم ویژگی‌های جدیدی به وجود بیاورم. چرا که کار من همین است و مصرف‌کنندگان هم به دنبال این ویژگی‌ها هستند.

اما از طرفی دیگر اگر یک فرد عملیاتی باشم، تمایل دارم که تا حد امکان ویژگی‌های جدید کمتری تولید بشود. به این دلیل که هر ویژگی جدید نشان دهنده‌ی تغییر است و هر تغییری نیز دارای ریسک است.

به دلیل ناسازگاری این دو طرف، دواپس به وجود آمد.

دواپس (DevOps) تلاش می‌کند که توسعه (Development) و عملیات (Operations) را با یکدیگر در یک گروه ترکیب کند. عقیده‌ی پشت آن این است که با این کار یک گروه مسئولیت تولید، پیاده‌سازی و کسب درآمد از بخش مربط به مشتری نرم‌افزار را بر عهده خواهد داشت.

مهندس دواپس کسی است که چرخه‌ی زندگی توسعه‌ی نرم‌افزار را می‌فهمد و با کمک ابزارها و فرآیندهای مهندسی نرم‌افزار سعی می‌کند که چالش‌های عملیاتی کلاسیک را برطرف کند.

دواپس در نهایت به این معناست که یک لوله‌کشی دیجیتال انجام بدهیم تا کد از لپتاپ دولوپر تا کسب درآمد حرکت کند.

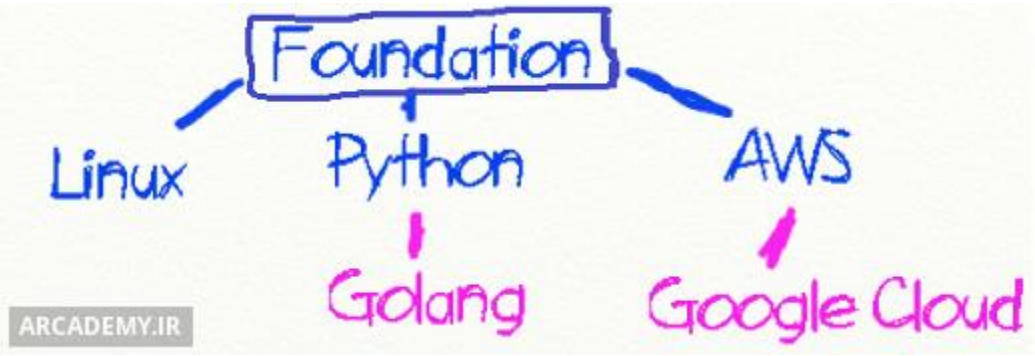
برای تبدیل شدن به یک مهندس دواپس کارکشته نیازمند سال‌های زیادی تجربه و درک بالایی از ابزارهای کاربردی است و متاسفانه هیچ میانبری برای کسب تجربه وجود ندارد.

مهندس دواپس به دنبال – **ایجاد یک لوله‌کشی دیجیتال و خودکار که ایده‌ها را به کدهایی تبدیل می‌کند که باعث تولید سرمایه می‌شوند** – هست.

نقشه‌ی راه DevOps

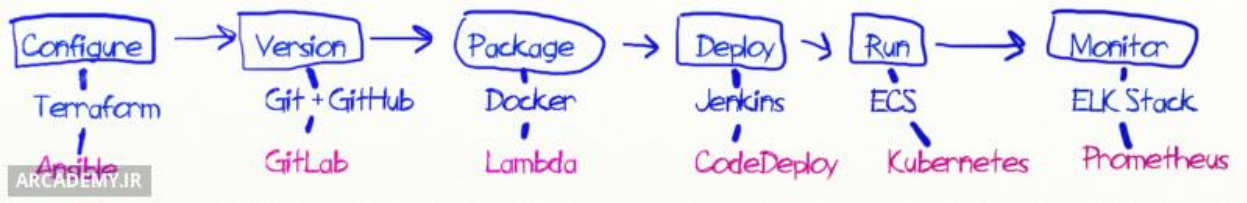
همین نقشه راه(plan) را یاد بگیرید تا بتوانید با اطمینان خودتان را مهندس دواپس خطاب کنید. نقشه‌ی راه زیر درباره‌ی چیزهایی که یک مهندس کاربلد دواپس باید بداند را نشان می‌دهد.

نکته: شما باید این نقشه را لایه به لایه و به صورت عرضی پیمایش کنید. با پایه و اساس شروع کنید و ادامه بدهید. تکنولوژی‌های نوشته شده به رنگ آبی را قبل از بقیه یاد بگیرید (Linux, Python, AWS) سپس اگر زمانی در اختیار داشتید یا بازار کار نیازمند آن بود به سراغ موضوعات بنفش بروید .(Golang, Google Cloud)



پایه و اساس ذکر شده در بالا چیزی است که هیچ وقت نمی‌توانید یاد گرفتن آن را متوقف کنید. لینوکس بسیار پیچیده است و یاد گرفتن آن به طور کامل نیازمند سال‌ها زمان است. پایتون نیاز دارد که دائما روی آن کار شود تا با پیشرفت‌های روز همگام بمانید. AWS (خدمات وب آمازون) به قدری سریع پیشرفت می‌کند که چیزی که امروز می‌دانید تا یک سال دیگر بخش کوچکی از نمای کلی آن خواهد بود.

اما به محض این که از پایه و اساس سر در آوردید، به سراغ توانایی‌های مورد نیاز در دنیای واقعی بروید. توجه کنید که شش ستون آبی رنگ وجود دارد که هر کدام آن‌ها برای یک ماه است.



نکته: چیزی که به طور واضحی در شکل بالا وجود ندارد تست کردن است. این کار از قصد صورت گرفته است. نوشتن تست واحد(UnitTest)، ادغام و پذیرش کار آسانی نیست و معمولا بر عهده‌ی دولوپر قرار می‌گیرد. حذف مرحله تست کاملا عمدی بوده، چرا که هدف این مطلب این است که برداشتی سریع از توانایی‌ها و ابزارهای جدید به دست بیاورید. بی‌تجربگی در زمینه‌ی تست کردن به نظر نویسنده مانع بزرگی برای رسیدن به یک سطح مطلوب از دواپس نیست.

ما به دنبال ایجاد یک فهم قوی از کلیت ماجرا هستیم. این ماجرا هم عبارت است از خودکارسازی نقطه به نقطه‌ی فرآیند – یک لوله‌کشی دیجیتال که قطعات را به صورت خط تولید جا به جا می‌کند. در ضمن، نباید به دنبال این باشید که تعدادی از ابزار را فرا بگیرید و متوقف شوید. ابزارها به سرعت تغییر می‌کنند اما مفاهیم کمتر. پس در حالت ایده‌آل باید از ابزارها برای یاد گرفتن مفاهیم سطح بالاتر بهره ببرید.

دانش بنیانی

زیر عبارت Foundation می‌توانید توانایی‌های مورد نیاز برای یک مهندس دواپس را مشاهده کنید.

در این‌جا سه بخش اصلی این صنعت نشان داده شده است: سیستم عامل، زبان برنامه‌نویسی و فضای ابری عمومی. شما نمی‌توانید این موارد را در اسرع وقت یاد بگیرید و از لیست خط بزنید و به سراغ مورد بعدی بروید. این توانایی‌ها نیازمند صبر و تلاش پیوسته هستند تا همراه به روز باقی بمانند.

لینوکس : مکانی که همه چیز در آن اجرا می‌شود. در این صورت، آیا می‌توانید در زمینه‌ی دواپس فعالیت کنید و به طور کل در اکوسیستم مایکروسافت باقی بمانید؟ البته که می‌توانید! در لینوکس هیچ قانونی وجود ندارد!

با این وجود لازم است بدانید که تمامی امور مربوط به دواپس در سیستم عامل ویندوز هم انجام شدنی هستند، اما انجام آن‌ها دشوارتر می‌شود و فرصت‌های شغلی کمتری در این زمینه وجود دارد. در حال حاضر این گونه تصور کنید که بدون آگاهی از لینوکس نمی‌توانید به یک مهندس دواپس واقعی تبدیل بشوید. پس چیزی که باید به دنبالش باشید لینوکس است.

پایتون : امروزه دنیای بک‌اند در اختیار زبان پایتون است. یادگیری آن بسیار آسان است و به همین دلیل بسیار از آن استفاده می‌شود. یکی دیگر از مزیت‌های پایتون این است که در حیطه‌ی هوش مصنوعی و یادگیری ماشین نیز کاربرهای فراوانی دارد. پس اگر روزی بخواهید به سراغ زمینه‌ی شغلی دیگری بروید نیز به دردتان خواهد خورد.

خدمات وب آمازون :مجددا باید بگویم که نمی‌توان بدون داشتن یک دانش مفصل از طرز کار فضای ابری عمومی تبدیل به یک مهندس دواپس شد و اگر آگاهی از فضای ابری هدف شماست، فضایی بهتر از خدمات وب آمازون پیدا نخواهید کرد AWS . بهترین ابزارها را در اختیار شما می‌گذارد تا به کمک آن‌ها کارهایتان را پیش ببرید.

آیا ممکن است که با فضای ابری گوگل یا مایکروسافت آژور شروع به کار کرد؟ البته که ممکن است! اما ما به دنبال بزرگترین بازیکن در این عرصه هستیم و به همین دلیل انتخاب AWS در سال ۲۰۱۹ معقول‌تر است. در زمان ثبت‌نام نیز یک فضای رایگان در اختیارتان قرار می‌گیرد که برای شروع فرصت بدی نیست.

پس از این که وارد کنسول AWS می‌شوید یک منوی ساده و قابل فهم پیش روی شما قرار می‌گیرد. و نیاز ندارید از تمامی تکنولوژی‌های آمازون آگاهی داشته باشید.

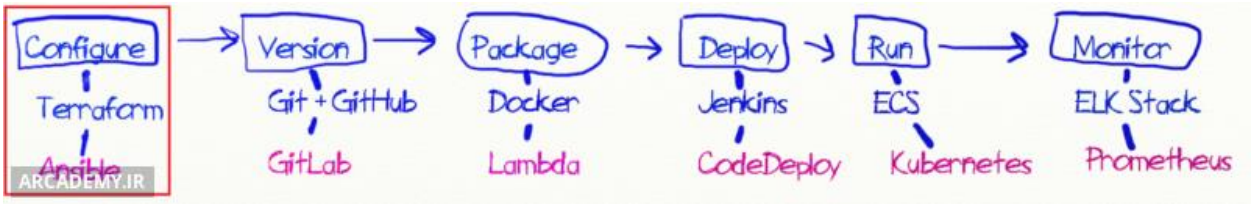
با مواردی از قبیل VPC، EC2، IAM، S3، CloudWatch، ELB و گروه‌های امنیتی شروع کنید. برای شروع این‌ها موارد بسیار خوب و مدرنی هستند که بسیاری از شرکت‌های ابری از این ابزارها استفاده‌های فراوانی می‌کنند.

ضمنا وبسایت آموزش AWS نیز مکان خوبی برای شروع به کار است.(https://www.aws.training/?src=training)

اکنون با لایه‌ی پایه و اساس آشنا شده‌اید. در لایه‌های بعدی سطح‌های بیشتری از پیچیدگی را بررسی خواهیم کرد. پیکربندی، نسخه، بسته‌بندی، به‌کارگیری، اجرا و نظارت نرم‌افزار، هر کدام به روشی کاملا خودکار!

چگونه در کمتر از شش ماه به یک مهندس دواپس (DevOps) تبدیل شویم؟ بخش دوم: پیکربندی

در این مقاله اولین مرحله از تولید خط لوله‌ی دیجیتال را پوشش می‌دهیم: تنظیم.



نمای کلی

در مرحله‌ی تنظیم چه اتفاقی می‌افتد؟

واضح است که کد نوشته شده‌ی ما برای اجرا شدن به یک ماشین نیاز دارد. در مرحله‌ی اجرا زیرساختی که کد ما را اجرا می‌کند ساخته می‌شود.

در گذشته فراهم‌سازی زیرساخت می‌توانست فرآیندی زمان‌بر، طاقت‌فرسا و همراه با مشکل باشد.

امروزه به دلیل فضای ابری فوق‌العاده‌ای که در اختیار ما است، تمامی امور مربوط به فراهم‌سازی زیرساخت‌ها با یک یا چند کلیک انجام شدنی است.

با این وجود به نظر می‌رسد که کلیک کردن برای انجام این کارها ایده‌ی چندان جالبی نیست. چرا؟

به دلیل این که کلیک کردن ویژگی‌های زیر را به همراه دارد:

- می‌تواند همراه با خطا باشد (انسان دچار اشتباه می‌شود).
- نسخه‌بندی صورت نمی‌گیرد(کلیک‌ها را نمی‌توان در git ذخیره کرد).
- تکرارپذیر نیست (ماشین‌های بیشتر یعنی کلیک‌های بیشتر).
- و در نهایت قابل آزمایش نیست (هیچ ایده‌ای نداریم که کلیک‌های ما به درستی کار خواهند کرد یا این که چیزهای دیگری را به هم می‌ریزند).

به عنوان مثال، به کارهایی که برای فراهم‌سازی محیط توسعه باید انجام بدهید فکر کنید، پس از آن محیط int، سپس پرسش و پاسخ، سپس صحنه‌سازی، سپس تولید در کشور، سپس تولید در سایر کشورها. این کارها می‌توانند به سرعت کسل‌کننده شوند و شما را آزار بدهند.

پس یک روش جدید لازم است. این روش جدید **زیرساخت به عنوان کد** نام دارد که مرحله‌ی تنظیم در آن خلاصه می‌شود.

زیرساخت به عنوان کد می‌گوید که هر کاری که برای فراهم‌سازی منابع پردازشی باید انجام بشود، فقط و فقط از طریق کدنویسی صورت بگیرد.

نکته: منظورم از منابع پردازشی تمامی چیزهای لازم برای اجرای درست یک برنامه در فاز تولید است: پردازش، حافظه، شبکه، پایگاه‌داده و غیره. به همین دلیل آن را زیرساخت به عنوان کد می‌نامند.

علاوه بر این، به جای کلیک کردن در طول یک زیرساخت، کارهای زیر را انجام می‌دهیم:

- وضعیت زیرساخت مد نظرمان را در Terraform می‌نویسیم.
- آن را در کنترل سورس کدمان ذخیره می‌کنیم.
- درخواست رسمی برای دریافت فیدبک ارائه می‌دهیم.
- آن را آزمایش می‌کنیم.
- آن را اجرا می‌کنیم تا از منابع مورد نیاز مطلع شویم.

{ Terraformیک زیرساخت منبع باز به عنوان ابزار نرم افزار کد است که توسط HashiCorp ایجاد شده است. این برنامه کاربران را قادر می سازد با استفاده از یک زبان پیکربندی سطح بالا موسوم به Hashicorp Language Configuration (HCL) یا اختیاری JSON ، یک زیرساخت دیتاسنتر را تعریف و تأمین کند Terraform [3] .از تعدادی از ارائه دهندگان زیرساخت ابری مانند خدمات وب آمازون ، IBM Cloud (سابقا Bluemix ، Google Cloud Platform، DigitalOcean [4] [6] [5] Linode، Microsoft Azure، Oracle Cloud Infrastrast، OVH پشتیبانی می کند.}

چرا این بله و آن نه؟

حالا سوالی که مطرح می‌شود این است که چرا از Terraform استفاده کنیم؟

سوال خوبی است! و البته طبق معمول حجم زیادی از جوهر مجازی بر سر بحث روی این موضوع در اینترنت به هدر رفته است.

اگر به طور خلاصه بخواهم بگویم، به این دلایل باید Terraform را یاد بگیرید:

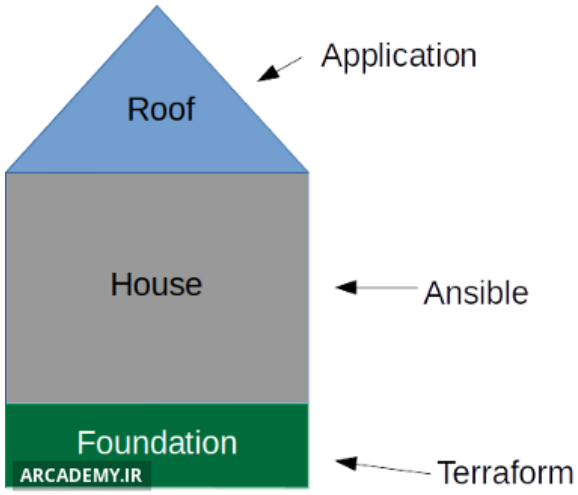
- در بازار کار بسیار مرسوم است که باعث می‌شود فرصت‌های شغلی بیشتری داشته باشید.
- یادگیری آن تا حدی آسان‌تر از سایر گزینه‌ها است.
- به پلتفرم خاصی محدود نیست.

با این وجود آیا می‌توانید گزینه‌ی دیگری را انتخاب کنید و موفق شوید؟ البته که بله!

اما باید اشاره کنم که این فضا با سرعت زیادی در حال رشد است و می‌تواند گیج‌کننده باشد. در ادامه کمی از اتفاقات اخیر و پیشرفت‌هایی که در آینده ممکن است رخ بدهد می‌گویم.

به طور مرسوم، ابزارهایی مانند Terraform یا CloudFormation در زمینه‌ی فراهم‌سازی زیرساخت‌ها استفاده شده‌اند و ابزارهایی مانند Ansible برای تنظیم آن به کار گرفته می‌شوند.

می‌توانید Terraform را به عنوان ابزاری برای ایجاد پایه و اساس و Ansible را به عنوان خانه‌ای که روی آن قرار می‌گیرد در نظر داشته باشید، سپس برنامه هر گونه که بخواهید می‌توانید استفاده کنید (به عنوان مثال این مورد هم می‌تواند Ansible باشد).



به عبارتی دیگر، به کمک Terraform ماشین‌های مجازی‌تان را می‌سازید و با Ansible سرورهایش را راه‌اندازی می‌کنید. ضمن این که می‌توانید برنامه‌تان را نیز به کار بگیرید.

به طور مرسوم این موارد به همین شکل در کنار یکدیگر کار می‌کردند.

اما Anisble می‌تواند بسیاری از کارهایی که Terraform انجام می‌دهد را به تنهایی انجام دهد (حتی ممکن است بتواند تمامی کارها را انجام دهد). پس معکوس آن نیز صادق است.

اجازه ندهید که این شما را گیج کند. تنها کافی است که بدانید Terraform یکی از بزرگترین ابزارهای حاضر در زمینه‌ی زیرساخت به عنوان کد است، پس پیشنهاد من این است که از همین مورد شروع کنید.

در واقع توانایی کار با Terraform در کنار خدمات وب آمازون یکی از کاربردی‌ترین ویژگی‌هایی است که می‌توانید به دست بیاورید.

با این وجود، اگر Ansible را به Terraform ترجیح می‌دهید، باز هم نیاز است که تعداد زیادی از سرورها را با برنامه‌نویسی راه‌اندازی کنید، این طور نیست؟

الزاما نه!

به‌کارگیری‌های تغییر ناپذیر

اما اگر نظر صادقانه‌ام را بگویم، پیشبینی می‌کنم که ابزارهای مدیریت تنظیمات مانند Ansible با گذر زمان از اهمیتشان کاسته شود و در مقابل ابزارهای فراهم‌سازی زیرساخت مانند Terraform یا CloudFormation به اهمیتشان افزوده شود.

چرا؟

به دلیل چیزی که به‌کارگیری‌های تغییر ناپذیر نام دارد.

ساده بگویم، به‌کارگیری‌های تغییر ناپذیر به این امر اشاره دارند که زیرساخت به کارگیری شده را هیچ وقت دستکاری نکنید. به عبارتی دیگر، واحد شما یک ماشین مجازی یا Docker Conrainer است، نه یک قطعه کد.

شما کد را در یک مجموعه از ماشین‌های مجازی استاتیک راه‌اندازی نمی‌کنید، بلکه کل ماشین مجازی را به همراه کد درون آن راه می‌اندازید.

شما تنظیمات ماشین‌های مجازی را تغییر نمی‌دهید، بلکه ماشین‌های مجازی جدیدی با تغییرات اعمال شده‌ی مد نظرتان را راه‌اندازی می‌کنید.

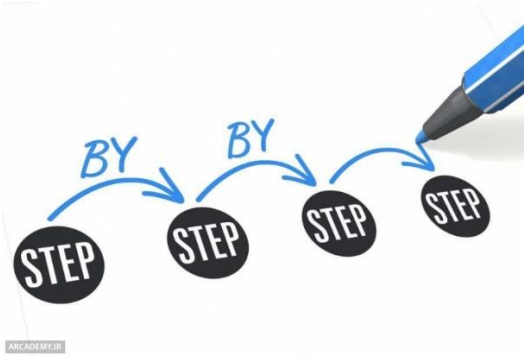
شما ماشین‌های تولید شده را پچ نمی‌کنید، بلکه ماشین‌های جدیدی راه‌اندازی می‌کنید که از قبل پچ شده‌اند.

شما نمی‌توانید یک مجموعه از ماشین‌های مجازی را در توسعه به کار بگیرید و سپس در تولید از یک مجموعه ماشین مجازی دیگر استفاده کنید. همه‌ی آن‌ها یکسان هستند.

در واقع شما می‌توانید با اطمینان تمامی دسترسی‌های SSH به ماشین‌های تولید شده را قطع کنید، به این دلیل که دیگر کاری برای انجام دادن وجود ندارد. نه تنظیماتی را می‌توان تغییر داد و نه لاگی وجود دارد که به آن توجه شود (در ادامه بیشتر به لاگ‌ها می‌پردازیم).

منظورم را فهمیده‌اید.

زمانی که به درستی استفاده شود، این الگو می‌تواند بسیار قدرتمند باشد و پیشنهاد من هم همین است.



نکته: به‌کارگیری‌های تغییر ناپذیر الزام می‌کنند که تنظیمات از کد جدا باشند. می‌توایند مانیفست 12 Factor App را مطالعه کنید که به طور جزئی به این مسئله و بسیاری ایده‌های خوب دیگر می‌پردازد. افرادی که در زمینه‌ی دواپس کار می‌کنند حتما باید این را مطالعه کنند.

جداسازی کد از تنظیمات بسیار مهم است. یقینا نمی‌خواهید که با هر تغییری در پسوردهای پایگاه‌داده، تمامی پشته‌ی برنامه را مجددا راه‌اندازی کنید. در عوض اطمینان حاصل کنید که برنامه آن را از یک فضای تنظیمات خارجی دریافت می‌کند(SSM/Consul/etc.)

علاوه بر این به راحتی قابل مشاهده است که با رشد به‌کارگیری‌های تغییر ناپذیر، ابزارهایی مانند Ansible نقش کم‌رنگ‌تری خواهند داشت.

دلیلش این است که تنها نیاز دارید که یک سرور را تنظیم کنید و چندین بار به عنوان بخشی از گروه مقیاس‌پذیری خودکار خود راه‌اندازی کنید.

یا اگر با کانتینرها کار می‌کنید، به طور قطع نیازمند به‌کارگیری‌های تغییر ناپذیر هستید. کانتینر توسعه‌ی شما به هیچ وجه نباید با کانتینر پرسش و پاسخ یا کانتینر تولید تفاوتی داشته باشد.

کانتینر باید در تمامی محیط‌های کاری یکسان باشد. با این کار می‌توان از بیراهه رفتن تنظیمات جلوگیری کرد و در موارد وقوع مشکل به راحتی اعمال بازیابی و بازگشت به عقب را انجام داد.

کانتینرها را کنار بگذاریم. افرادی که تازه شروع کرده‌اند باید بدانند که فراهم‌سازی زیرساخت خدمات وب آمازون به کمک Terraform یک الگوی واجب برای دواپس است و حتما باید فراگرفته شود.

اما صبر کنید! اگر مجبور شوم که برای برطرف کردن یک مشکل به لاگ‌ها نگاه کنم چه؟ خب دیگر برای نگاه کردن به لاگ‌ها با ماشین ارتباط برقرار نمی‌کنید، بلکه به سراغ زیرساخت ارتباط مرکزی می‌روید تا لاگ‌هایتان را مشاهده کنید.

باز هم می‌گویم که می‌توانید تمامی دسترسی‌های از راه دور را غیرفعال کنید و خوشحال باشید که از بسیاری افراد دیگر امنیت بیشتری دارید!

اگر بخواهم مطالب گفته شده را خلاصه کنم، ماجراجویی ما در زمینه‌ی دواپس کاملا خودکار با تأمین منابع پردازشی مورد نیاز برای اجرای کدمان شروع می‌شود که مرحله‌ی تنظیم نام دارد. بهترین راه دستیابی به آن نیز از طریق به‌کارگیری‌های تغییر ناپذیر است.

در انتها اگر هنوز هم دقیقا نمی‌دانید که از کجا باید شروع کنید، پیشنهاد من این است که ترکیب Terraform در کنار خدمات وب آمازون مکان ایده‌آلی برای آغاز به کار است.

با این توضیحات مرحله‌ی تنظیم به پایان می‌رسد.

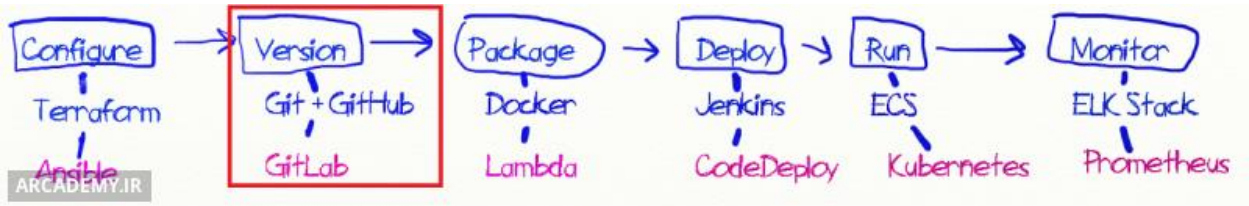
قسمت سوم این مقاله که مربوط به نسخه و یا همان Version می‌باشد را می‌توانید از اینجا مطالعه کنید.

در بخش دوم به معرفی Terraform و روش فراهم‌سازی بستر استفاده از کد در آینده به کمک آن پرداختیم.

در ادامه و به طور ویژه در این مطلب بررسی می‌کنیم که چگونه می‌توانیم بخش‌های مختلف کد را با نظم جدا از هم نگه داریم و از درهم ریختگی آن‌ها جلوگیری کنیم. اسپویلر! همه‌ی این‌ها کار گیت است. همچنین درباره‌ی روش استفاده از گیت برای ایجاد و ترویج برند شخصی خودتان نیز توضیحاتی ارائه می‌دهیم.

چگونه در کمتر از شش ماه به یک مهندس دواپس (DevOps) تبدیل شویم؟ بخش سوم: نسخه

برای یادآوری باید اشاره کنم که در حال حاضر در این نقطه از ماجرا هستیم:



چه نیازی به این کار است؟

وقتی از نسخه‌گذاری صحبت می‌کنیم، منظورمان چیست؟

تصور کنید که در حال کار بر روی قطعه‌ای از یک نرم‌افزار هستید. دائما آن را تغییر می‌دهید و بر حسب نیاز ویژگی‌هایی را به آن اضافه می‌کنید یا از آن حذف می‌کنید. معمولا آخرین تغییری که ایجاد می‌کنید یک تغییر "شکننده" خواهد

بود. به عبارتی دیگر، هر کاری که آخرین بار انجام دادید، باعث شد که زحماتی که تا آن مرحله کشیده بودید از کار بیافتند.

در این صورت باید چه کاری انجام داد؟

اگر بخواهید به روش گذشته عمل کنید، احتمالا اولین قطعه کدتان را این گونه نام‌گذاری خواهید کرد:

awesome_code.p1

سپس شروع به اعمال تغییرات می‌کنید و می‌خواهید قطعه کدی که سالم کار می‌کند را نگه دارید تا در صورت نیاز به سراغ آن بروید.

پس فایل کدتان را به این شکل نام‌گذاری می‌کنید:

awesome_code.01.01.2019.p1

تا این لحظه مشکلی وجود ندارد. اما ممکن است یک روز بخواهید بیش از یک تغییر اعمال کنید، پس مجبور می‌شوید که به چنین شکلی کدتان را ذخیره کنید:

awesome_code.GOOD.01.01.2019.p1

و الی آخر.

البته در یک محیط حرفه‌ای، تیم‌های متعددی به طور همزمان در حال کار بر روی یک کد پایه هستند که باعث می‌شود این مدل پیچیده‌تر از این بشود.

کنترل سورس کد

در این مرحله به سراغ کنترل سورس کد می‌رویم: یک روش برای نگهداری فایل‌ها در یک مکان متمرکز که در آن چندین تیم می‌توانند با یکدیگر روی یک کد پایه‌ی مشترک فعالیت کنند.

این ایده‌ی جدیدی نیست. برای اولین بار در سال ۱۹۷۲ از به‌کارگیری چنین روشی صحبت شد. پس این که باید کدهایمان را در یک مکان متمرکز ذخیره کنیم به هیچ وجه چیز جدیدی نیست.

با این حال چیزی که می‌توان آن را جدید نامید، ایده‌ی نسخه‌گذاری تمامی مصنوعات تولید شده است.

اما این یعنی چه؟

این یعنی هر چیزی که در محیط تولید شما به کار گرفته می‌شود باید طبق پیگیری، بررسی و تاریخچه‌ی تغییرات در کنترل نسخه ذخیره شود. علاوه بر این، پیاده‌سازی قانون "تمامی مصنوعات تولیدی باید نسخه‌گذاری شوند" باعث می‌شود که با ذهنیت اتوماسیون به سراغ مشکلات بروید.

برای نمونه، وقتی تصمیم می‌گیرید که در محیط Dev AWS خود با کلیک کردن از یک مشکل پیچیده بگذرید، ممکن است با خود فکر کنید که آیا این کلیک کردن‌ها یک مصنوع نسخه‌گذاری شده است؟

البته که جواب این سوال خیر است! با وجود این که ایجاد سریع نمونه‌های اولیه با کمک رابط کاربری برای چک کردن مشکلات ایرادی ندارد، اما این اعمال باید در زمان کوتاهی پایان بیابند. اگر قصد دارید که بلند مدت از آن استفاده کنید، مطمئن شوید که تمامی کارها را در Terraform یا یک ابزار مشابه آن انجام می‌دهید.

بسیار خب، فرض بر این است که همه چیز یک مصنوع نسخه‌گذاری شده است. حالا چگونه باید این موارد را ذخیره و مدیریت کنیم؟

جواب این سوال در گیت خلاصه می‌شود.

گیت

تا قبل از روی کار آمدن گیت، سیستم‌های کنترل سورس کد مانند SVN ساده و کاربرپسند نبودند و در مجموع تجربه‌ی جالبی به کاربر القا نمی‌کردند.

تفاوت گیت با سایرین در این است که گیت مفهوم کنترل توزیع‌شده‌ی سورس کد را به خوبی پذیرفته است.

به عبارتی دیگر، زمانی که شما در حال کار بر روی کد هستید، سایر تیم‌ها نیز قابلیت دسترسی همزمان به آن کد را دارند و سیستم برای آن‌ها قفل نمی‌شود. شما بر روی یک کپی از کد تغییرات را اعمال می‌کنید و سپس آن کپی در فضای اصلی ذخیره‌سازی ادغام می‌شود.

در نظر داشته باشید که این تنها یک توضیح بسیار ساده از طرز کار گیت است. اما برای کاربرد ما در این مطلب همین میزان کفایت می‌کند. با این حال نگاهی مفصل به طرز کار درونی گیت می‌تواند برای یک مهندس دواپس بسیار کارآمد باشد.

در حال حاضر همین را بدانید که گیت مانند SVN نیست. گیت یک سیستم کنترل سورس کد توزیع‌شده است که در آن چندین تیم می‌توانند با اطمینان روی یک کد مشترک کار کنند.

چه نیازی به این کار است؟

رک و راست بگویم، به عقیده‌ی من امکان ندارد که بدون آگاهی از طرز کار گیت بتوانید به یک مهندس دواپس حرفه‌ای تبدیل شوید. تمام!

متاسفانه اگر در گوگل آموزش گیت را جستجو کنید، نتایج متعدد و معمولا پیچیده‌ای به نمایش گذاشته می‌شود که می‌تواند گیج‌کننده باشد. اما آموزش‌هایی نیز وجود دارد که بسیار خوب و مفید هستند.

یکی از این مجموعه‌ها که به همه توصیه می‌کنم آن را مطالعه کنند و یاد بگیرند، آموزش‌های گیت Atlassian است. همه‌ی بخش‌های این آموزش بسیار جالب است اما یک بخش آن توسط تمامی مهندسان نرم‌افزار حرفه‌ای در سراسر دنیا استفاده می‌شود و آن هم روندهای کاری گیت است.

یکی دیگر از آموزش‌های مناسب برای گیت، Learn Git Branching است.

آموزش‌های Atlassian به صورت خواندنی و آموختنی است، در صورتی که آموزش‌های Learn Git Branching به صورت اینترکتیو است(https://learngitbranching.js.org/) و کاربر را درگیر می‌کند. در هر صورت نیاز است که گیت را بیاموزید، چرا که بدون داشتن درک درستی از طرز کار گیت نمی‌توانید پیشرفت به خصوصی در شغلتان داشته باشید.

حس می‌کنم باز هم باید تاکید کنم! بارها و بارها اتفاق افتاده است که عدم آگاهی از ویژگی شاخه‌ای (انشعابی) گیت یا ناتوانی در توضیح طرز کار Gitflow باعث شده که کاندیداهای مهندسی دواپس در یک شرکت پذیرفته نشوند.

نکته‌ی اساسی همین است. شما می‌توانید برای مصاحبه وارد شرکت شوید و از Terraform یا ابزارهای زیرساخت به عنوان کد روز آگاهی نداشته باشید. این ایرادی ندارد. می‌توان حین کار بر آن تسلط پیدا کرد. اما ناآگاهی از گیت و طرز کار آن به مدیران اعلام می‌کند که شما درک اساسی از بهترین شیوه‌های مهندسی نرم‌افزار مدرن یا دواپس را ندارید و شیب یادگیری شما بسیار کند خواهد بود! یقینا شما هم نمی‌خواهید که چنین برداشتی از شما داشته باشند.

اما بر خلاف آن، اگر بتوانید با اعتماد به نفس از گیت و بهترین شیوه‌های کار با آن صحبت کنید، به مدیران شرکت‌ها اعلام می‌کنید که یک ذهنیت عالی برای مهندسی نرم‌افزار در اختیار دارید و این دقیقا چیزی است که شما می‌خواهید به آن‌ها نشان بدهید.

پس به طور خلاصه: نیازی نیست که حرفه‌ای‌ترین متخصص گیت باشید تا بتوانید شغل دواپس مورد نظرتان را برای خود کنید، اما باید مدتی با گیت زندگی کنید و پیچ و خم‌های آن را بدانید تا بتوانید با اعتماد به نفس درباره‌ی آن صحبت کنید و تصویر درستی از خودتان به کارفرما ارائه بدهید.

در کمترین حالت باید بتوانید کارهای زیر را انجام بدهید:

۱. یک مخزن را فورک کنید.
۲. شاخه (انشعاب) ایجاد کنید.
۳. تغییرات رو به عقب و رو به جلو را ادغام کنید.
۴. درخواست Pull ارائه بدهید.

قدم بعد چیست؟

بعد از این که آموزش‌های مقدماتی گیت را گذراندید، برای خودتان یک اکانت در گیت‌هاب بسازید.

نکته : گیت‌لب نیز گزینه‌ی مناسبی است اما در زمان نوشته شدن این مطلب، گیت‌هاب رایج‌ترین مخزن کد باز گیت است، پس سعی کنید جایی باشید که دیگران هم هستند.

پس از این که اکانت گیت‌هاب خودتان را ساختید، کدتان را در آن قرار دهید. مطمئن شوید هر چیزی که یاد می‌گیرید و نیاز به کد زدن دارد، به طور پیوسته در گیت‌هاب شما قرار می‌گیرد.

با این کار نه تنها نظم خوبی برای کنترل سورس کدتان دارید، بلکه به ایجاد برند شخصی شما نیز کمک می‌کند.

نکته : زمانی که در حال یادگیری گیت و گیت‌هاب هستید، توجه ویژه‌ای به درخواست Pull داشته باشید.

برند

برند روشی است که به دنیا نشان بدهید چه کارهایی از شما بر می‌آید.

یک راه (در حال حاضر یکی از بهترین راه‌ها!) این است که حضوری پررنگ در گیت‌هاب داشته باشید و از آن به عنوان یک پروکسی برای برندتان استفاده کنید. تقریبا تمامی کارفرماها چنین چیزی را از شما می‌خواهند.

به همین دلیل، سعی کنید که یک اکانت گیت‌هاب ایده‌آل داشته باشید و دائما به آن رسیدگی کنید، به طوری که بتوانید آن را روی رزومه‌ی خود قرار دهید و به آن افتخار کنید.

در بخش‌های بعد درباره‌ی ایجاد یک وبسایت ساده اما جذاب روی گیت‌هاب و با کمک فریمورک Hugo صحبت خواهیم کرد. اما فعلا تنها قرار دادن کدهایتان در گیت‌هاب کفایت می‌کند.

با مرور زمان و کسب تجربه، شاید بهتر باشد که دو اکانت در گیت‌هاب داشته باشید. یکی از آن‌ها برای موارد شخصی و ذخیره‌ی کدهای تمرینی و دیگری برای ذخیره‌ی کدهایی که می‌خواهید دیگران مشاهده کنند.

سخن پایانی

در انتها، نیاز است که به پیشرفت‌های اخیر در این زمینه مانند GitOps هم توجه داشته باشید.

GitOps تمامی مواردی که درباره‌شان صحبت کردیم را به سطح دیگری می‌برد که در آن تمامی کارها از طریق گیت، درخواست Pull و خط لوله‌های به‌کارگیری انجام می‌شود.

توجه داشته باشید که مخاطب GitOps و رویکردهای مشابه آن، قسمت تجاری داستان است. به این معنی که قرار نیست از موارد پیچیده‌ای مانند گیت به این دلیل که خفن هستند استفاده کنیم! بلکه می‌خواهیم از گیت برای بهبود چابکی کسب و کار، افزایش سرعت نوآوری و ارائه‌ی سریع‌تر ویژگی‌ها استفاده کنیم .چرا که در نهایت این‌ها باعث می‌شوند که کسب و کارمان به درآمد بیشتری برسد.

چگونه در کمتر از شش ماه به یک مهندس دواپس (DevOps) تبدیل شویم؟ بخش چهارم: پکیج

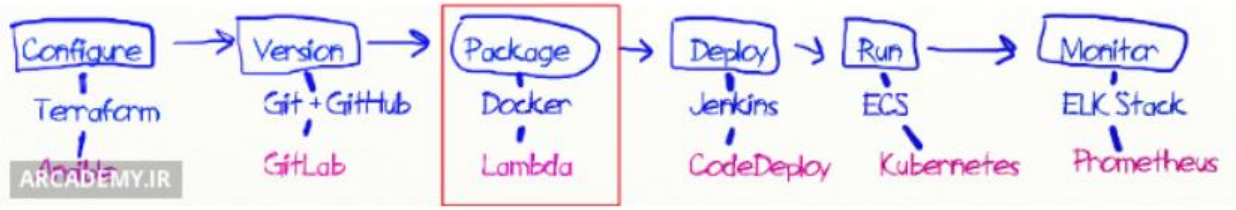
در بخش اول درباره‌ی فرهنگ دواپس و نیازمندی‌های آن صحبت کردیم.

در بخش دوم به راه و روش فراهم‌سازی بستر کد با در نظر داشتن پیاده‌سازی‌های آینده اشاره کردیم.

در بخش سوم نیز درباره‌ی مرتب و منظم نگه داشتن کارها و کدهایتان توضیحاتی ارائه دادیم.

حالا در این بخش بررسی می‌کنیم که چگونه یک کد را بسته‌بندی کنیم تا پیاده‌سازی و اجرای آن در آینده آسان و بی‌دردسر باشد.

برای یادآوری بیشتر، الان در این مرحله از ماجرای دواپس هستیم.



نکته : می‌توانید ببینید که هر بخش به کمک بخش قبلی ساخته می‌شود و همچنین بنیان‌گذار بخش بعد از خود است. این نکته اهمیت بالایی دارد و بی‌دلیل به این گونه طراحی نشده است.

دلیل آن هم این است که اگر در حال گفتگو با کارفرمای فعلی یا آینده‌ی خود هستید، باید بتوانید با مهارت هر چه تمام‌تر درباره‌ی دواپس و اهمیت آن صحبت کنید. این کار را هم با بیان یک داستان منسجم انجام می‌دهید. داستان این که چگونه در سریع‌ترین زمان و به بهینه‌ترین حالت ممکن کد را از لپتاپ دولوپر برداشته و به یک محصول درآمدزا تبدیل کنید.

از این رو، هدف ما این نیست که تعدادی مطلب نامرتبط به یکدیگر را یاد بگیریم، بلکه می‌خواهیم مجموعه‌ای از توانایی‌های مختلف را بدست بیاوریم که در بازار کار تقاضا دارند و ابزارهای فنی آن‌ها نیز در اختیارمان قرار دارد.

همچنین فراموش نکنید که باید به مدت یک ماه برای یادگیری هر بخش زمان بگذارید که مجموعا شش ماه طول می‌کشد.

الفبای مجازی‌سازی

سرورهای فیزیکی را به خاطر دارید؟ همان سرورهایی که باید هفته‌ها صبر می‌کردید تا تایید شوند، ارسال شوند، مرکز داده آن‌ها را بپذیرد، جایگذاری شوند، شبکه شوند، بر روی آن‌ها سیستم عامل نصب شود و پچ شوند.

بله، اول آن‌ها آمدند!

اساسا این طور تصور کنید که تنها راه برای خانه‌دار شدن این باشد که یک خانه‌ی جدید بسازید. یعنی وقتی که به یک مکان برای زندگی نیاز داشته باشید، باید صبر کنید تا از صفر ساخته شود! به نوعی خوب است چون همه صاحب خانه می‌شوند اما اصلا مطلوب نیست، چرا که ساختن یک خانه مدت زیادی زمان می‌برد .در این مثال سرور فیزیکی حکم یک خانه را دارد.

پس از مدتی کاربران از طولانی بودن این فرآیند گله‌مند شدند و تعدادی آدم بسیار باهوش ایده‌ی مجازی‌سازی به ذهنشان رسید: می‌توانیم چندین ماشین نمادین را روی یک ماشین فیزیکی اجرا کنیم و هر ماشین تقلبی ادای یک ماشین واقعی را دربیاورد. فوق‌العاده است!

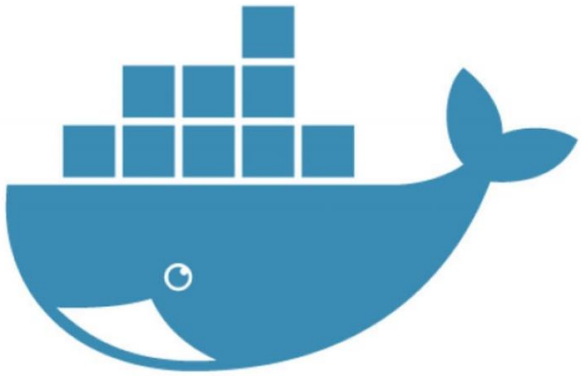
بسیار خب، پس اگر واقعا به دنبال یک خانه بودید، می‌توانید خانه‌ی خودتان را بسازید و شش هفته صبر کنید. یا این که می‌توانید در یک آپارتمان زندگی کنید و منابعتان را با سایر همسایه‌ها به اشتراک بگذارید. شاید خیلی ایده‌آل نباشد اما کارمان را راه می‌اندازد! و مهم‌تر از آن، نیازی به صبر کردن نیست!

مدت زیادی روی این موضوع کار شد و شرکت‌هایی مانند VMWare بهره‌ی زیادی از آن بردند.

تا این که آدم‌های باهوش دیگری تصمیم گرفتند که فشردن چند ماشین مجازی روی یک ماشین فیزیکی به اندازه‌ی کافی خوب نیست. بهتر است که فرآیندهای بیشتری به شکل فشرده‌تری در منابع کمتری قرار داده شوند!

در این سناریو، خانه همچنان گران است و آپارتمان هم قیمت بالایی دارد .اگر بشود به صورت موقت یک اتاق اجاره کنیم عالی است! هر وقت بخواهم می‌آیم و می‌روم.

به طور خلاصه، کار داکر (Docker) همین است.



تولد داکر

داکر چیز تازه و جدیدی است اما ایده‌ی پشت آن قدمت بالایی دارد. یک سیستم عامل به نام FreeBSD مفهومی به نام زندان داشت که به تاریخ ۲۰۰۰ میلادی بر می‌گردد!

ایده‌ی کار این است که فرآیندهای منحصر به فرد در یک سیستم عامل را ایزوله کنیم که این کار با نام "مجازی‌سازی سطح سیستم" عامل شناخته می‌شود.

نکته :این کار با مجازی‌سازی کامل تفاوت دارد. در مجازی‌سازی کامل ماشین‌های مجازی در کنار یکدیگر روی یک ماشین فیزیکی اجرا می‌شوند.

اما این عبارت در عمل به چه معناست؟

در عمل افزایش محبوبیت داکر تقریبا با افزایش محبوبیت میکروسرویس‌ها – یک رویکرد مهندسی که در آن نرم‌افزار به چندین بخش منحصربفرد تقسیم می‌شود - قابل قیاس است.

و این اجزا نیاز به یک خانه دارند. پیاده‌سازی آن‌ها به صورت مجزا و به عنوان برنامه‌های جاوا یا برنامه‌های اجرایی دودویی دردسر بالایی دارد. روش مدیریت یک برنامه‌ی جاوا در قیاس با یک برنامه‌ی C++ یا گولنگ متفاوت است.

در این شرایط، داکر یک رابط مدیریتی در اختیار مهندسان نرم‌افزار می‌گذارد که به کمک آن می‌توانند برنامه‌های مختلفی را به یک شکل ثابت بسته‌بندی، پیاده‌سازی و اجرا کنند.

بسیار کارآمد است!

خب، وقت آن است که به سراغ مزایا و معایب داکر برویم.

مزایای داکر

ایزوله‌سازی فرآیند

داکر به تمام سرویس‌ها اجازه می‌دهد که به طور کامل ایزوله باشند. سرویس الف در کانتینر مخصوص خود و با تمام وابستگی‌هایش زندگی می‌کند. سرویس ب هم به همین شکل در کانتینر خود و در کنار وابستگی‌های خودش زندگی می‌کند و این دو سرویس هیچ گونه تداخلی با یکدیگر ندارند.

علاوه بر این، اگر یکی از کانتینرها از کار بیافتد، تنها همان کانتینر آسیب می‌بیند و بقیه‌ی آن‌ها بدون هیچ مشکلی به کار خود ادامه می‌دهند.

این کار از جنبه‌ی امنیتی هم مزایای خود را دارد. اگر یک کانتینر آسیب ببیند، خارج شدن از آن کانتینر و هک کردن سیستم عامل پایه بسیار دشوار است (اما غیرممکن نیست!)

در نهایت اگر یک کانتینر رفتار درستی از خود نشان نمی‌دهد (مصرف بیش از حد پردازنده یا حافظه) می‌توان بدون تحت تاثیر قرار دادن سیستم، آن کانتینر را از کار انداخت.

پیاده‌سازی

به این فکر کنید که برنامه‌های مختلف در عمل چگونه ساخته می‌شوند.

اگر این برنامه به زبان پایتون باشد، پکیج‌های مختلفی از پایتون را به همراه خواهد داشت. برخی از آن‌ها به عنوان ماژول pip نصب می‌شوند و برخی دیگر نیز پکیج‌های rpm یا deb هستند. سایر پکیج‌ها نیز فایل‌های نصبی git clone هستند. اگر این کار را در یک محیط مجازی انجام بدهید، این کار با یک فایل زیپ که شامل تمام وابستگی‌ها است صورت می‌گیرد.

از طرفی دیگر، اگر با یک برنامه‌ی جاوا سروکار داشته باشید، شامل یک gradle build خواهد بود که تمامی وابستگی‌های آن درخواست و جمع‌آوری شده‌اند.

منظورم را فهمیده‌اید. برنامه‌های مختلف با زبان‌های مختلف و زمان اجراهای مختلف، در زمان پیاده‌سازی می‌توانند چالش برانگیز باشند.

چگونه می‌توانیم همه‌ی این وابستگی‌ها را ارضا کنیم؟

ضمن این که اگر تداخلی وجود داشته باشد، مشکل دشوارتر هم می‌شود. اگر سرویس الف نیازمند کتابخانه‌ی پایتون نسخه ۱ باشد و سرویس ب وابسته به کتابخانه‌ی پایتون نسخه ۲ باشد چطور؟ در این صورت یک تداخل به وجود می‌آید، چرا که کتابخانه‌های نسخه ۱ و ۲ نمی‌توانند به صورت همزمان روی یک ماشین فعالیت داشته باشند.

در این جا داکر وارد بازی می‌شود.

داکر نه تنها اجازه‌ی ایزوله‌سازی کامل فرآیندها را می‌دهد، بلکه در رابطه با وابستگی‌ها نیز به همین صورت عمل می‌کند. اجرای چندین کانتینر با کتابخانه‌ها و پکیج‌های تداخلی در کنار یکدیگر روی یک سیستم عامل، کاری شدنی و حتی رایج است که با کمک داکر انجام می‌شود.

مدیریت زمان اجرا

همان طور که گفته شد، مدیریت برنامه‌های مختلف به شکل‌های متفاوتی انجام می‌شود. راه‌اندازی و نظارت بر کد جاوا با راه‌اندازی و نظارت بر کد پایتون تفاوت دارد. همین طور پایتون نیز در این زمینه‌ها با گولنگ تفاوت دارد و الی آخر.

داکر به ما کمک می‌کند که به کمک یک رابط مدیریتی واحد، برنامه‌های مختلفی را اجرا، نظارت، متوقف و راه‌اندازی مجدد کنیم. این یک مزیت بزرگ است که تا حد زیادی از سربار اجرای سیستم‌های تولیدی می‌کاهد.

نکته :از دسامبر ۲۰۱۸، دیگر مجبور نیستید که میان شروع سریع و امنیت ماشین‌های مجازی یکی را انتخاب کنید. به لطف آمازون، Project Firecracker سعی می‌کند که بهترین مزایای هر کدام از این دو مورد را ارائه بدهد. البته این تکنولوژی بسیار جدید است و هنوز مناسب پیاده‌سازی برای تولید نیست.

اما در نهایت داکر معایبی هم به همراه دارد.

لامبدا وارد می‌شود

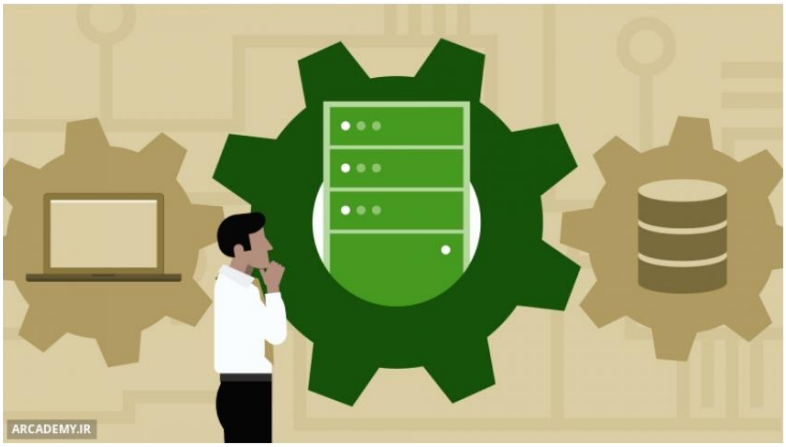
اجرای داکر هم اجرای سرور است. سرورها هم آسیب‌پذیر هستند. باید آن‌ها را مدیریت کرد و به آن‌ها رسیدگی نمود و اهمیت داد.

ضمن این که داکر به طور ۱۰۰٪ ایمن نیست. حداقل به اندازه‌ی ماشین مجازی ایمنی ندارد. دلیلی وجود دارد که شرکت‌های بزرگ کانتینرهای هاست‌شده را درون ماشین‌های مجازی اجرا می‌کنند. آن‌ها دنبال سرعت بالای کانتینرها و امنیت ماشین‌های مجازی هستند!

همچنین در حقیقت هیچ کس داکر را به تنهایی استفاده نمی‌کند. معمولا داکر به عنوان بخشی از فابریک یک کانتینر پیچیده مانند Kubernetes ، ECS، docker-swarm یا Normad به کار گرفته می‌شود. این‌ها پلتفرم‌های پیچیده‌ای هستند که برای اجرا به پرسنل مخصوص خود نیاز دارند (در ادامه بیشتر به این راهکارها می‌پردازیم.)

با این وجود من به عنوان یک دولوپر ترجیح می‌دهم که تنها کد را بنویسم و یک شخص دیگر آن را به جای من اجرا کند. یادگیریDocker ، Kubernetes و سایر این برنامه‌ها کار آسانی نیست. واقعا نیاز است که این کار را انجام بدهم؟

جواب کوتاه این است که بستگی دارد!



برای کسانی که می‌خواهند یک نفر دیگر کدشان را اجرا کند، AWS Lambda مناسب است.

AWS Lambda به شما اجازه می‌دهد که بدون نظارت یا مدیریت سرورها کدتان را اجرا کنید. تنها برای زمانی که در حال پردازش هستید هزینه پرداخت می‌کنید و اگر کد شما در حال اجرا نباشد خرجی برای شما ندارد.

اگر تا کنون نام جنبش بدون سرور را شنیده‌اید، بدانید که همین است! دیگر سروری وجود ندارد که اجرا شود و نیاز به مدیریت کانتینر نیست.تنها کافی است که کدتان را بنویسید و آن را در قالب فایل زیپ به آمازون آپلود کنید و بگذارید آن‌ها سایر کارها را انجام دهند!

علاوه بر این، از آن جایی که لامبداها عمر کوتاهی دارند، چیزی برای هک شدن وجود ندارد. طراحی لامبدا بسیار ایمن است.

عالی نیست؟

البته که هست اما باز هم باید نکاتی را در نظر داشت.

اولا که لامبداها حداکثر برای ۱۵ دقیقه می‌توانند اجرا شوند (از نوامبر 2018). این یعنی که فرآیندهای بلند مدت مانند مصرف کنندگان کافکا یا برنامه‌های محاسباتی نمی‌توانند در لامبدا اجرا بشوند.

دوما، لامبداها تابع به عنوان سرویس محسوب می‌شوند. این یعنی که برنامه‌های شما باید کاملا به میکروسرویس‌ها تجزیه شوند و در کنار سرویس‌های PaaS پیچیده‌ی دیگر مانند AWS Step Functions هماهنگ شوند. هر سازمانی در این سطح از معماری میکروسرویس قرار ندارد.

سوما، برطرف کردن ایرادات لامبداها کار دشواری است. آن‌ها بومی ابری (cloud-native)محسوب می‌شوند و برطرف کردن باگ‌ها در اکوسیستم آمازون انجام می‌شود. این کار معمولا چالش برانگیز و دشوار است.

به طور خلاصه، هیچ وقت کفه‌ی ترازو کامل به یک سمت نیست.

نکته :در حال حاضر راهکارهای کانتینر ابری بدون سرور نیز وجود دارد AWS Fargate . نمونه‌ای از این راهکار است. کلیت ماجرا تفاوت چندانی ندارد. در واقع اگر تازه شروع کرده‌اید، پیشنهاد می‌کنم که حتما Fargate را امتحان کنید. این روش راه آسانی برای اجرای درست کانتینرها است.

جمع‌بندی

داکر و لامبدا دو مورد از رویکردهای مدرن بومی ابری برای بسته‌بندی، اجرا و مدیریت برنامه‌های تولیدی است.

آن‌ها معمولا به صورت مکمل یکدیگر کار می‌کنند و هر کدام مناسب سناریوها و برنامه‌ها تقریبا متفاوتی هستند.

با این حال، یک مهندس دواپس مدرن باید بر هر دوی آن‌ها تسلط داشته باشد .یادگیری لامبدا و داکر می‌تواند اهداف کوتاه مدت یا میان مدت خوبی برای شما باشد.

نکته : تا این جا به موضوعاتی پرداخته‌ایم که از مهندسان دواپس سطح پایین تا سطح متوسط انتظار می‌رود آن‌ها را بدانند. در بخش‌های بعد، تکنیک‌هایی را بررسی خواهیم کرد که بیشتر مناسب مهندسان دواپس سطح متوسط تا سطح بالا است. طبق معمول، هیچ میانبری برای کسب تجربه وجود ندارد!

چگونه در کمتر از شش ماه به یک مهندس دواپس (DevOps) تبدیل شویم؟ بخش پنجم: استقرار

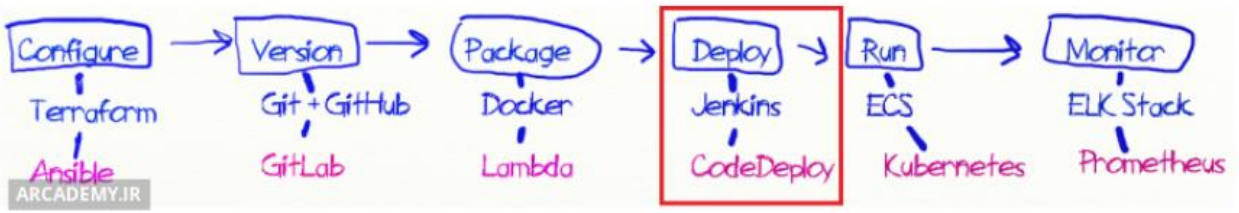
در بخش اول درباره‌ی فرهنگ دواپس و نیازمندی‌های آن صحبت کردیم.

در بخش دوم به راه و روش فراهم‌سازی بستر کد با در نظر داشتن پیاده‌سازی‌های آینده اشاره کردیم.

در بخش سوم درباره‌ی مرتب و منظم نگه داشتن کارها و کدهایتان توضیحاتی ارائه دادیم.

در بخش چهارم نیز نشان دادیم که چگونه کدتان را بسته‌بندی کنید تا استقرار آن راحت باشد.

برای یادآوری بیشتر، الان در این مکان از نقشه‌ی دواپس هستیم.



طبق قرار، اگر به مدت یک ماه برای هر بخش زمان بگذارید، الان باید در ماه چهارم باشیم.

تا اینجای ماجرا می‌دانیم که چگونه زیرساختی که نرم‌افزارمان روی آن اجرا می‌شود را آماده کنیم، می‌دانیم که چطور آن را نسخه‌دهی کنیم و می‌دانیم که به چه شکلی آن را بسته‌بندی کنیم.

حالا زمان آن است که چگونگی استقرار کدمان را بررسی کنیم.

استقرار کد

توجه کردید که نگفتم "چگونه به راحتی کدمان را مستقر کنیم"؟ این حرف دلیل خود را دارد. متاسفانه استقرار درست و حسابی کد از محیط توسعه به تولید یک فرآیند اذیت‌کننده است که با خطاها و مشکلات زیادی همراه است.

اما چرا؟

دلایل متعددی وجود دارد اما به نظر من اکثریت آن‌ها به تفاوت‌ها مربوط می‌شوند. به طور ویژه، تفاوت‌های میان محیطی که کد در آن تولید شده و محیطی که کد در آن اجرا می‌شود.

در واقع بهترین کاری که برای بهبود کلی استقرار کد و حتی زمان اجرای پس از استقرار آن می‌توانید انجام بدهید این است که این تفاوت‌ها را به حداقل برسانید.

بسیار خب، حالا چگونه باید تفاوت‌های میان این محیط‌هایمان را کاهش بدهیم یا از بین ببریم؟

این روی ماشین من کار می‌کرد!

اگر زیرساخت توسعه‌ی شما مانند شکل زیر است

اما زیرساخت تولید شما به این صورت است

شما با مشکل مواجه‌اید.

اگر به جای این که کارها را دستی انجام بدهید، از راهبرد زیرساخت به عنوان کد استفاده می‌کنید، ۹۰٪ از مسیر راه پیش رفته‌اید.

اگر این طور نیست هم نیاز به ناراحتی ندارد. شما تنها نیستید! یه روز وقت بگذارید و کم و کاستی‌هایتان را شناسایی کنید و طبق قاعده به نوبت به آن‌ها رسیدگی کنید.

اما در انتهای ماجرا اگر همچنان کارها را به صورت دستی پیش می‌برید، بعید است که بتوانید در زمینه‌ی مدیریت یک تیم فنی مدرن موفق باشید.

پس اولین کاری که باید انجام بدهید این است که مطمئن شوید هر چیزی که با بخش تولید سر و کار دارد، یک مصنوع نسخه‌گذاری شده است که توسط سرورهای شما استقرار پیدا کرده است.

با فرض این که این کار انجام شده است، من معتقدم که بهترین راه برای استقرار کد این است که کد را مستقر نکنیم.

رویکردهای مدرن برای استقرار کد

درست است – استقرار کد در ماشین‌های تولیدی از دهه ۹۰ میلادی انجام می‌شود.

بزرگترین مشکل مستقر کردن کد در مجموعه‌ای از ماشین‌های تولیدی این است که سرورهای تولیدی شما (محلی که کد اجرا می‌شود) با سرورهای توسعه (محلی که کد نوشته شده است) تفاوت دارند. پس جای تعجب ندارد که پس از استقرار با انبوهی از مشکلات مواجه بشوید که تا کنون وجود نداشته‌اند – همه چیز متفاوت است!

به همین دلیل باید حداکثر سعی‌تان را بکنید تا مطمئن شوید که چیزی که مستقر می‌کنید کل سیستم است، نه تنها یک قطعه کد. به عبارتی دیگر، یک بار کدتان را در محیط توسعه مستقر کنید، سپس کل ماشینی که کد روی آن اجرا می‌شود را کلون کنید و آن را هر جا که نیاز است کپی کنید.

این کار "استقرار تغییر ناپذیر" نام دارد و به شما کمک می‌کند که از سردردهای بی‌اندازه‌ی پس از استقرار دوری کنید.

اگر از کانتینرها استفاده می‌کنید هم همین روش پابرجا است. یک کانتینر را همه جا مستقر می‌کنید.

اما ممکن است بگویید که "تولید من با توسعه تفاوت دارد! نام کاربری و رمز عبور پایگاه داده، استرینگ‌های ارتباطی، مکان‌های S3 Bucket آمازون و غیره. همه‌ی این‌ها تفاوت دارد".

بله، این‌ها متفاوت هستند.

راه حل این مسئله در قاعده‌ی ۱۲ فاکتور تنظیم برنامه نهفته است. تمامی تنظیمات شما باید به صورت خارجی انجام شوند و سپس به عنوان متغیرهای محیطی وارد ماشین بشوند.

به عنوان نمونه، اگر در محیط AWS هستید، از SSM به عنوان انبار پارامتر خارجی استفاده کنید. SSM به زیبایی با Cloud Formation ادغام می‌شود. همچنین ست کردن متغیرهای محیطی به صورت مستقیم با دستورات cli درون aws ssm به آسانی انجام می‌شود. البته سایر سرویس‌دهندگان ابری نیز مکانیزم‌های مشابهی دارند.

علاوه بر این، زمانی که اوضاع آن طور که باید پیش نمی‌رود، از "تعمیر "کردن ماشین تولیدی خودداری کنید. این ماشین‌ها تغییر ناپذیر هستند و این یعنی که هر گونه تعمیری باید تنها از طرف توسعه صورت بگیرد.

در واقع هدف شما باید این باشد که همواره و تحت هر شرایطی، دسترسی به ماشین تولیدی غیرممکن باشد. نه ssh، نه scp و نه دسترسی تولیدی. نه برای شما و نه برای هر کس دیگری.

اما اگر برای پیدا کردن مشکلات به لاگ نیاز داشته باشم چطور؟

درست حدس زدید. لاگ‌های شما باید خارج از ماشین باشند. حالت ایده‌آل این است که لاگ‌ها با برنامه‌هایی مانند ElasticSearch، Logstash، Kibana، SumoLogic یا Datadog به جای دیگری منتقل شوند.

ماشین‌های تولیدی شما مانند یک گله هستند. با کوچکترین نشانی از ناسالم بودن جایگزین می‌شوند. آن‌ها حیوان خانگی نیستند تا ساعت‌ها زمان صرف مراقبت و درمانشان بشود و به سلامت کامل دست پیدا کنند.

نکته :بله، این تشبیه بیش از حد استفاده می‌شود و افرادی که از گله مراقبت می‌کنند به من گفته‌اند که این گونه نیست، اما نکته همچنان پابرجاست. ماشین تولیدی را تعمیر نکنید! ماشین توسعه را تعمیر کنید و مجددا آن را مستقر کنید.

جنبه‌ی فنی استقرار کد

بسیار خب، حالا می‌دانید که باید چه کاری بکنید، اما چگونه؟

متاسفانه در این جا پای Jenkins به وسط کشیده می‌شود. اگر این نام برای شما تازگی دارد، بدانید که Jenkins یکی از محبوب‌ترین سرورهای خودکار استقرار به صورت کدباز است. اما از کلمه‌ی متاسفانه استفاده کردم، چرا که Jenkins همانند محصول ماقبل خود یعنی Hudson، به مدت یک دهه در حال فعالیت هستند و می‌توان این را تشخیص داد.

راه‌اندازی آن بسیار پیچیده است و نگه‌داری از آن حتی پیچیده‌تر است Jenkins ! پلاگین‌های بی‌شماری دارد که قابل اطمینان نیستند و معمولا در حساس‌ترین مواقع از کار می‌افتند و کل سیستم را هم از کار می‌اندازند.

در حقیقت، راه‌اندازی‌های واقعا مقاوم و حرفه‌ای Jenkins بسیار نادر هستند و تنها در بزرگترین سازمان‌ها دیده می‌شوند.

پس با این شرایط چرا Jenkins را به شما پیشنهاد می‌کنم؟

به دلیل این که با وجود تمامی ایراداتش، Jenkins همچنان از محبوبیت بالایی برخوردار است و در صنعت ما استفاده‌ی فراوانی دارد. آگاهی از Jenkins و ساختار Jenkinsfile یک مزیت بزرگ برای شغل شما به حساب می‌آید و نباید از آن چشم‌پوشی کرد.

به خاطر داشته باشید که در زمان یادگیری Jenkins از مسیر جدیدتر BlueOcean پیروی کنید، نه از مسیر قدیمی Jenkins Job.

این اهمیت بالایی دارد. چرا که شما می‌خواهید پایپ لاین CI/CD شما در مخزن گیت کدتان حاضر باشد. در این صورت خود پایپ لاین به یک قطعه‌ی نسخه‌گذاری شده از کد تبدیل می‌شود.

این به حدی مهم است که دوباره تکرار کردن آن ارزشش را دارد.

همه چیز کد است.

برنامه‌ی شما، طریقه‌ی استقرار آن، روش نظارت بر آن، روش تنظیم آن، هر چیزی که فکرش را بکنید قطعه کدی است که در گیت‌هاب یا گیت‌لب یا هر جای دیگری قرار دارد و به شکل مناسب نسخه‌دهی شده است.

در این جا هدف این است که یک محیط واقعا آسیب ناپذیر را برای دولوپرهای مرکزی فراهم کنیم.

برای مثال، من باید بتوانم کد میکروسرویس کوچکم را بنویسم، هر آزمایشی که به نظرم مناسب است را اضافه کنم، یک Jenkinsfile اضافه کنم، تنظیمات نظارت به عنوان کد را اضافه کنم، پارامترهایم را در یک فایل "env.yaml" تعیین کنم، همه این‌ها را در یک مخزن ذخیره کنم، اجازه بدهم که Jenkins مخزن نام برده را به طور خودکار پیدا کند، آن را بسازد، آزمایش کند و مستقر کند و در پایان کار با یک ایمیل به من اطلاع بدهد!

هدف ما این است! در واقع ماموریت اصلی و اساسی مهندسان دواپس همین است.

جایگزین‌هایی برای Jenkins

همان طور که اشاره کردم، مدت زیادی است Jenkins در این عرصه حضور دارد و الان به نظر من گزینه‌های بهتری هم وجود دارد، حتی اگر از محبوبیت کمتری برخوردار باشند.

یکی از آن‌های سرویس AWS CodeDeploy است. این سرویس محدودیت‌هایی دارد اما دولوپرهای CodeDeploy در سال قبل بهبودهای قابل توجه‌ای اعمال کرده‌اند و اگر در محیط AWS هستید شدیدا توصیه می‌کنم که این مورد را امتحان کنید.

یکی از موارد دیگر GitLab CI است. اگر سازمان شما بر روی GitLab فعالیت می‌کند، بهتر است که از همین مورد کارتان را شروع کنید، چرا که به خوبی با سایر بخش‌های GitLab ادغام شده است.

در انتها نیز گیت‌هاب Actions را معرفی کرد که متعلق به خودش است.

حقیقتش را بخواهید فکر نمی‌کنم که در این جا ابزار مورد استفاده آن چنان حائز اهمیت باشد. چیزی که اهمیت دارد این است که بدانید همه چیز، حتی پایپ لاین‌های استقرار کد شما، مصنوعات نسخه‌دهی شده‌ای هستد و این که هیچ چیز وارد بخش تولید نمی‌شود، مگر این که از سمت توسعه وارد شود.

در هر صورت، اگر کارتان را با Jenkins شروع می‌کنید، آن را به عنوان یک کانتینر راه‌اندازی کنید. کار بسیار دشواری نیست و فرصت یادگیری خوبی برای سر در آوردن از چگونگی استقرار یک سرور کانتینر شده‌ی Jenkins با نودهای کارگر کانتینر شده‌ی Jenkins است.

در واقع می‌توانید کارتان را آسان‌تر و بدون هیچ کانتینری شروع کنید که موضوع پست بعدی ما خواهد بود. با ما همراه باشید!

در حال حاضر تا این بخش بود و در آینده ممکن است بخش بعدی را آماده کنند.

https://arcademy.ir